ABSTRACT

DEY, SUMON. Design of a Scalable, Configurable, and Cluster-based Hierarchical Hardware Accelerator for a Cortically Inspired Algorithm and Recurrent Neural Networks. (Under the direction of Paul D. Franzon.)

Machine learning algorithms based on deep learning have met with enormous success to achieve higher performance in applications ranging from object recognition to defeating a human expert in the complex game. Furthermore, it can broaden the horizon by processing a massive amount of multimodal natural data (video, audio) and learning useful join representations in applications. In addition to deep learning, cortical learning algorithms can also learn representations in much closer to in a way a human brain works. Unlike the use of dense data in deep learning, binary data is used in cortical learning to model sparse distributed memory for different representations. These algorithms use artificial neural networks to model them into hardware. However, implementation of such networks in hardware relies on throughput and memory bandwidth of hardware architecture, which requires dealing with massive amount of data. To advance the research of these rapidly evolving techniques, there is a direct need for the design and implementation of specialized hardware to accelerate these algorithms.

In this work, a scalable, configurable, and cluster-based hierarchical hardware accelerator is designed and implemented through an application-specific integrated circuit (ASIC) for Sparsey, a cortical learning algorithm. Also, an application-specific instruction set processor (ASIP) is designed and implemented for recurrent neural networks (RNNs). A distributed on-chip memory organization is designed and implemented in ASIC to improve memory bandwidth and accelerate the memory read and write operations for synaptic weight matrices. A bit-level data process from memory, storage, and special multiply-accumulate hardware are implemented for multiply-accumulation operations. The fixed-point arithmetic and fixed-point storage are also adapted in ASIC implementation. At 16nm, the ASIC of Sparsey achieved an overall speedup of $25.24\times$ and $353.12\times$ reduction in energy per frame, and $1.43\times$ reduction in silicon area against a GPU. In ASIP, the emerging 3D-stacked memory is used to increase the off-chip memory bandwidth and sized on-chip memory to improve data locality inside the processor. A set of short instructions are also implemented in ASIP architecture after analyzing different complex, time-consuming, special operations into high-level functional blocks, and a look-up table based special function operations to improve its performance. State-of-the-art mixed precision training and inference are also adapted in this architecture. A high-level programming environment is also developed to generate Very Long Instruction Word (VLIW) instructions for ASIP to process a variant of RNNs. At 16nm, an ASIP achieved $1.5\times - 5.6\times$ faster processing, $4.3\times - 40.8\times$ reduction in energy per sequence, and $1.5\times$ area benefit than a GPU.

Design of a Scalable, Configurable, and Cluster-based Hierarchical Hardware Accelerator for a Cortically Inspired Algorithm and Recurrent Neural Networks

by
Sumon Dey

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

Computer Engineering

Raleigh, North Carolina

2019

APPROVED BY:

Winser Alexander	Dror Baron
Eric Laber	Paul D. Franzon Chair of Advisory Committee

DEDICATION

To my family.

BIOGRAPHY

Sumon Dey was born in September 1980, Mymensingh, Bangladesh. He received the bachelor's degree in Electrical and Electronic Engineering from Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 2004, and the master's degree in Electrical and Computer Engineering from University of Illinois at Chicago (UIC), USA, in 2012. After graduating from BUET, he worked as a Network Switching Subsystem Engineer from 2004 to 2010 on GSM Network Architecture, OSI Reference Model, Interconnection, and Signaling Protocols at Telecommunication Industry, Dhaka, Bangladesh. During his masters, he worked as a research assistant at the Nanoengineering Research Laboratory (NRL), Chicago, USA. He began work towards his Ph.D. in Computer Engineering in Spring 2013.

During his Ph.D. he worked as a graduate intern on system identification through machine learning at Hewlett Packard Enterprise (June 2017 – August 2017). His research interests include the design of application-specific processors, cortical processors for different cortical learning and deep learning algorithms.

ACKNOWLEDGEMENTS

Joining the Ph.D. program in the Department of Electrical and Computer Engineering at North Carolina State University was one of the best decisions of my life. I am especially indebted to my advisor, Paul D. Franzon, who has provided the opportunity to work with him. I am also thankful to him for introducing me to research centered on building hardware for next-generation artificial intelligence. He is an excellent advisor who gave me continuous support of my Ph.D. studies and research, gave me the freedom and encouragement to pursue my ideas and goals. More importantly, his patience, enthusiasm, and knowledge helped me throughout my graduate studies. I am thankful to Winser Alexander for his guidance and encouragement during my teaching assistantship of his course. I would like to thank Winser Alexander, Dror Baron, and Eric Laber for serving on my committee and agreeing to read my dissertation so quickly.

I was also fortunate to be a part of Cognitive Computing research group at Microelectronics Systems Laboratory (MSL) and would like to thank my colleagues, Steve Lipa, Lee Baker, Josh Schabel, Weifu Li, Josh Stevens, and Zachary Johnston for stimulating discussions, debates and continuous support for my research. A very special thanks goes to Steve Lipa for all of his help and support on design kits. My sincere thanks also go to Josh Schabel for giving me support on design kits, scripts, and helping me debug issues of different design flows. I thank Lee Baker for brainstorming discussions, exciting research debates, and also for benchmarking Sparsey on parallel computing platform through software implementation. I thank Weifu Li for discussing various properties and implementation aspects of the Hierarchical Temporal Memory algorithm. I would also like to thank Gerard (Rod) Rinkus, President of Neurithmic Systems, for his interesting discussions and continuous updates on Sparsey.

I am thankful to my undergraduate buddies who always showed their support in different ways. I would also like to thank my friends in Raleigh for good times, laughs and making my stay so enjoyable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Machine learning algorithms have enjoyed enormous success in many modern applications ranging from healthcare to e-commerce to social network, etc [1, 2, 3, 4]. The most common form of machine learning is deep learning, which can process natural data. In recent years, deep learning algorithms have outperformed most other machine learning algorithms especially in pattern-recognition, natural language processing, navigation (self-driving cars), and cybersecurity [5, 6, 7, 8, 9]. Deep learning using deep neural networks (DNNs) have achieved revolutionary success in learning and recognition of spatial context and temporal sequences, respectively, from raw natural data [5]. Over the last few years, the accuracy of these applications has rapidly improved by using larger network sizes. The classification accuracy of visual object recognition (top-1 accuracy) has increased 62.5% in 2012 to 80.0% in 2016 using AlexNet and Inception-v4, respectively, on ImageNet dataset [10, 11]. On the other hand, the automated translation (English-to-French, English-to-German) system used the deep Long Short-Term Memory (LSTM) and reduced the translation error by 60% compared to the Google's phrase-based system [12, 13, 14]. However, these algorithms work only on labeled data provided that the training set is labeled with the expected prediction. For example, each photo of a cat presented to a DNN is clearly labeled "cat", which referred to as supervised learning. Cortical learning algorithm (CLA), that can learn with minimal or no supervision on lightly labeled data or completely unlabeled data, are gaining interest. These algorithms process data in a manner much closer to the way a human brain learns. The recent implementation of these algorithms shows promising success and getting much attention for its ability to understand the spatio-temporal pattern [15, 16].

However, implementation of such deep neural networks rely on high throughput and memory intensive hardware architecture to deal with massive amounts of computation and data. The accuracy improvement of these models also comes at the cost of high hardware resources. The inference accuracy improvement (approximately 17.5%) using a larger model (AlexNet to

Inception-v4) increased the computation 16.14 times (2 billion to 22.6 billion operations) with slightly decreased the parameter sizes (approximately 2.5 times) where the size of a single image is 224×224 (real-life image size 1600×1200) [10, 11]. The increased model size also increases the inference time per image 18ms to 150ms (approximately eight times) on NVIDIA K40 GPU [17]. The inference accuracy improvement of machine translation (English to France) using LSTM also comes at the price of a network with huge parameters (384 million) and computation is approximately 4 TOPS to process words [14, 18]. However, the number of computations is highly dependent on the mini-batch size and the hidden size. In addition to computation, the network parameters have also increased 88 times (4.3 million to 384 million) from 2013 to 2015 in language processing task [6, 18]. The training of image recognition and machine translation models also takes between six to ten days and involves multiple graphics processor units (GPUs) to enhance data storage and memory bandwidth [10, 14]. On the other hand, the tiny version of a cortical algorithm implements 7×10^4 neurons with 1.54 million synapses, as compared to the human brain has approximately 2×10^{10} neurons and approximately 5×10^{14} synapses. This small model also requires approximately 3.08 million operations to process a single 16×24 image [19].

In addition to computation and memory resources, power consummation is another key challenge that limits the implementation of these networks in hardware. In CLA, parameters of the network are binary, sparse and require less storage compared to the dense and floating-point parameters. However, the number of neurons is much higher than that of RNNs [20]. Thus the use of on-chip memory for synaptic weights can reduce the access energy, in other words, meet the power budget of the hardware. In contrast to CLA, RNNs use to model deep learning algorithms have a smaller number of neurons with large dense floating-point parameters, which requires the use of the off-chip memory of GPUs to store parameters (samples and weights) and access huge amount of data for training and inference. Running these networks on multiple GPUs (up to 8) requires almost 235W to 2000W in power consumption [10, 14]. Also, GPUs are designed for general purpose computation and not optimized for the performance of these algorithms. The overall power overhead of these networks on general purpose hardware is significant. Therefore, specialized hardware can benefit enormously by improving the power budget of these algorithms.

From these perspectives, an effort to design a custom hardware of Sparsey, a cortical learning algorithm, through Application-Specific Integrated Circuit (ASIC) catering to the distributed on-chip storage needs for synaptic weight matrices and tailoring the aggregated memory bandwidth of distributed memory, and access energy of on-chip storage. The custom hardware of Sparsey achieved 25.24× acceleration and 353.12× better energy efficiency compared to a Tesla P100 GPU on KTH action recognition dataset [21]. In addition, an Application-Specific Instruction Set Processor (ASIP) is designed to support state-of-the-art variants of recurrent neural networks (RNNs) using the emerging high bandwidth and low energy off-chip 3D memory. The ASIP solution is designed in a way to support both training and inference of RNNs and potentially others. The hardware is used to process the popular Google's Neural Machine Translation System. It performed training and inference for different batch configuration and achieved an overall $6.5\times$ – $61.0\times$ performance improvement than a Tesla P100 GPU [22].

1.1 Motivation

One algorithm that can learn in unsupervised or lightly supervised mode is the Neurithmic's Sparsey algorithm [23]. Sparsey is a scalable online probabilistic cortical algorithm inspired by the cortical column of a brain and seeks to mimic the essential computational properties of the neocortex. The initial implementation of Sparsey achieved 91% recognition accuracy on MNIST dataset [19, 24, 25]. The primary goal of Sparsey is to model human episodic memory, that is, the memory of specific events. It memorizes patterns in a sparse distributed representation (SDR) network by increasing the connections – the weights in an SDR are binary. In Sparsey, any input pattern activates only a small fraction of the total neurons. It is anticipated that Sparsey would not perform well on GPUs because GPUs are optimized for computations of dense matrices on floating-point data and Sparsey requires massive multiply-accumulate (MAC) operations on sparse binary data.

Sparsey uses a neural network where the total number of neurons is large and while only a fraction of total neurons are active, every neuron needs to access its own binary weight matrices in each neural cycle. This computation requires a large memory bandwidth to obtain synaptic weights simultaneously whereas GPU has limited on-chip storage with huge memory bandwidth but a substantial off-chip memory with limited memory bandwidth. The total size of weight matrix of a single neuron is relatively small (compared to a floating-point weight matrix) and can fit in the on-chip SRAM, which is faster to access (increases memory bandwidth) than the off-chip memory of GPUs. Sparsey also learns in an online manner, which demands high-speed processing for each incremental training cycle (approximately 16 ms/frame) in hardware.

To map sizeable Sparsey networks onto GPUs requires significant memory bandwidth to read (or write) the off-chip weight matrices and increase the energy consumption. The memory access rate is the bottleneck in Sparsey and dominates the overall power consummation and limits the performance needed to implement the sizable Sparsey network on a GPU. The access energy can be reduced approximately 130 times (640 pJ/word to 5 pJ/word in 45nm technology) using on-chip SRAM [26].

Sparsey is designed to mimic the human brain, which can have 100 billion neurons. To implement a cortex model in hardware, Sparsey requires a large silicon area. As mentioned in the previous section, Sparsey is dominated by binary MAC operations instead of traditional floating-point operations. However, each neural cycle also requires some floating-point operations and special function operations. The fixed-point hardware implementations of these floating-point operations and special function operations can decrease the overall area and power (13.6× and 21.5× in 45nm, respectively) and helps to accommodate more neurons in silicon [26]. The primary motivation of this work is to estimate the meaningful upper bound performance of an ASIC compared to the peak hardware performance of state-of-the-art GPUs.

Performance and flexibility are the two significant requirements of hardware architecture needed to implement Convolutional neural network (CNN) and different variants of recurrent neural network (RNN). The architecture community has mainly focused on CNN, with little attention to RNNs or MLPs, where CNN only process tiny portion of the workload in the complete system [27]. Indeed, most of the hardware research on RNNs has focused on the inference engine where training is still heavily involved with multiple GPUs. One of the primary motivations of ASIP accelerator of RNNs is to fill that research gap and implement flexible hardware for training as well as inference.

The training of state-of-the-art RNNs runs on multiple GPUs and takes more than a week. A single forward propagation of training involved an estimated 2 TFOPS and 12 Tb/s memory bandwidth [14]. The parameters of RNNs are floating-point, dense matrices and cannot be stored in on-chip. Thus, the computations are mostly depended on the off-chip memory bandwidth. The appropriate level of floating-point hardware resources is also required to accelerate the training and inference for different hyperparameters.

State-of-the-art RNNs have millions of parameters (384 million) and requires massive amount of storage and computational resources for training and inference [13, 14]. To fetch 384 million parameters (approximately 12 Gb) from off-chip memory requires almost 14.6 W of power considering conditions at 60 fps and 640 pJ access energy [26]. The training dataset also needs to access from off-chip memory during training. The computation of training is usually done in single-precision floating-point to maintain training accuracy, which also increases the overall power consumption in GPUs or similar hardware platforms.

Flexibility of hardware is essential because RNN has different variants and the data flow of different deep learning networks can be different. In addition, an application specific RNN shows better performance accuracy compared to generic RNNs. Thus, a new hardware solution is required to improve training time, react in a faster manner to real-life inputs, and keep up with the change of algorithms and network variants. Flexibility of hardware is also required to map networks into hardware in a different fashion (model parallelism, data parallelism, etc.) and achieve higher performance [28, 29].

Another critical challenge of RNN in hardware is the area. As discussed, hardware of RNNs requires a massive amount of off-chip storage and cell area of off-chip memory as well as integration (2D or 3D) plays a vital role in the overall area. The half-precision floating-point arithmetic is area and power efficient (14.45× and 7.6×, respectively) and significantly decreases the total area budget [26]. Unfortunately, half-precision training loses the training accuracy (convergence) for some variants of DNNs. However, the mixed precision training, combination of single-precision and half-precision, can maintain training accuracy as good as single-precision in hardware and also can reduce the area of silicon [30].

The data flow of RNNs is another important reason to design hardware. The training of RNNs has a pattern of irregular data access from the off-chip memory (i.e. DRAM) instead of contiguous behaviors. The irregular pattern of memory access makes DRAM much harder to operate on burst mode and maintain the highest DRAM bandwidth during training. Thus, mapping the network parameters to a DRAM and associated functionality in the hardware could increase the memory bandwidth and accelerate the training of RNNs.

These are the motivational factors which led to the design of custom hardware (ASIC) for Sparsey and an application-specific instruction set processor (ASIP) for RNNs targeting both training and inference.

1.2 Contribution and Outline

A benchmark of a three-layer Sparsey of 336×10^3 neurons is implemented in a GPU-accelerated parallel computing platform that analyzes different aspects to optimize algorithm into custom hardware. The key contributions of this work are the following innovations specifically designed to create an efficient engine on which to execute Sparsey:

- A distributed on-chip memory technique that consists of storage associated with each neuron for synaptic weights and on-chip shared memory, which increases the bandwidth utilization 91.75× compared to a GPU.
- A special multiply-accumulate (MAC) technique that consists of binary dot product followed by mixed-width integer accumulations, dedicated to the on-chip memory of neurons, which efficiently utilize the bandwidth of on-chip memory and accelerate the hotspot operation of Sparsey.
- Fixed-point custom hardware for arithmetic, a special function unit (SFU), and network-on-chip (NoC) based on data flow of Sparsey that can further reduce the area and power of the custom hardware.
- An in-depth scalability analysis of the performance, communication, and power-budget of the hierarchical architecture across a range of accelerators (or clusters).

- An efficient custom hardware of Sparsey that can learn and recognize events of KTH action recognition dataset and achieve 25.24× speedup, 1.43× reduction in area, and 353.12× energy efficiency against a Tesla P100 GPU.

A novel application-specific instruction set processor (ASIP) is designed analyzing different aspects of the recurrent neural networks to improve performance on hardware. The contributions of this work are:

- An emerging 3D-stacked memory to increase the off-chip memory bandwidth and a sized on-chip memory to enhance data locality for reducing long latency off-chip memory access and accelerating the algorithm.
- A set of short instructions designed after analyzing data flow of different complex, time-consuming, special operations into high-level function blocks for training and inference of RNN variants to improve the flexibility of hardware.
- A state-of-the-art training technique, called mixed precision training that consists of half-precision multiplier followed by the single-precision adder, which can improve the hardware performance maintaining the training accuracy similar to the single-precision.
- A look-up table (LUT) technique, a combination of on-chip memory based LUT and fast numerical method and associated high-level instructions, that can compute different operations (divider, square root, log) and also non-linear activation (Sigmoid, Tanh, etc.) to improve the performance in hardware.
- A high-level programming environment to generate Very Long Instruction Word (VLIW) instructions for this ASIP architecture and process RNN variants.
- An efficient ASIP architecture that can perform both training and inference of RNNs and MLPs by increasing the memory bandwidth and parallelism at different levels. This work processed training and inference of language modeling task, and achieved 1.5× − 5.6× processing speedup, 4.3× − 40.8× reduction in energy, and 1.5× area reduction than a Tesla P100 GPU.

The remaining chapters of this dissertation are organized as follows:

Chapter 2 provides the fundamentals of neural networks, the learning (or training) algorithms of Sparsey and Long Short-Term Memory (LSTM), one of the most popular and successful variants of RNN. The most recent real-world applications of LSTM and success stories of Sparsey are also discussed briefly in this chapter.

Chapter 3 surveys the related hardware platforms to accelerate Sparsey and LSTM. Different types of hardware solution of these neural networks are discussed in the respective sections.

Chapter 4 presents the ASIC solution of Sparsey. This chapter starts with a discussion of the various algorithmic aspects of Sparsey in hardware needed to consider during designing of the custom hardware. The implementation details and experimental setup are presented in the subsequent sections. The speedup and energy efficiency of ASIC compared to a GPU are presented in the result section. The scalability studies are shown in the discussion section.

Chapter 5 presents the ASIP solution of RNNs. The algorithmic aspects, mapping networks in hardware, memory access pattern of RNNs are analyzed before giving the details of hardware implementation. It also discusses the experimental setup to estimate performance improvement against a GPU. This chapter also discusses the flexibility and scalability of the proposed ASIP architecture.

Chapter 6 summarizes the dissertation by discussing the lessons learned, the future work of deep learning accelerators, and presents a conclusion.

1.3 Abbreviations

ML Machine Learning

DL Deep Learning

CLA Cortical Learning Algorithm

AN Artificial Neuron

ANN Artificial Neural Network

RNN Recurrent Neural Network

MLP Multilayer Perceptron

CNN Convolutional Neural Network

DNN Deep Neural Network

GRU Gated Recurrent Unit

LSTM Long Short-Term Memory

VLIW Very Long Instruction Word

SDR Sparse Distributed Representation

SDM Sparse Distributed Memory

SDC Sparse Distributed Code

Mac Macrocolumn

CM Competitive Module

CSA Code Selection Algorithm

SISC Similar-Inputs-to-Similar-Codes

FC Fully Connected

GD Gradient Descent

SGD Stochastic Gradient Descent

BPTT Backpropagation Through Time

FV Feature vector

MAC Multiply-Accumulate

SRAM Static Random-Access Memory

DRAM Dynamic Random-Access Memory

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction Set Processor

GPU Graphics Processing Unit

TPU Tensor Processing Unit

FPGA Field-Programmable Gate Array

PE Processing Element

NoC Network-on-Chip

SFU Special Function Unit

DSDR Dual Single Data Rate

RBM Restricted Boltzmann Machine

Chapter 2

Background

The fundamental of neural networks, target neural networks, how they work, and some recent applications are discussed in this chapter. Both cortical learning and deep learning algorithms use an artificial neural network (ANN) to solve various machine learning tasks. An artificial neural network is a collection of neurons (or compute unit) that are connected in various fashions. The fundamental properties (number of neurons, storage, connection types, etc.) of ANN varies depending on whether it is used for cortical learning or deep learning applications. The cortical learning uses ANN to build a mathematical model (sparse distributed memory) of the neocortex and uses that model to learn (or recognize) events. This type of ANN usually uses binary connections, and a massive number of neurons. However, at one time, only a few fractions of neurons are active (or fired) to generate sparse binary code for Hebbian learning [20, 31]. It also uses multiple layers of neurons stacked vertically in what is called deep neural network (DNN); the number of multiple layers is limited by the layers of the neocortex (usually 6) [31]. On the contrary, DNNs inspired by deep learning use multiple layers (can be more than 6) of neurons, but there are fewer number of neurons per layer than that of CLA [10, 11]. In addition, DNN uses float connections between neurons and float vector to represent an event. These types of DNNs often works on labeled data and requires an optimizer (Gradient Descent) to learn the strength of connections [32].

2.1 Fundamental of Artificial Neural Networks

The fundamental of a neuron, neural networks, different aspects of the neural network, and various types of neural networks are discussed in this section. The nervous system of humans is a complex neural network consisting of nearly 2×10^{10} biological neurons that are connected to each other through synapses. The biological neuron composed of a body, an axon, and multiple dendrites is shown in Figure 2.1.

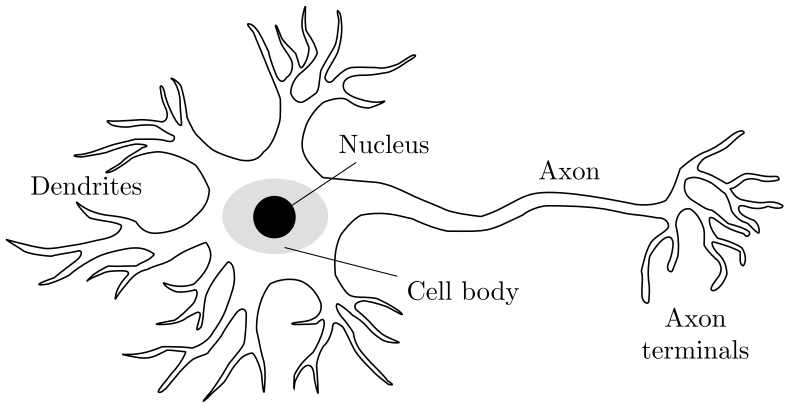

Figure 2.1: A biological neuron

Neurons communicate with each other through synapses that usually formed between axon terminals (multiple terminals of an axon) of one neuron and dendrite of another neuron. The cell body of a neuron performs the summation of incoming signals from synapses and fires (send an impulse to other neurons through axon) only if the summation reaches its threshold value.

A mathematical model of a biological neuron, artificial neuron (AN), is shown in Figure 2.2. An artificial neuron (AN) is the basic building block or compute unit of an ANN. It consists of three primary components that include synaptic weights, a threshold, and an activation function. The synaptic weights (W) are factors associated with each input and determine the strength of the input vector (X) by multiplying the weight matrix and input vector (W^TX). There is an internal threshold in AN, called bias, added to the weighted-input (W^TX) and can affect the activation of the output. The weighted-input is then passed through an activation function to produce the output and transmitted to other neurons similar to the way the biological neuron propagates the summation of the incoming signal to other neurons through axon. There are different types of activation functions (Sigmoid, Tanh, ReLU, etc.) used in ANNs and the primary purpose of using activation function is to introduce non-linear properties in ANN. In ANN, there is one input layer, one output layer, and one (or multiple) hidden layer(s). DNNs are simply neural networks with more than one hidden layer. For simplicity, the rest of the dissertation will refer to artificial neuron (AN) as neuron and artificial neural network (ANN) as neural network.

There are different types of neural networks depending on their architecture and connectivity. The most common and successful neural networks are multilayer perceptron (MLP), convolution neural network (CNN), and recurrent neural network (RNN). An MLP is a fully-connected, feed-forward neural network where there are connections from every neuron of $Layer_i$ to every neuron of $Layer_{i+1}$. The data flow of MLP is in one direction (input to output). Google's

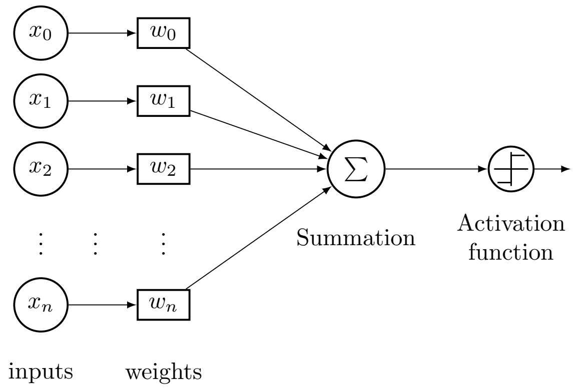

Figure 2.2: A perceptron or artificial neuron

Tensor Processing Unit (TPU), deployed in the datacenter, represents 61% of its workload for MLP [27]. In addition to MLP, there is another type of feedforward neural network, Convolutional Neural Network (CNN, or ConvNet), are mostly applied to analyzing visual imagery. Unlike MLP, a CNN has spatial locality of input signals and shared weights in space which makes the number of weight parameters much smaller than a fully-connected layer. Due to data locality and reusability in CNN, it is usually computation bound rather than data bound. Hardware implementation of CNN in TPU shows only 5% of Google's datacenter workload [27]. A special non-linear activation function, which is computationally less expensive in hardware called Rectifier Liner Unit (ReLU) is often used to accelerate MLP and CNN in hardware [27, 33]. Both MLP and CNN map input to output (or activations), which is known as feature mapping or activation mapping. A different type of neural network, Recurrent Neural Network (RNN), can map the history of previous inputs to output activations and can be applied to solve sequential prediction problems (Speech, Video). Two types of connections (feed-forward and feedback) are usually present among neurons in RNN. However, there are many variants of RNN based on their connectivity. In RNN, there are internal states or memories that used to process or summarize the history. The feedback connections that are ubiquitous in the human brain are inspired researcher to model neocortex using different variants of RNN and performs sequence processing tasks similar to the human brain. From the hardware perspective, operations per weight are lower in RNN, which makes it memory bound rather than computation bound. Thus, the implementation of RNN in hardware is less efficient because hardware is relatively cheap for computation than fetching data. Long Short-Term Memory (LSTM), a popular variant of RNNs, implemented on TPU processes 29% of Google's datacenter workload [27].

Learning of these DNNs can be performed either in supervised or unsupervised manners. Most of the DNNs use supervised learning where it maps input to output based on "labeled" training data (examples of an input-target pair). Some variants of these DNNs can also learn in

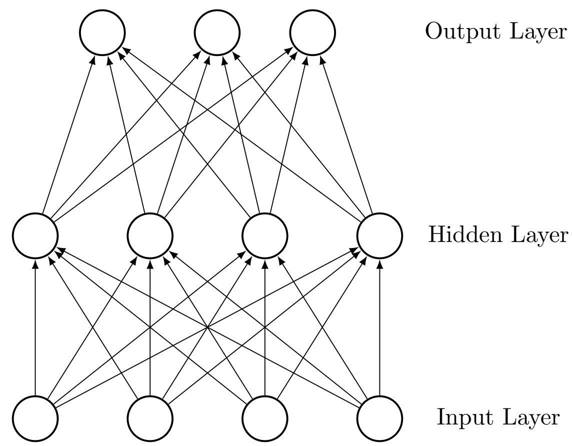

Figure 2.3: A multilayer perceptron (MLP) neural network

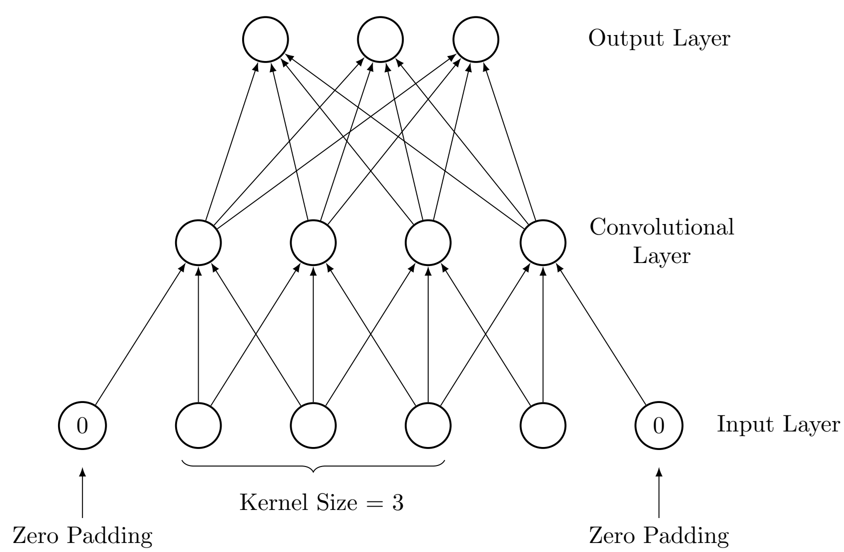

Figure 2.4: A 1D convolutional neural network (CNN) for stride of one

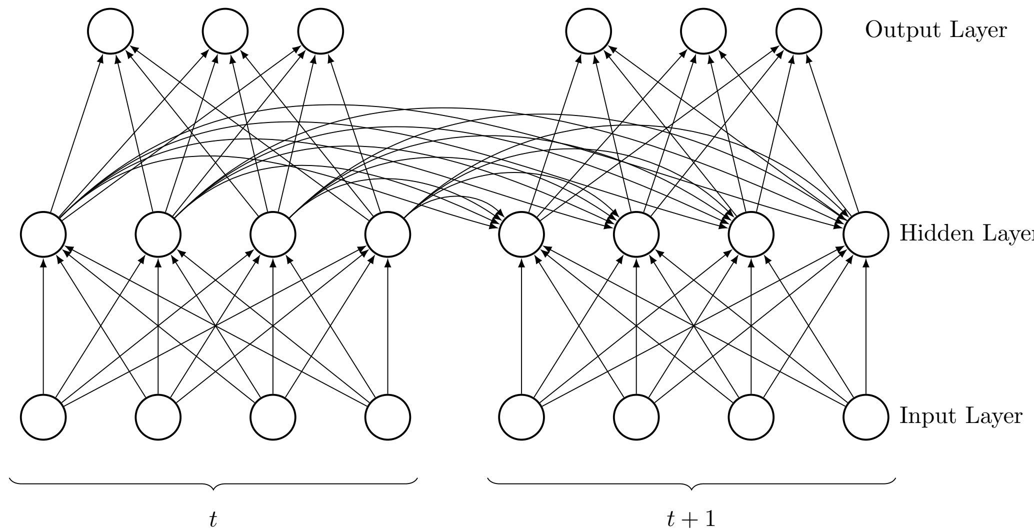

Figure 2.5: A recurrent neural network (RNN)

an unsupervised manner based on "unlabeled" data. For instance, Auto-encoders and Generative Adversarial Networks (GANs) are two useful variants of CNN that perform unsupervised training [34, 35]. Unsupervised learning is also used to train LSTM for discriminating different types of sequences [36]. In addition, brain-inspired recurrent networks (Hierarchical Temporal Memory (HTM), Sparsey, Cogent Confabulation) also use unsupervised learning on binary input data [20, 37, 38]. The basic connectivity of MLP, CNN and RNN are shown in Figure 2.3, Figure 2.4, and Figure 2.5, respectively. In these figures, RNN clearly shows a much higher number of connections than others.

2.2 Target Neural Networks

The target neural network architectures of hardware implementation are discussed in this section. A brain-inspired neural network (Sparsey) can learn unsupervised, and a successful variant of RNN, LSTM, can learn in supervised fashion are chosen for hardware acceleration.

2.2.1 Sparsey

This section describes the basic concepts and important properties of Sparsey network as well as the computational steps of Sparsey's core learning algorithm, the code selection algorithm (CSA). Similar to other neural networks, a Sparsey network has an input layer, an arbitrarily many internal layers, and an output layer. These layers are mapped in the similar way as is in the physical layout of the human brain [37].

Basic Concepts of the Sparsey Network

The neuron is the basic computational unit of the Sparsey network just like other neural networks. A single neuron has three types of connections (bottom-up (U), horizontal (H), and top-down (D)) where connections (or synapses) have binary weights. The signal propagating in the bottom-up and top-down connections carry the spatial information and the horizontal connections carry the temporal information. The weights of connections are updated using the Hebbian learning rule [37]. There is also an activation function (usually Sigmoid) associated with each neuron. In Sparsey, neurons are arranged in minicolumns, minicolumns organized in macrocolumns, and macrocolumns in layers. The minicolumn is considered analogous to a specific portion – the pool of layer ∼20 2/3 pyramidal neurons – of the human cortical minicolumn [37]. Minicolumns and macrocolumns are also referred to as competitive modules (CM) and mac (Mac), respectively. A three-layer Sparsey network is shown in Figure 2.6.

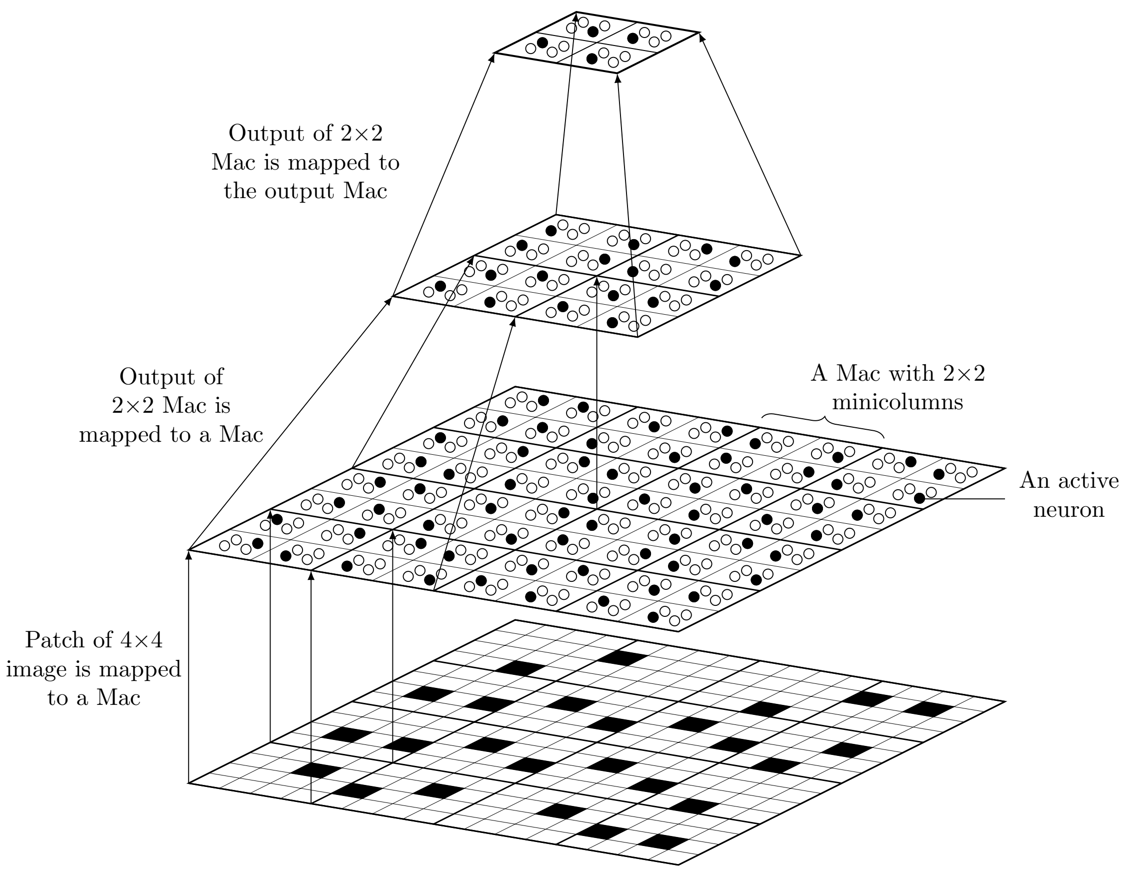

Figure 2.6: A three-layer Sparsey network

Properties of Sparsey Network

The interesting properties of Sparsey are: 1) it has similar-inputs-to-similar-codes (SISC) property i.e. it learns in an unsupervised manner and uses CSA to select highly intersecting sparse distributed codes (SDCs) for matching to similar inputs; 2) it forms long-lasting memory traces of sequences based on single unsupervised learning of each sequence; 3) the number of computational steps needed to either learn a new frame or retrieve (recognize) a previously learned frame remains constant for the life of system regardless of the information stored in the system; 4) it can recognize an individual training sequence (exactly or slightly perturbed version) prompted with the first frame of the sequence; and 5) it can recognize the novel sequences compared to the closely approximated versions of known sequences.

Algorithms of Sparsey

The core algorithm of Sparsey has two different modes of operations: 1) learning or training mode; and 2) inference or recognition mode. The CSA choose one neuron from each CM to generate an SDC corresponding to Sparsey's input feature vectors during both learning and inference mode.

In learning mode, a Mac becomes active if either the number of active features (or active bits) of bottom-up connections lies within specified limits or it is already active for the number of frames that is less than its persistence. The persistence is the activation duration of Mac in each layer, which is usually increased in upper layers of the model. Every neuron (K) of a layer performs the summation of weighted inputs independently for different types of connections (bottom-up, horizontal, and top-down) followed by normalization each summation. Neurons then compute in a three-way match (the overall local measure of support (V)) for different input sources followed by calculating the highest local support in each CM (Q). The highest local support is used to calculate various parameters (η, G, etc.) of the non-linear activation function. The non-linear activation function, Sigmoid, is used to calculate the relative probability of winning (ψ) for each neuron before the softmax operation. A winner neuron is selected from each CM using a random number determined by applying the probability distributions (ρ) of neurons' winning in that CM. The learning rule is applied updating the weights leading to and from the winning neurons [37]. In Sparsey, the freezing of learning, based on a threshold, is applied to each Mac to limit the saturation. The detail aspects of the freezing mechanism at different levels are still under exploration [37].

In the inference or retrieval mode, a simple winner choice method is used in which, each CM, the neuron with the highest local likelihood value, $V(i)$ is chosen as the winner. In this mode, the network is asked to find the closest match and not asked to learn anything new. The optimal pure retrieval mode will be the maximum likelihood estimation in conjunction

Algorithm 2.1: The Code Section Algorithm (CSA) during Learning

for *episode* $\leftarrow 0$ **to** T **do**
  if M $=$ *active* **then**
    for $j \leftarrow 0$ **to** Q **do**
      for $i \leftarrow 0$ **to** K **do**
        $u(i) \leftarrow (\sum x(t-0).w, \forall(x,w) \in (X_u, W_u))$
        $h(i) \leftarrow (\sum x(t-1).w, \forall(x,w) \in (X_h, W_h))$
        $d(i) \leftarrow (\sum x(t-1).w, \forall(x,w) \in (X_d, W_d))$
        $U(i), H(i), D(i) \leftarrow$ Normalize$(u(i), h(i), d(i))$
        if $t \neq 0$ **then**
          Local Support$(V(i)) \leftarrow U(i).H(i).D(i)$
        else
          Local Support$(V(i)) \leftarrow U(i)$
        end if
      end for
    end for
    for $j \leftarrow 0$ **to** Q **do**
      $V(j) \leftarrow$ Max$(V(i)), \forall i \in j$
    end for
    Parameters of Sigmoid$(G, \eta) \leftarrow (V(j), Q, K)$
    for $j \leftarrow 0$ **to** Q **do**
      for $i \leftarrow 0$ **to** K **do**
        $\psi(i) \leftarrow$ Sigmoid$(V(i), G, \eta)$
      end for
    end for
    for $j \leftarrow 0$ **to** Q **do**
      for $i \leftarrow 0$ **to** K **do**
        $\rho(i) \leftarrow$ Softmax$(\psi(i))$
      end for
    end for
    for $j \leftarrow 0$ **to** Q **do**
      Select a Winner $\leftarrow$ Winner-take-all$(\rho(i))$
      Update Weights $\leftarrow$ Hebbian Learning
    end for
  else
    return
  end if
end for

Algorithm 2.2: The Code Section Algorithm (CSA) during Inference

```
for episode ← 0 to T do
    if M = active then
        for j ← 0 to Q do
            for i ← 0 to K do
                u(i) ← (∑ x(t − 0).w, ∀(x, w) ∈ (X_u, W_u))
                h(i) ← (∑ x(t − 1).w, ∀(x, w) ∈ (X_h, W_h))
                d(i) ← (∑ x(t − 1).w, ∀(x, w) ∈ (X_d, W_d))
                U(i), H(i), D(i) ← Normalize(u(i), h(i), d(i))
                if t ≠ 0 then
                    Local Support(V(i)) ← U(i).H(i).D(i)
                else
                    Local Support(V(i)) ← U(i)
                end if
            end for
        end for
        Maximum Likelihood Estimation ← V(i)
    else
        return
    end if
end for
```

with the back-off policy currently under exploration [37]. The deterministic algorithm called the "maximum likelihood" method is much faster since it obviates the later steps of the CSA and for many applications, it yields much better performance and higher expected accuracy [37]. Algorithm 2.1 and Algorithm 2.2 explains the learning and inference steps of the CSA, respectively.

2.2.2 Long Short-Term Memory

Recurrent neural networks (RNNs) are very powerful for sequence modeling tasks that can capture temporal information of inputs by sharing weights in time. As temporal dependency increases, these RNNs are very difficult to train for vanishing or exploding gradient problems and rarely used in deep learning applications [39]. Long Short-Term Memory (LSTM) is a popular variant of RNNs because of its immunity to the gradient vanishing or exploding problems [12]. The basic concept of LSTM network, important properties, and algorithmic steps that involve two different phases (training and inference) are discussed in this section.

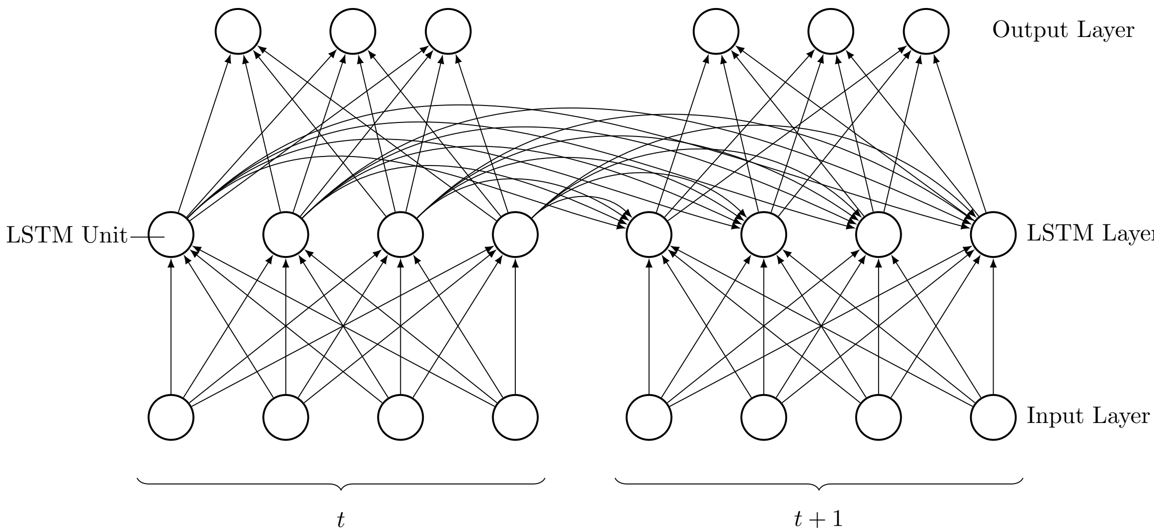

Figure 2.7: A single layer LSTM network

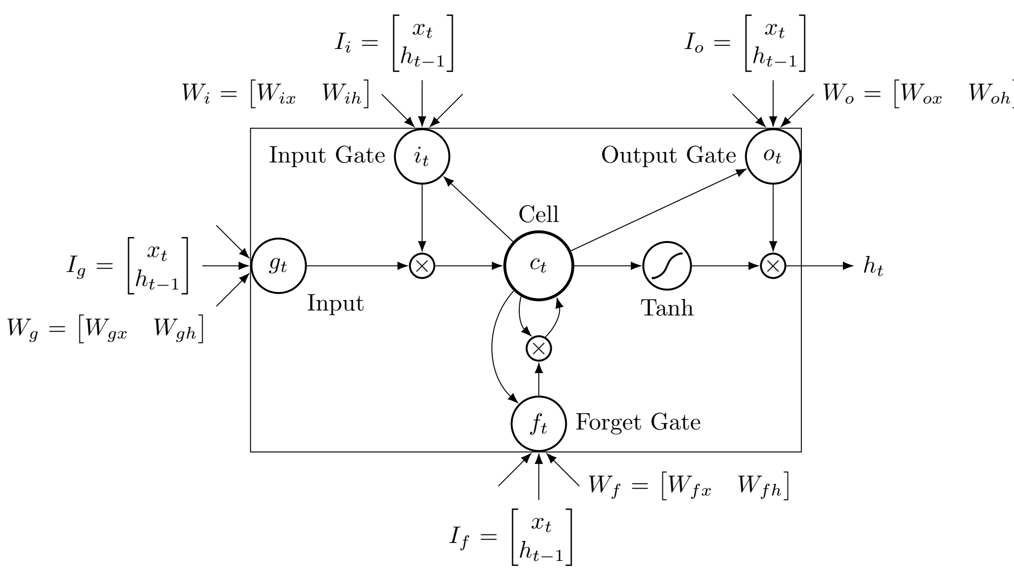

Figure 2.8: An LSTM Unit

Basic Concepts of LSTM Network

Memory cell or LSTM unit (shortly unit) is the basic computational unit arranged in layers. A single unit is guarded by three different multiplicative gates known as input gate, output gate, and forget gate. The multiplicative input gate usually protects the unit (or memory content) from irrelevant inputs, and the multiplicative output gate protects other units from the currently perturbed unit. The multiplicative forget gate protects the unit from the irrelevant past content (memory state) by resetting it at an appropriate time [40]. A single unit and associated gates usually have connections (or weight parameters) from an input and also from itself. In addition to units arranged in stacked LSTM layers, there are also output units arranged in the one or multiple output layers. An output unit usually has connections from all outputs (or activations) of the subjacent layer [i.e. output layer is usually fully-connected (FC)]. Similar to the perceptron, a non-linear activation function (Sigmoid, Tanh) used in a unit as well as in gates. Despite several variants of LSTM, a general LSTM network (an input layer, an LSTM layer, and an output layer) is shown in Figure 2.7, and a closer view of an LSTM unit is shown in Figure 2.8 [41]. An LSTM unit with three multiplicate gates (Figure 2.8) has similar inputs I ($I_g = I_i = I_f = I_o$) at particular time step (t) which is the concatenation of the current feedforward inputs (x_t) and the activations from previous hidden states (h_{t-1}). On the other hand, the LSTM unit and its multiplicate gates maintain separate weight matrices (W_g, W_i, W_f, and W_o) where each weight matrix is the concatenation of the feedforward weights and the recurrent weights. For instance, the weights of an input gate (W_i) is the concatenation of inputs to input gate weights (W_{ix}) and hidden states to input gate weights (W_{ih}). Also, an LSTM unit and its multiplicative gates can have their bias parameters. The peephole connections of an LSTM unit are the connections from the cell (c_t) to the gates (i_t, f_t, and g_t) and are optional in many implementations.

Properties of LSTM Network

The unique features of LSTM than other RNNs are, first, that it has the mechanism to truncate gradient during training and can learn to bridge more than 1000 discrete-time steps [12]. This feature helps LSTM to solve problems that require learning very long-range temporal dependencies. Second, it also has the forget/remember mechanism to wipe out/retain previous memory based on current context [40].

Algorithm 2.3: The Forward Porpagation of an LSTM Layer

for $t \leftarrow 0$ **to** T **do**
  for $g \leftarrow 0$ **to** $Unit$ **do**
    $i_t \leftarrow \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$
    $f_t \leftarrow \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$
    $o_t \leftarrow \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$
    $g_t \leftarrow \tanh(W_{gx}x_t + W_{gh}h_{t-1} + b_g)$
    $c_t \leftarrow f_t \odot c_{t-1} + i_t \odot g_t$
    $h_t \leftarrow o_t \odot \tanh(c_t)$
  end for
end for

Algorithm 2.4: The Backward Porpagation of an LSTM Layer

$W_x \leftarrow [W_{gx}, W_{ix}, W_{fx}, W_{ox}]^T$
$W_h \leftarrow [W_{gh}, W_{ih}, W_{fh}, W_{oh}]^T$
$gates_t \leftarrow [g_t, i_t, f_t, o_t]^T$
$b \leftarrow [b_g, b_i, b_f, b_o]^T$
for $t \leftarrow T$ **to** 0 **do**
  for $g \leftarrow 0$ **to** $Unit$ **do**
    $\delta h_t \leftarrow \Delta_t + \Delta h_t$
    $\delta c_t \leftarrow \delta h_t \odot o_t \odot (1 - \tanh^2(c_t)) + \delta c_t \odot f_{t+1}$
    $\delta g_t \leftarrow \delta c_t \odot i_t \odot (1 - g_t^2)$
    $\delta i_t \leftarrow \delta c_t \odot g_t \odot i_t \odot (1 - i_t)$
    $\delta f_t \leftarrow \delta c_t \odot c_{t-1} \odot f_t \odot (1 - f_t)$
    $\delta o_t \leftarrow \delta out_t \odot \tanh(c_t) \odot o_t \odot (1 - o_t)$
    $\delta x_t \leftarrow W_x^T \cdot \delta gates_t$
    $\Delta_{t-1} \leftarrow W_h^T \cdot \delta gates_t$
  end for
end for

Algorithm 2.5: The Weight Update using SGD

$W_x \leftarrow [W_{gx}, W_{ix}, W_{fx}, W_{ox}]^T$
$W_h \leftarrow [W_{gh}, W_{ih}, W_{fh}, W_{oh}]^T$
$b \leftarrow [b_g, b_i, b_f, b_o]^T$
for $t \leftarrow T$ **to** 0 **do**
    $\delta W_x \leftarrow \delta W_x + \delta gates_t \cdot x_t$
    $\delta W_h \leftarrow \delta W_h + \delta gates_{t+1} \cdot h_t$
    $\delta b \leftarrow \delta b + \delta gates_{t+1}$
end for
${W_x}^{new} = {W_x}^{old} - \eta \cdot \delta W_x$
${W_h}^{new} = {W_h}^{old} - \eta \cdot \delta W_h$
$b^{new} = b^{old} - \eta \cdot \delta b$

Algorithms of LSTM

To perform training of LSTM network, the first-order optimization technique, Gradient Descent (GD), is mostly used where it calculates the gradient of the loss function (a summation of per-timestep losses) of LSTM network for parameters. The key step of GD is to calculate the gradients or derivatives which are computed using the backpropagation through time (BPTT) algorithm. Due to calculate gradients using BPTT, it is required to calculate each layer's activations (hidden states and output states) through a forward propagation (a.k.a, forward pass) from the input layer to the output layer. The inference results (from forward propagation) are usually passed through the loss function to calculate the loss or inference error. Next, the BPTT is used to calculate the gradients iteratively from the output layer to the input layer. This is known as backpropagation or backward propagation (a.k.a, backward pass). Based on the gradient of loss with respect to parameters and the step size [i.e. learning rate (η)], the GD is applied to update the weights. The GD optimization is memory inefficient for huge training datasets where the gradients need to be calculated for all examples before applying GD optimization. Instead of computing the true gradients, the gradients on a single random sample or a small batch of random samples are often calculated to do the optimization. The optimization based on a random example and a small batch of random samples is known as Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent (mini-batch GD), respectively. The mini-batch GD is an efficient and dominant method to train the LSTM networks.

A single training iteration is comprised of a forward propagation, a backward propagation, and a weight update using mini-batch GD and Adam. On the other hand, an inference iteration is simply a forward propagation. It usually requires millions of training iterations to perform training of an LSTM network. The pseudocodes for forward propagation, backward propagation, and weight update of an LSTM layer are presented in Algorithm 2.3, Algorithm 2.4, and Algorithm 2.5, respectively. The $\odot$ and σ in Algorithm 2.3 represent the element-wise product (or Hadamard product) and the sigmoid function. The Δ_t and Δh_t in Algorithm 2.4 are the output difference as computed by the subsequent layers and the next time step of same layer, respectively.

2.3 Development of Neural Networks

Use higher-level deep learning frameworks is one of the fastest ways to implement these networks in hardware. These frameworks accelerate computations (training and inference) on designated hardware by characterizing the workloads and mapping workloads into the hardware accordingly. A list of popular frameworks are Caffe, Torch, PyTorch, Tensorflow, Theano, and Keras which are discussed in [42, 43, 44]. Most of the frameworks support multiple CPUs, GPUs and flexibility to construct dataflow graph for tensor computations. In the case of GPU, it can also use Nvidia's cuDNN libraries for fast execution on Nvidia GPUs [45]. In addition to CPUs and GPUs, Tensorflow also supports Google's custom hardware (TPU) to accelerate the inference of DNNs [27]. A high-level deep learning framework also uses Caffe framework as a programming interface to accelerate different DNNs on FPGA [46, 47]. The recent implementations of Sparsey and LSTM on various hardware accelerators are discussed in the next chapter.

Chapter 3

State-of-the-art

3.1 Introduction

In the previous chapter, the fundamentals of neural networks followed by architectures and algorithms of Sparsey and LSTM were presented. Also, the development environments to accelerate these networks on hardware platforms are discussed. In this chapter, the state-of-the-art of these hardware platforms, especially architecture and performance, are presented.

3.2 Hardware Plaforms

There are different kinds of hardware platforms for deep learning applications. This section will present the different types of state-of-the-art hardware for Sparsey and LSTM implementations.

3.2.1 General Purpose Processor

The implementation of multilayer neural networks is mostly done in CPU and GPU-optimized CUDA-based (cuDNN accelerated) platforms using various deep learning frameworks. The cuDNN library is optimized specially for CNN and RNN. It accelerates these deep learning frameworks through efficient memory management and parallel computations. A three-layer Sparsey model (total 84×10^3 neurons) was implemented in CUDA-accelerated server class Nvidia Tesla K20c GPU [48]. Single learning and inference of this implementation were 86.63ms and 4.35ms, respectively, on KTH action recognition datasets [21].

The multi-stage code-optimized LSTM networks were implemented in Nvidia M40 GPU for different units (hidden states), layers, and mini-batch sizes in [18]. This implementation achieved approximately $6\times$ speedup over the standard optimization for four layers with 512 hidden states and mini-batch of 64. The reported speedup achieved on this implementation was only for forward propagation (inference) measured on 100 iterations [18]. The recent implementation of LSTM network using multiple GPUs shows faster training for Seq2Seq (machine translation) processing task [14], [49]. Data parallelism was used in this implementation to split a mini-batch among multiple GPUs, and the results were merged accordingly. Three popular deep learning frameworks (TensorFlow, MXNet, and CNTK) were used to develop an LSTM network in multiple Quadro P400 GUPs and achieved the highest training throughput of 280 samples/s for mini-batch size 64. The key observations of these state-of-the-art multi-GPUs LSTM benchmarking are: 1) the throughput of training (samples/s) increases with minibatch sizes; 2) the utilization of compute units increases with the mini-batch sizes; and 3) the use of compute units is relatively low (approximately 40% utilization for mini-batch of 64) for LSTM i.e. LSTM is memory bound, not compute.

In addition to GPU, there are few other hardware approaches to accelerate the performance of LSTM implementation. However, most of the accelerators are geared to improve the performance of inference and depend on GPU for training and preprocessing.

3.2.2 Field-Programmable Gate Array

Field-Programmable Gate Arrays (FPGAs) are often used to take advantage of the programmability and rapid-prototyping capabilities of a neural network. FPGAs are mostly used to implement the preprocessed LSTM networks for inference mode only [50]. A relatively small size LSTM network without classification layer was implemented in [51] and benchmarked against Cortex-A9 CPU. In [52], different levels of parallelism and approximated activation functions were used to implement LSTM network in FPGA and performed training and inference. The performance (area, speed) was much better than that of CPUs. However, the accuracy of trained network decreased and the performance (speed) of FPGA implementation for the large-scale network was worse than for GPU. The FPGA and ASIC solution were presented in [53] for Gated Recurrent Unit (GRU) and benchmarked against a GPU and a CPU. The inference network of GRU was studied and stored precomputed values to lower the computational complexity. The ASIC and FPGA performed better than CPU and GPU for the non-batch operation that involve relatively small networks. Preprocessed and quantized parameters of an LSTM network stored in the on-chip memory of FPGA and low power speech processing system was proposed in [54]. The compressed and pruned version of LSTM network was also implemented in [55]. The inference speed was 3× faster and 11.5× energy efficient than a GPU.

3.2.3 Custom Hardware

A novel compression method of LSTM is proposed in [56] to reduce memory use up to 95%. The circulant matrix-vector multiplication, different quantize methods, and an approximation of the non-linear activation function were used to implement the compressed LSTM on hardware. This work achieved significant performance improvement with minimal accuracy loss. However, the retrain process was used offline to improve the accuracy of this compressed LSTM network. The state-of-the-art ASIC acceleration of three popular neural networks (MLP, CNN, and LSTM) was in Google's Tensor Processing Unit (TPU) [27]. The TPU was implemented to accelerate Google's datacenter computation demands. The TPU was optimized to reuse weights across a large batch of independent input during inference only. The 65,536 MAC units (8-bit fixed-point), 28 MiB on-chip memory, and systolic execution-based TPU's architecture was 15× faster and 30× higher TOPS/Watt during inference than an Nvidia K80 GPU. In addition, the TPU work revealed several important findings of the computations and resources required for three popular DNNs. CNNs are computation bound with only 5% of the total workload, and MLPs and LSTMs are memory bound. EIE [57] proposed an energy efficient solution to accelerate sparse matrix-vector multiplication and was specialized for data sharing technique for compressed CNN and LSTM. This solution was designed to target only inference of deep CNN (used weight sharing and dynamic Sparsity of activation). In [58], DNPU presented a reconfigurable, energy-efficient processor with on-chip memory for combined CNN-RNN inference. The fixed-point adaptation and table-based multiplication helped DNPU processor to achieve 6.5× energy efficiency over EIE.

3.2.4 Domain-specific Processor

A domain-specific Instruction Set Architecture (ISA) was presented in Cambricon [59] that targeted ten different neural networks. The load-store based architecture, Cambricon, has 64 general purpose registers (32-bit each) and scratchpad memory instead of vector register files. Cambricon defined 43 different instructions after a careful evolution of 10 different neural networks and reduced code density over GPU. The accelerator benchmarked small LSTM network (training and inference) against GPU and on average it achieved 3.09× speedup. The integration and exploration of high bandwidth memory (3D stacking and non-volatile memory) of Cambricon were left for future work.

Recent exploration of DNN algorithms has shown promising results (classification accuracy) of CNN and RNN limited to binary (-1, 0, 1 or -1, 1) weights during training process [60, 61, 62, 63]. In this method, the full-precision weights are stored and updated after binarized forward and backward propagation. Some other compressed networks using this approach have shown good accuracy for the specific benchmark but were not mature enough globally [64]. The training processes of this technique are still performed on traditional hardware platforms (CPUs or GPUs).

3.3 Conclusion

The FPGA and ASIC implementations were based on quantized, preprocessed, and/or compressed deep neural networks for inference. The compressed, fixed-point and/or binary networks were also generated for inference hardware during extensive training in GPUs. The Cambricon is the hardware capable of both training and inference. This work presents an instruction based 3D architecture with stacks of emerging memories for both training and inference. It is different than [59] in the sense that it enhances memory bandwidth using an emerging 3D memory system, issues up to five instructions (VLIW), performs special function operations using lookup table based hardware, uses mixed precision computation (required for mixed precision training) and can process a large DNN on natural datasets (needed high storage).

Chapter 4

A Custom Hardware for Sparsey

4.1 Introduction

A cortical algorithm, Sparsey that can learn with completely unlabeled data are gaining interest. The primary goal of Sparsey is to model a hierarchical sparse distributed memory similar to the episodic memory of a human. Sparsey models this episodic memory by storing/retrieving contextual information to/from memory of a specific event during learn/inference. The algorithm is highly memory dependent and does not perform well in conventional hardware (GPUs and CPUs) which precipitated efforts to design custom hardware catering to the distributed memory needs of this algorithm. In this chapter, a scalable and configurable hardware accelerator for Sparsey is proposed. It accelerates the memory read and write operations of synaptic weight matrices using on-chip memory in every neuron.

In the following sections, the design space exploration of an accelerator including the computational complexity, the arithmetic precision, the memory bandwidth of on-chip memory, and the interconnection network at different levels are discussed. The next section discusses the hardware implementation details of a cluster for this accelerator. The experimental setup and the performance are discussed in the subsequent section. The discussion section explores the scalability of this proposed accelerator. This chapter concludes by discussing the lesson learn from this work.

4.2 Design Space Exploration

This section discusses several important design considerations and requirements of the proposed custom accelerator based on software implementation in traditional hardware.

4.2.1 Computation

There are two types of arithmetic operations required for learning and inference of Sparsey network. The binary and integer operations (binary multiplication followed by integer accumulations) is the first steps of learning and inference. These operations create a bottleneck of a large number of neurons required to perform this operation parallelly on synaptic weights. For instance, the total number of binary and integer operations of a layer in a given Sparsey network are approximately equal to the total number of synapses (binary weights) in that layer. In a small Sparsey network (three layers network implemented in GPU) for 160×120 image processing task can have at least 450 million synapses. To perform real-time processing (at 60 frames/s), this network requires at least 54 GOPS related to binary multiplications and integer accumulation. The operations/s increases exponentially with the real-life image sizes (1600×1200), the number of neurons in a layer and also the number of layers. The upper bound of multiply-accumulation operations of a layer in a given Sparsey is $O(KQMW)$, where K, Q, M, and W are the number of neurons in a minicolumn, the number of minicolumns in a macrocolumn, the number of macrocolumns in a layer, and the total number of weights of a neuron in that layer. To achieve the highest degree of acceleration, the MAC units should be designed in a way so that the operations/s of MAC hardware is approximately equal to the bandwidth of distributed on-chip SRAMs. In other words, the custom hardware is designed to utilize the fully available on-chip memory bandwidth to maximize the throughput during binary multiply-accumulation operations.

Next consider the floating-point operations which are required for subsequent steps of the CSA. The total number of floating-point operations of a given Sparsey network are insignificant compared to the number of MAC operations. Sparsey network is also required to perform non-linear activation function (Sigmoid) for every neuron, find a maximum value (MaxV) in a vector, softmax (SoftMax) operation, and winner-take-all (WTA) operation on floating-point values throughout its algorithmic steps. The upper bound of Sigmoid, MaxV, probabilities of SoftMax, and WTA are $O(KQM)$, $O(Q)$, $O(KQM)$, and $O(Q)$, respectively.

4.2.2 Memory

The most important design consideration for this accelerator is the amount of memory and memory bandwidth available to access the synaptic weight matrices. The first step of learning and inference (Algorithm 2.1 and Algorithm 2.2) involves accessing the weight matrices. The last step of learning requires simultaneously updating the weights of each neuron. A neuron's bottom-up (N_u), horizontal (N_h), and top-down (N_d) synaptic storage (in bits) can be calculated from equations (4.1) – (4.3).

$$N_u = \begin{cases} N_{bu} & \text{if } l = 1 \\ N_{l-1} \times M_{l-1} & \text{if } l > 1 \end{cases} \tag{4.1}$$

$$N_h = N_l \times (1 + M_l) \tag{4.2}$$

$$N_d = \begin{cases} N_{l+1} \times M_{l+1} & \text{if } l < l_{max} \\ 0 & \text{if } l \geq l_{max} \end{cases} \tag{4.3}$$

$$N_s = N_u + N_h + N_d \tag{4.4}$$

The N_{bu}, N_l, M_{l-1}, M_l, and M_{l+1} are the number of bottom-up input bits, the total number of neurons in a Mac of layer l, the number Macs in layer $l-1$ whose sparse distributed code (SDC) are feeding to a single neuron of next layer's Mac, the number of neighbor Macs (to the north, south, east, and west) of that neuron in layer l and the number of Macs in layer $l+1$ feeding to the subjacent layer's neurons. The total memory requirement for synaptic weight matrices of a neuron (N_s) in layer l can be calculated from equation (4.4). The software implementation of Sparsey on Tesla K20c GPU made it clear that the most significant bottleneck of Sparsey on GPU is moving weight matrices from memory to processor and back to the memory for each frame or episode. One of the solutions to this bottleneck is making the local memory to the processor large enough to hold the synaptic weights of the network inside the processor. A custom hardware can be optimized with large distributed on-chip SRAMs with multiple ports to support enough storage and memory bandwidth required for Sparsey network.

The inputs (frames of an episode) of Sparsey are binary sparse matrices where each element represents an input feature vector. The weight matrices (bottom-up, horizontal, and top-down) of neurons in Sparsey network are also binary instead of floating-point. The memory systems of traditional hardware are byte addressable instead being bit addressable and thus results in the need to allocate excessive storage and memory bandwidth for storing and retrieving the binary weight matrices. However, the recent advance of GPU software supports a special function called the population count (*__device__ int __popcll*) where 64-bit data are packed and transferred from memory to processor for counting the number of bits that are set to 1. Custom hardware can also be designed in a way that it can process data at bit-level in hardware to relax the amount

Table 4.1: The range and format of different parameters for this accelerator

Parameter Name	Minimum Value	Maximum Value	Storage Format
Random Number	0	1	Q0.16
exponential	0	8	Q3.13
λ	0	32	Q5.11
ϕ	0	8	Q3.13
G_T	0	1	Q0.16
π	0	63	Q06.0

of storage and bandwidth requirement of the custom hardware.

4.2.3 Precision

The CSA of Sparsey is online and probabilistic (the neuron to which the code is assigned is based on the random number). The algorithm (code assignment using CSA) is highly dependent on the feature vectors of the input layer and the random numbers rather than the arithmetic precision of the intermediate steps. Therefore, 16-bit fixed-point hardware can be used to replace all intermediate floating-point operations. The 16-bit fixed-point adder and multiplier are respectively 63× and 4× less area, and 18× and 2.4× more energy efficient compared to the 32-bit floating-point version [26]. Replacing 32-bit floating-point arithmetic with 16-bit fixed-point substantially improves the overall area and power consumption of this accelerator. The entire computational steps of learning and inference were carefully analyzed for the suitability of using a 16-bit fixed-point format. For instance, the 16-bit fixed-point multiplication of two normalized fractional operands (algorithm step 10 of Algorithm 2.1 and Algorithm 2.2) is always a fractional number and still less than one. This makes use of the entire 16-bit range for the fractional representation (Q0.16). Similarly, the controlling parameters (λ, ϕ, G_T, and π) of the Sigmoid function (algorithm step 22 of Algorithm 2.1) are carefully analyzed as per [31] and the range of values are determined accordingly. The fixed-point format of these controlling parameters are Q5.11, Q3.13, Q0.16, and Q6.0, respectively. The final probabilities of winning (step 22 of Algorithm 2.1) of neurons are also always less than one and the entire 16-bit range (Q0.16) used to store the winning probability. The pre- and post-adjustment of operands and results can be done using shift operation in this implementation, which may result in loss of high precision. Moreover, the combination of fixed-point arithmetic and Taylor series are used to implement the Sigmoid and the exponential function in this implementation. The storage

Table 4.2: The configurable units of a PE

Computational Unit	Minimum Value	Maximum Value
Neurons/CM	0	64
CMs/Mac	0	64

formats of fixed-point representation in this design are listed in Table 4.1.

4.2.4 Configurability

The basic functional unit of Sparsey network is the macrocolumn (Mac) which is designed in hardware and referred as the processing element (PE). A cluster of the hierarchical accelerators can have an arbitrary number of PEs. The number of computational units (neurons per CM and CMs per Mac) can vary based on the application. The recent implementation of Sparsey network on the MNIST dataset has 9 neurons per CM and 11 CMs per Mac [25]. On the other hand, the maximum 14 neurons per CM and 7 CMs per Mac were used for the episodic recognition of 64×64-frame from natural snippets [65]. Thus in custom hardware design, a PE should be configurable, that is, it may require to process a smaller number of computational units (neurons per CM and CMs per Mac) than that of a real cortical macrocolumn (~20 neurons per CM and ~70 CMs per Mac). A configurable PE is required to design the hierarchical accelerator and it should be at least in compliance with the structural properties of the cortical macrocolumn [31]. The hardware specification of a configurable PE in this implementation is listed in Table 4.2. A single Mac can be configured up to 64 CMs and a single CM configured up to 64 neurons.

$$S = \frac{N_{bu} + N_d + M_{neighbor} \times N_{npe}}{8} \tag{4.5}$$

4.2.5 Communication

There are two types of communication required among PEs in the design of hierarchical accelerator (comprised of multiple clusters). In the Sparsey network, a Mac directly communicates with only its nearest neighbor in a layer, i.e. to the north, south, east, and west. Thus PEs in a cluster can be interconnected as a rectangular mesh structure where each PE can also communicate with its neighbors. The benefit of using a mesh topology is that the Sparsey network has a high degree of physical locality during inter-PE communication. In CSA, every neuron of a PE receives the same activations from every neuron of the nearest PEs after every frame interval. The message size (S) in byte of a PE in any layer for mesh interconnect network can be calculated (equation 4.5) using the number of neurons in that PE (N_{npe}), the number of neighbor PEs ($M_{neighbor}$), and the number of bottom-up (N_{bu}) and top-down (N_d) inputs of that PE. The traffic pattern of mesh NoC depends on the type of inputs. The horizontal inputs of Sparsey are confined within a layer while bottom-up and top-down inputs are confined between consecutive layers. The traffic pattern of horizontal inputs is uniform but bottom-up and top-down traffic are permutation traffic (traffic from a PE or group of PEs is directed to a neuron) [66].

The second type of communication among PEs occurs when an arbitrary number of clusters are connected using a global interconnect network. Due to design a scalable and hierarchical accelerator, a scalable interconnect network is required to form a global interconnect network. The *Fat tree* topology is a scalable interconnect network and also it has path diversity and fault tolerance properties. In addition, it is relatively easy to implement and is cost-efficient [67, 68].

4.3 Hardware Implementation Details

This section describes the hardware architecture of hierarchical accelerator and the on-chip interconnection network for the proposed accelerator. The functional units of a cluster and two different interconnection topologies for local (intra-PE) and global (inter-cluster) communication are also discussed in this section.

4.3.1 Cluster

A cluster in a hierarchical accelerator is a 2D array of PEs. Figure 4.1 shows the hardware architecture of a single cluster. The cluster has 4×4 PEs, a local mesh interconnection network, and a cluster controller. During the configuration phase of a cluster, the input configurations (neurons per CM, CMs per Mac, the number of input feature vectors (FVs), etc.) are passed to each of the PEs. Based on the configuration, a PE can configure itself for a range of different computations as well as controlling parameters of activation functions. The local mesh NoC maintains the inter-PE communications within this cluster and also communicates with the outside using global interconnection network. The cluster controller generates the required control signals of all PEs and performs synchronization among all PEs. The basic operation that takes place in every PE of a cluster is to compute the SDC output after training and inference of each frame or episode.

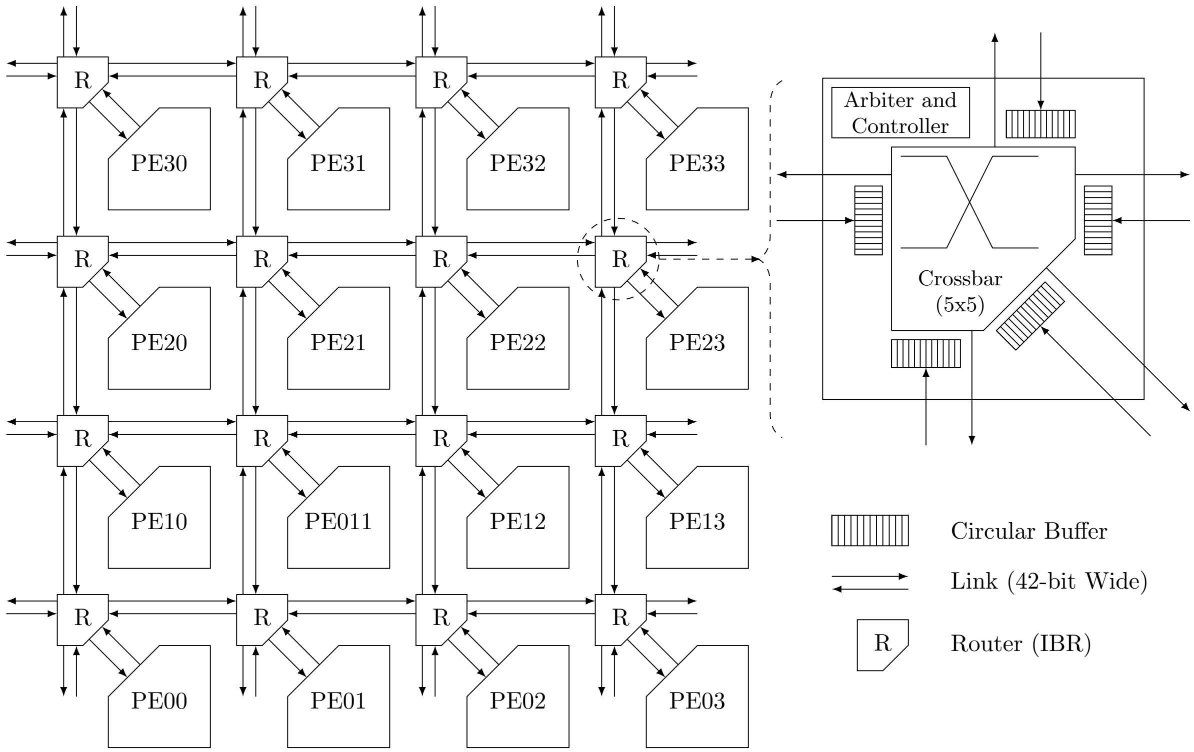

Figure 4.1: The hardware architecture of a cluster

4.3.2 Processing Element

Figure 4.2 shows the hardware architecture of a PE. A PE consists of a group of 16 CMs, a weight summation submodule, and a fixed-point arithmetic logic unit (ALU). The input of the weight summation submodule (Weight Sum submodule of Figure 4.2) is the sparse binary matrix, where it calculates the number of active features (the number of bits that are set to 1) of its input. A modified version of multiply-accumulate submodule (Figure 4.5) was used during designing the weight summation submodule. The custom design fixed-point ALU supports addition/subtraction, multiplication, and division operation on 16-bit fixed-point operands. The fixed-point ALU in a PE usually used to calculate the global familiarity (G) and the expansively (η) during learning mode. An on-chip register file size of 1 Kb was used to store the intermediate output of all CMs. For instance, a physical CM is configured up to 4 logical CMs during the configuration phase of a cluster and the temporary values (especially Vmax and index of winner neurons) of 4 CMs are saved in on-chip register files for the next computational steps. In addition to the on-chip register file, a small on-chip SRAM size of 64 Kb is always available in a PE for storing the input FVs and the output SDC from all CMs. Since the CMs of a PE operate identically on different datasets, a single PE controller is used to control all

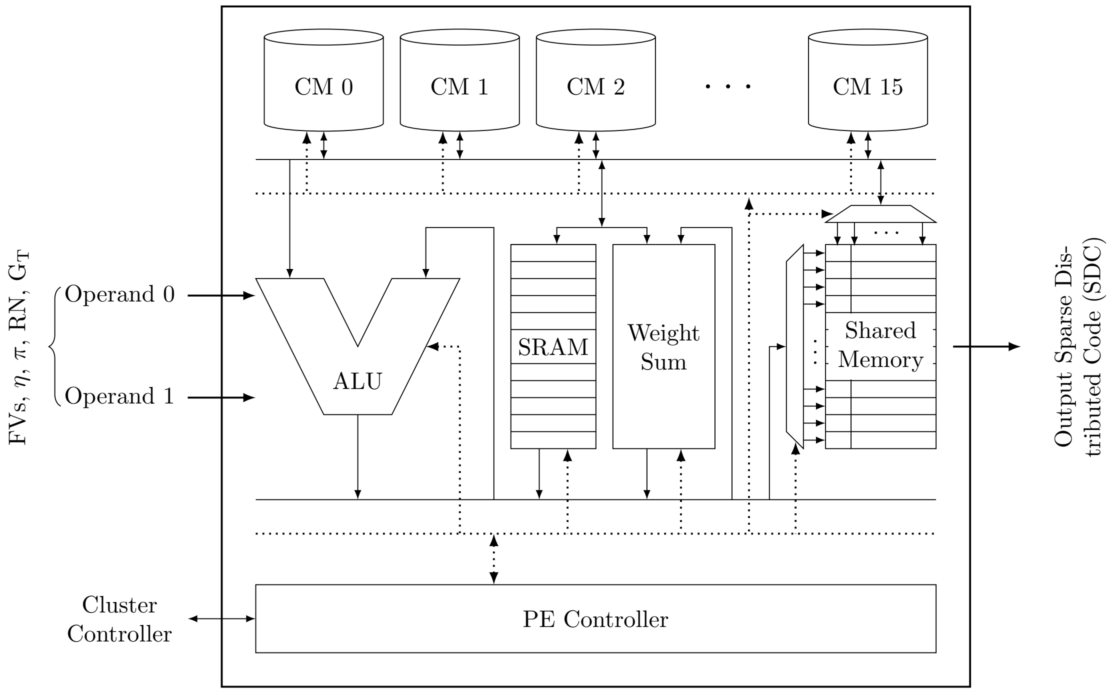

Figure 4.2: The hardware architecture of a PE

CMs of that PE along with the synchronization. The controller of PE also generates a control signal ($Active(Mac)$) based on the number of active features (the output of weight summation module) and the input threshold (π_u). This control signal is used to turn on/off the PE based on the input threshold. The key submodules of a PE are discussed in the subsequent section.

Competitive Module (CM)

The hardware architecture of a CM is shown in Figure 4.3. A CM consists of 16 physical neurons, and three other submodules: MaxV; Binary Search; and Prefix Sum. The MaxV submodule determines the highest value of local support or likelihood from local support values of all neurons in a CM. It uses the divide-and-conquer approach for finding the maximum value in an array (the local likelihood). The Binary Search submodule determines the position or index of a winner neuron based on a random number (RN) and the winning probabilities of all neurons in that CM. The RNs were supplied from the hardware outside of this design. The binary search is the second part of the winner-take-all (WTA) process where the first part calculates the cumulative probabilities of all neurons in a CM. The Prefix Sum submodule is used to calculate the cumulative probabilities and the summation of relative probabilities for all neurons. The summation of relative probabilities is usually passed to neurons for calculating the winning probability of each neuron. The Prefix Sum and Bin Search submodules are

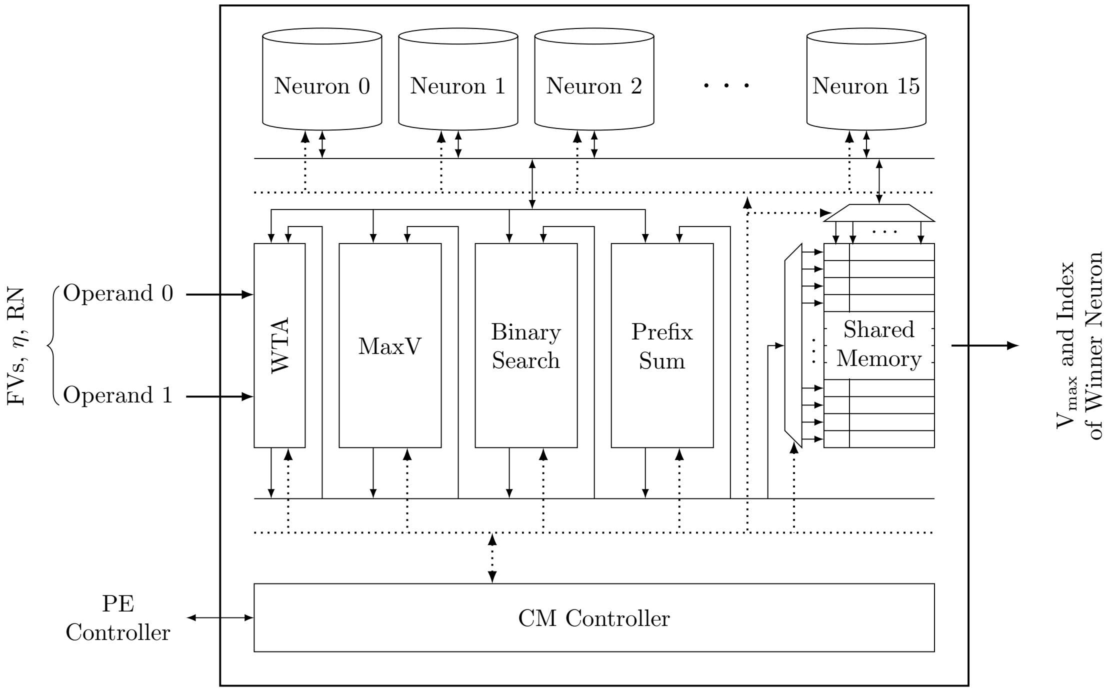

Figure 4.3: The hardware architecture of a CM

designed based on the parallel prefix-sum and the binary search algorithm where they are used to perform the complete WTA processes. These submodules are also designed for 16-bit fixed-point operands. In addition to these submodules, a CM has 4 Kb shared memory that stores the intermediate/temporary data for neurons. For instance, a physical hardware of a neuron is configured up to 4×4 ways based on the initial configuration of this accelerator (Table 4.2). The multi-ported register files are used to implement this shared memory. The inputs of a CM are feature vectors (FVs), expansivity (η), and a random number (RN) and the outputs are maximum local support (V_{max}), an index of a winner neuron. The controller in CM generates all control signals required for all submodules and neurons in that CM. It also receives control from the top-level controller (the PE controller) and performs synchronization of all neurons. The synchronization mechanism of all neurons is similar to the barrier synchronization of GPU [69].

Neuron

Figure 4.4 shows the hardware architecture of a neuron. The neuron is the basic computational unit in this hardware architecture. A single neuron has a dedicated on-chip SRAM (single port SRAM with word size 32-bit) for synaptic storage, a neuron controller, a custom multiply-accumulate submodule, a special function unit (SFU), and a 16-bit fixed-point arithmetic logic

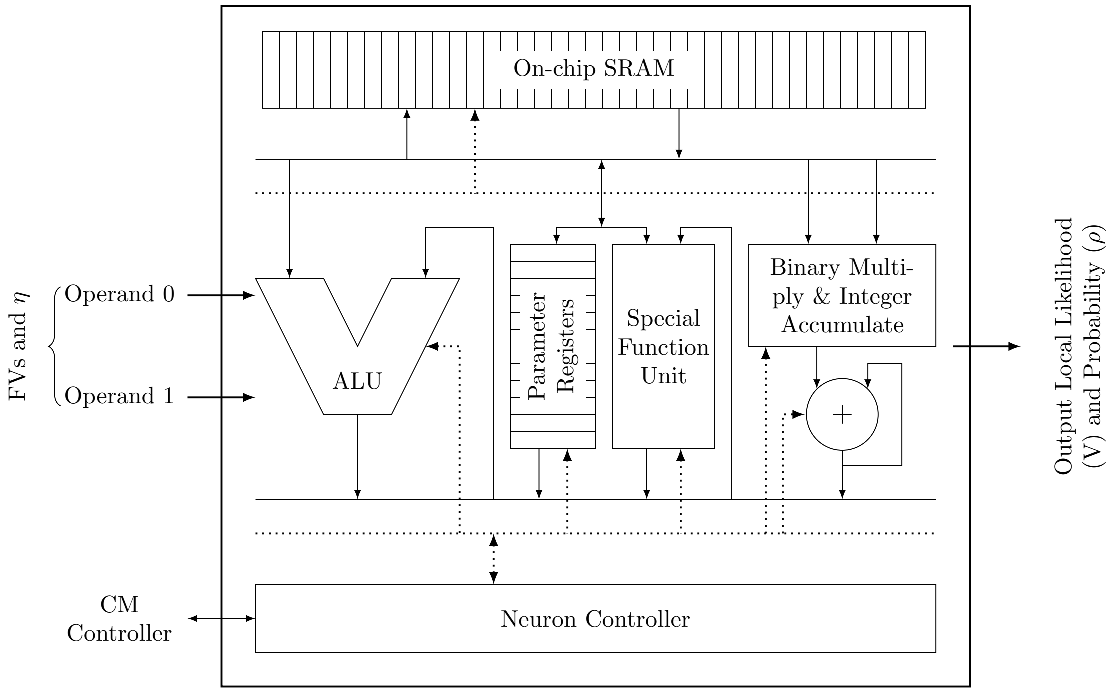

Figure 4.4: The hardware architecture of a neuron

unit (ALU). The fixed-point ALU is custom designed for addition, multiplication, and division operations. The SFU submodule calculates the parameterized (configure during the configuration phase) non-linear activation function (Sigmoid) based on inputs and controlling parameters. The neuron controller generates control signals of all submodules based on the CM control signal.

Multiply-accumulate

The submodule multiply-accumulate performs the multiply-accumulate operation on binary input feature vectors (FVs) and binary synaptic weights for U, H, and D-inputs from on-chip SRAM. Implementation of a three-layer Sparsey network of 84×10^3 neurons on Tesla K20c GPU shows (through CUDA profiler) that the binary multiply-accumulate operation takes more than 80% processing time because of fetching weight parameters from the memory [48]. This is the major bottleneck during learning and inference of Sparsey network in software implementation. A custom fully pipelined version of this submodule was designed in a way so that the operations per second (OPS/s) of this submodule is equal to the bandwidth of on-chip SRAM in 32 Gb/s. Figure 4.5 shows the design of this MAC submodule where it takes 32 weight parameters (32-bit wide) from an on-chip SRAM (single port and 32-bit wide) and performs MAC operation with 32 input FVs. It basically performs the binary dot product of the input operands and

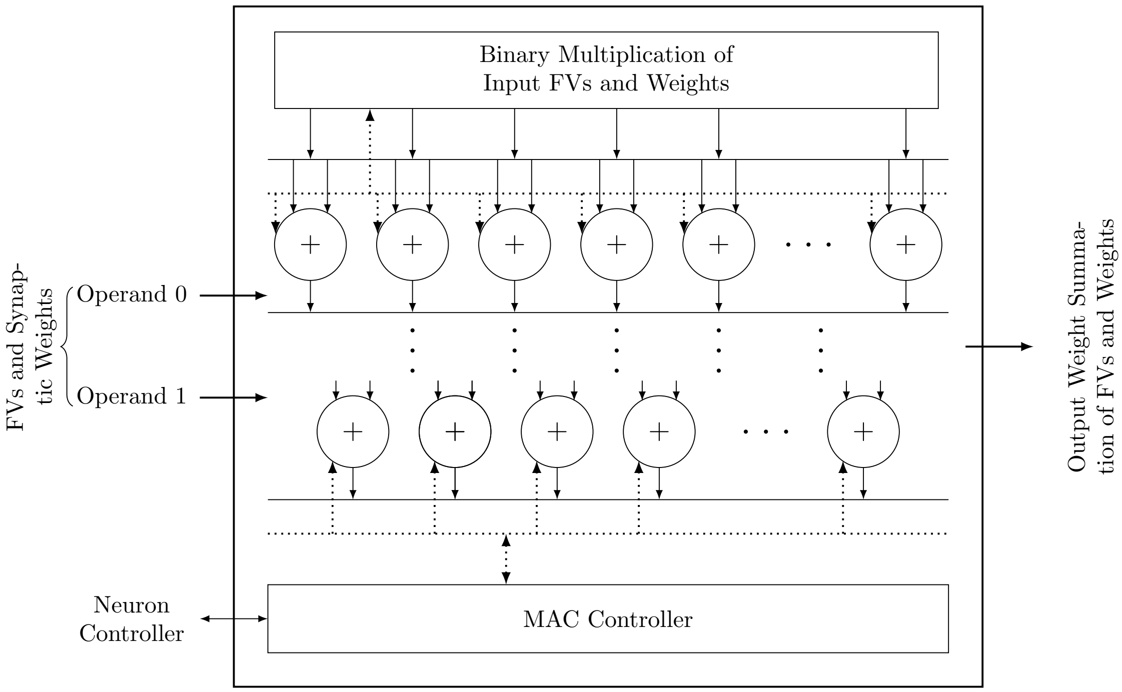

Figure 4.5: The hardware architecture of multiply-accumulate submodule

then combines these results, in parallel, using fully pipelined mixed-width integer accumulation operations. The depth of pipeline is 5 and a throughput of 1 per cycle.

4.3.3 Network-on-chip

Due to physical locality of inter-PE communication, 2-D mesh NoC was designed for local communication within a cluster. The architecture of local NoC is shown in Figure 4.1. Each PE of a cluster has a 5-port router where it is connected to four routers of its neighboring PEs and itself through a 5×5 crossbar switch. The dimension and different properties of this mesh NoC is shown in Table 4.3. The size of flit is 42 bits where 32 bits are used for data and 10 bits for control. The packet size of NoC is 336 bits and the buffer size of each channel is eight flits (very small). The small buffer size helped to design buffer using register files instead of SRAM and reduce the overall area. Since the traffic pattern of inter-PE communication of Sparsey network is deterministic, a deterministic routing scheme (XY) was used designing this mesh NoC. The XY routing scheme is very simple and less expensive in terms of area, power, and latency. The round-robin arbiter with grant-hold circuit was designed giving access to the output port of a PE. The store-and-forward switching strategy was used to implement the data transmission. A generic hardware architecture of input-buffered router (IBR) is also shown in Figure 4.1.

The global interconnection network, *Fat tree*, supports inter-cluster communication. A

Table 4.3: The configuration of 2-D mesh NoC in this design

Configuration Name	Configuration Value
Dimension of NoC	4×4
Flit Size	(32 + 10)-bits (Data + Control)
Flits per Packet	8
Channel Width	42 bits
Channel Buffer Size	8 flits
Channel Buffer Type	Circular
Arbiter Type	Round robin
Crossbar Switch Ports	5×5
Routing Scheme	XY

Table 4.4: The configuration of global NoC in scalability studies

Configuration Name	Configuration Value
Dimension of NoC	16-node 2-ary
Number of Switches	32
Number of Channels	96
Flit Data Width	64 bits
Flits per Packet	32
Channel Buffer Size	64 flits

Table 4.5: The baseline network of three-layer Sparsey

Layer	Macs/Layer	CMs/Mac	Neurons/CM	Total Neurons
Layer 1	400 (40×40)	20	40	320000
Layer 2	16 (4×4)	20	40	12800
Layer 3	4 (2×2)	20	40	3200

cycle-accurate NoC simulator, *bookism*, was configured for scalability studies of this accelerator [66]. The configuration of global NoC is listed in Table 4.4. The dimension of *Fat tree* NoC is 16-node 2-ary, which supports maximum 16 clusters. The traffic pattern of global NoC was kept uniform. The discussion section includes the details role of this global interconnect network.

4.4 Experimental Setup

A three-layer Sparsey network (Table 4.5) of 336×10^3 neurons (four clusters) was optimized for implementation in a CUDA-accelerated platform. The CUDA program was run in a state-of-the-art Tesla P100 GPU with HMB2 high-speed memory, where the host server is a multiprocessor with 48 cores and 256 GB of RAM. The detail configurations of Tesla P100 GPU is shown in Table 4.6. The CUDA program was written in a way that it utilizes the highest number of processing cores and ensures the maximum performance. The CUDA program also uses the population count (*__device__ int __popcll*) function where it packed 64 weight and input parameters into two 64-bit operands and transferred from/to HBM2 memory to/from the memory of CUDA cores. This ensures the efficient use of HBM2 high-speed memory bandwidth. The CUDA profiler shows that memory bandwidth bottleneck consumes more than 80% of the processing time. The KTM action recognition datasets were used [21] to evaluate performance. The feature extractor or front-end was developed in OpenCV using a multi-step edge detection algorithm. Four single input sparse matrices (4×160×120) was partitioned into 16×12 patches and mapped to Macs of the input layer of the baseline Sparsey network. In the baseline network, there are 40 CMs per Mac and 20 neurons per CM. The same Sparsey network and feature vector were used for performance estimation of the accelerator.

The register transfer level (RTL) model synthesis and place-and-route results were used to estimate the area, power, and speed of a cluster. The basic building block of a cluster is PE and a single configurable PE was designed and implemented in RTL. Based on the functional simulation of a PE in ModelSim followed by synthesis using Synopsys DesignWare and place-and-route in GF 65nm standard cell library, the area of a PE was calculated. The

Table 4.6: The hardware specification of Tesla P100 GPU

Name	Tesla P100 GPU
Multiprocessors	56
Core Count	3584
Die Area	610 mm^2
Technology	16 nm
Max Clock Rate	1480 MHz
Memory Size	16 GB
L2 Cache Size	4096 KB
Memory Bus Type	4096-bit HBM2
Memory BW	732 GB/s
Bus Interface	PCIe Gen3
Board Power	300 W

Table 4.7: The total on-chip SRAM requirement for a cluster

Layer	Bottom-up Weights (Kb)	Horizontal Weights (Kb)	Top-down Weights (Kb)	Total Memory (Mb)
Layer 1	15000	312500	62500	381
Layer 2	62500	7500	2500	71
Layer 3	2500	625	0	4
			Total	455

power and timing analysis of a single PE was based on the place-and-route of a PE using Cadence Encounter followed by Synopsys Primetime. The on-chip SRAM in this design was generated using Edbedit in a 65nm standard cell library. A cycle-accurate simulator of mesh NoC was written in C++ to calculate the inter-PE communication latencies. For calculating the overall performance metrics, a single router was designed and implemented in RTL and performed synthesis and place-and-route for performance estimation of the local NoC. The spreadsheet calculation was performed to determine the performance of a cluster.

Table 4.8: The key hardware results of a PE

Name	PE	PE†
Technology (nm)	65	16
Clock Frequency (MHz)	200	1000
Core Voltage (V)	1.0	0.86
Total Area (mm^2)	62.8	6.30
Total Cell Area (mm^2)	3.13	0.52
On-chip SRAM Area (mm^2)	59.6	5.70
On-chip Register (KB)	262	262
On-chip SRAM (KB)	8224	8224
SRAM Area/Total Area (%)	89.89	89.89
Memory Bandwidth (Tb/s)	1.64	8.2
Core Power (mW)	224.31	36.06

†technology scaling as per ITRS roadmap [70]

4.5 Results

This section summarizes the area, power, and timing analysis of a cluster against a Tesla P100 GPU. The performance comparison of a cluster was analyzed using a baseline Sparsey network on the KTH action recognition dataset. The total on-chip SRAM requirement of a cluster for synaptic weights is shown in Table 4.7. The storage of horizontal weights for the first layer of baseline network is significant compared to others. In Sparsey, the top-down weights are optional and were deactivated during runs of the baseline network in GPU and also in the accelerator.

The key hardware results of a PE are also shown in Table 4.8. The PE is synthesized at 200 MHz in 65nm technology followed by place-and-route for calculating the average power. Since this is an SRAM based hardware accelerator, 90% of the area is occupied by the on-chip SRAM. The peak bandwidth of SRAM is 1.64 Tb/s and used for read and write of synaptic weight matrices during each learning and inference iteration. The on-chip register files for shared memory (in PE and CM) and sequential cells add up total 262 KB of on-chip registers in a PE.

The overall results of a cluster of this work are shown in Table 4.9. The 16nm scaling factor (ITRS) was used to estimate the key results of a cluster [70]. The aggregated memory bandwidth of distributed on-chip SRAM is 26.24 Tb/s and 131.2 Tb/s for 65nm and 16nm, respectively. The area of mesh NoC in a cluster is 0.05 mm^2, which is insignificant compared to the area of core and memory of this work. At 16nm, the average power of a cluster to process a single frame of KTH video file is 0.58 W. The average power also includes the power

Table 4.9: The hardware results of a cluster

Name	Cluster	Cluster†
Technology (nm)	65	16
Clock Frequency (MHz)	200	1000
Core Voltage (V)	1.0	0.86
Total Area (mm^2)	1120	106.67
Total Cell Area (mm^2)	90.89	8.66
On-chip SRAM Area (mm^2)	1006.32	95.84
NoC Area (mm^2)	0.525	0.05
On-chip SRAM (MB)	128	128
SRAM Area/Total Area (%)	90	90
Memory Bandwidth (Tb/s)	26.24	131.2
Core Power (W)	3.6	0.58

†technology scaling as per ITRS roadmap [70]

Table 4.10: The area, power, and bandwidth comparision against GPU

Name	Tesla P100 GPU	This ASIC	Inprovement
Technology (nm)	16	16	–
Die Area (mm^2)	610	426	1.43×
Average Power (W)	34.13	2.44	13.99×
Memory Bandwidth (GB/s)	5.72	524.8	91.75×

Table 4.11: The processing time of GPU and this work

Type of Processing	Total Neurons	Tesla P100 GPU (µs)	This Work (µs)	Inprovement
Layer 1	320000	704.71	31.35	22.48×
Layer 2	12800	617.06	10.21	60.44×
Layer 3	3200	118.34	4.44	26.65×
NoC	–	–	11.06	–
Total	336000	1440.11	57.06	25.24×

Table 4.12: The operations per second comparision for hotspot processing

Type of Processing	Total Neurons	Tesla P100 GPU (GOPS)	This Work (GOPS)	Inprovement
Layer 1	320000	3807	85582	22.48×
Layer 2	12800	996	60176	60.42×
Layer 3	3200	216	5765	26.70×
Total	336000	5019	151523	30.20×

of mesh NoC for intra-PE communication. The area, power, and bandwidth comparisons of a cluster and GPU are reported in Table 4.10. In this implementation, there is approximately 92× bandwidth improvement for accessing synaptic weight matrices. In addition, the on-chip SRAM and fixed-point arithmetic reduces the overall power significantly and improves approximately 14× compared to the power (calculated from NVIDIA profiler, *nvprof*) of GPU. The fixed-point arithmetic also reduces the die area by 1.43×. Comparison of processing time at each layer for Sparsey baseline implementation is shown in Table 4.11. The processing time of this work also includes the communication time among PEs in a cluster. The total communication time among PEs for different types of intermediate inputs/activations is 11.06 µs. The overall processing speedup of a frame is 25.24× than that of Tesla P100 GPU. The operations per second comparision for the hotspot processing is also shown in Table 4.12. The operations per second (GOPS) improvement (for the hotspot) in this implementation is approximately 30×. Four clusters have a total of 16384 neurons in the silicon area of 426.68 mm^2 in 16nm technology to achieve this acceleration.

4.6 Discussion

In the previous section, four clusters were sufficient to accelerate the Sparsey baseline network against the Tesla P100 GPU. However, many applications and problem sizes require one and sometimes more than four clusters. Also, the primary motivation of Sparsey is to model the episodic memory of human where it can have as much as 2×10^{10} neurons. Thus, this section discusses the scalability studies of this accelerator.

4.6.1 Inter-cluster Communication

The scalability of this accelerator relies on a scalable interconnection network for inter-cluster communication. As discussed in the section of implementation details, the *Fat tree* topology was chosen for scalability studies. The configuration of *Fat tree* interconnect network was given in Table 4.4. The dimension (16-node 2-ary) of this *Fat tree* supports a maximum of 16 clusters, i.e. a total 65,536 neurons in silicon, that can be configured up to 270 million neurons (at least 75$\times$ less than a human brain). The reason for choosing *Fat tree* network as global communication was that it scales better for a hierarchical system and is cost effective. In addition, it has less impact on performance because of static routing [67, 68]. The communication latency of global interconnection network has a major role in the overall performance than the area and power of the global interconnect network. Thus, the *BookSim* simulator was used to model the global interconnect network for studies of the hierarchical model of this accelerator [71]. The overall architecture of the proposed hierarchical accelerator is shown in Figure 4.6.

4.6.2 Scalability

In the scalability studies, the increase of silicon area was maintained with the increase of feature vector size. For instance, the area of silicon (the number of clusters) increases twice for every doubling of the input FVs count and also adds an extra layer on top of the existing three-layer Sparsey network. There are two possible scenarios for global communication: 1) each cluster is an independent network and edge-PE of clusters are not required to communicate with PEs of its adjacent clusters; 2) edge-PEs of the adjacent clusters communicate with each other. The second scenario is an exhaustive one (every cluster communicates with every other clusters) and is considered for this scalability analysis. In addition, clusters are configured similar to the baseline network (i.e. 100 PEs/cluster, 20 CMs/PE, and 40 neurons/CM). However, it is possible to configure a cluster in a different way, the trend of overall performance matrices will be somehow similar except the performance of global (or inter-cluster) communication. Figure 4.7 shows the performance metrics (area, speed/frame, and also power/frame) using 16nm technology with increasing the number of FVs per PE as well as the number of clusters. In this exhaustive

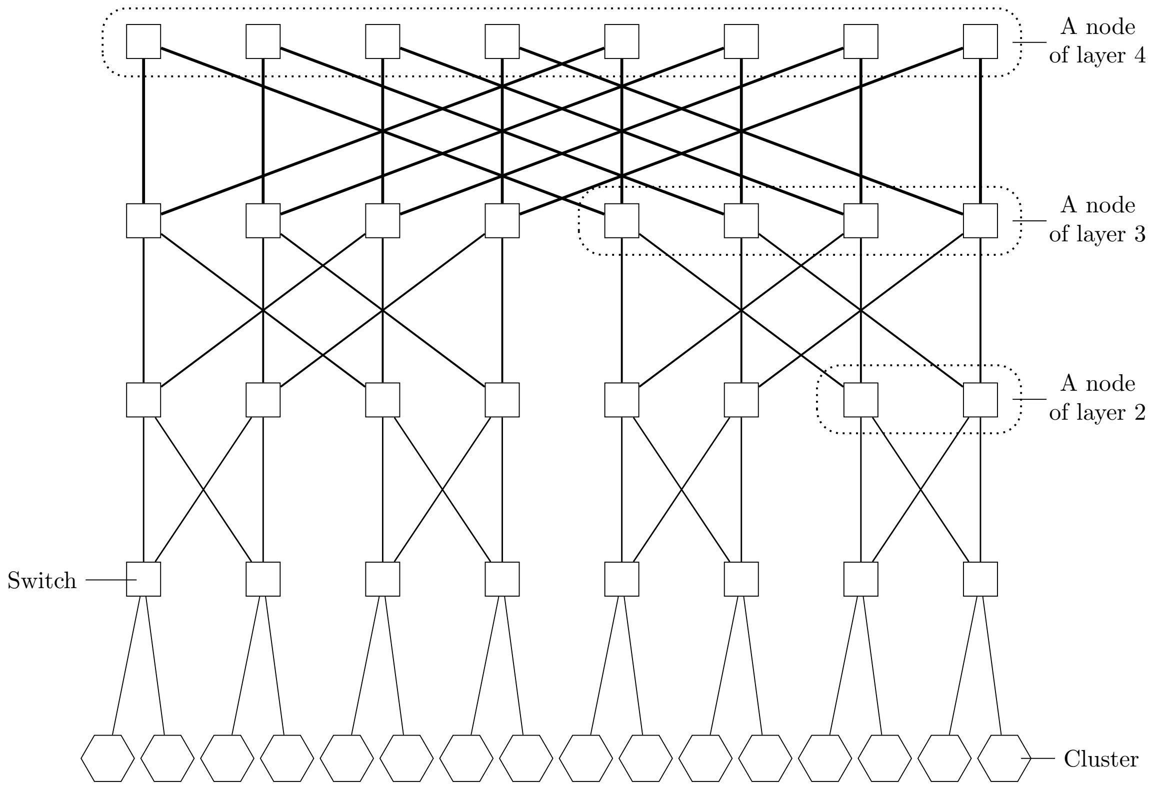

Figure 4.6: The global interconnection network connects 16 clusters

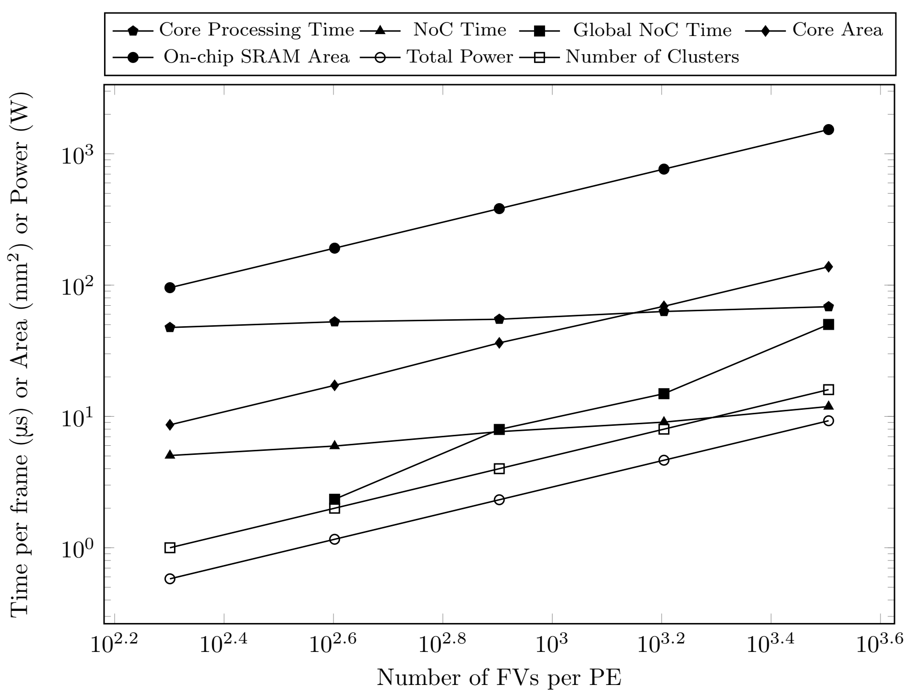

Figure 4.7: The performance and scalability of hierarchical accelerator

scenario, the inter-cluster communication latencies dominate the overall processing time when the number of clusters goes beyond sixteen. On the other hand, the intra-PE communication time (local NoC) and the core processing time of clusters remain constant, the area of on-chip SRAM and core increases linearly though. Finally, the computations of Sparsey are usually scaled linearly with the hardware resources but not the communication.

4.7 Conclusion

This chapter presents a scalable, configurable, and hierarchical hardware accelerator of Sparsey, a cortical algorithm that models Sparse Distributed Memory (SDM) and learns in an unsupervised manner. This work demonstrated that a custom hardware would give performance benefits over a server-class GPU. The performance improvement was obtained by the acceleration of memory access using two types of memory management at different levels of its parallel computation. The first one is the neuron-level storage for synaptic weight matrices and the second one is the shared register file for intermediate/temporary values. The distributed neuron-level storage enhances the amount of local memory as well as the memory bandwidth. The fixed-point arithmetic also helped to improve the area, power, and speed performance. Lastly, the design of local NoC and its topology based on routing and traffic pattern of Sparsey network contributed to the overall performance enhancement.

Chapter 5

An ASIP for RNNs

5.1 Introduction

Modern recurrent neural network (RNN) has many variants and some variants offer higher accuracy for a particular application than others. For instance, bidirectional RNN gives the state-of-the-art performance on speech processing [72]. RNNs usually require higher computational complexity and massive storage for a large number of parameters. The Long Short-Term Memory (LSTM) requires much higher storage and computational complexity than regular RNN. In addition, the training of LSTM requires much higher storage and computation than the inference. Moreover, LSTM itself has many variants and a group of variants gives better performance for specific representative tasks (speech recognition, handwriting recognition, and music modeling) compared to other groups [41]. Thus, the performance and flexibility are the two significant requirements of a hardware platform for efficient RNN implementation. Due to these requirements, an application-specific instruction set processor (ASIP) is designed and implemented to improve the memory bandwidth and parallelism, which will increase the computations and at the same time maintain the flexibility for network variants.

In the following sections, the design space of LSTM network in hardware is analyzed for efficient ASIP implementation. The implementation details and various aspects of ASIP architecture are discussed in the subsequent section. It also presents the hardware modeling methodology, and performance measurement processes on the language modeling dataset. The speedup and energy efficiency are demonstrated in the result section. The flexibility and scalability are discussed in the discussion section followed by a conclusion.

5.2 Design Space Exploration

In this section, different aspects of computation, parallelisms, and memory access patterns are analyzed for the LSTM network in order to set design guidelines of the proposed ASIP architecture.

5.2.1 Computation

The types and amounts of computations are different for three different stages, forward propagation, backward propagation, and mini-batch Gradient Descent or Adam, for a given LSTM network. The prevalent computation types during forward propagation are matrix-vector multiplication, element-wise matrix-matrix multiplication, element-wise matrix-matrix addition, and different special function operations (usually Sigmoid, Tanh) immediately after matrix-vector multiplication. The matrix-vector multiplications are the most computationally expensive in hardware during forward propagation. A mini-batch of B samples contain T time steps requires $8mnBT$ number of MAC operations, where m is the dimension of inputs (or input features) and n is the dimension of hidden states (h_t). On the other hand, the element-wise and special function operations (SFUs) are approximately $3nBT$ and $5nBT$, respectively. The upper bound of computational complexity during forward propagation is $O(n^4)$.

The backward propagation requires matrix-matrix multiplication, differentiation of non-linear activation functions, and element-wise addition operations. For the same LSTM layer, the total MAC operations involve with matrix-matrix multiplication are approximately $16mnBT$ and the differentiation of non-linear activations, element-wise multiplication, and element-wise additions are nBT, $14nBT$, and $(2nB + 8mn)$, respectively. The upper bound of computation during backward propagation is also $O(n^4)$. The iterative optimization technique, mini-batch Gradient Descent (GD) or Adam, is used to minimize the error or cost function and require scalar-matrix multiplication, element-wise addition, square root, and element-wise division. The total number of multiplications, additions, square roots, and divisions needed to perform Adam optimization for a single LSTM layer are approximately $48mn$, $32mn$, $8mn$, and $8mn$, respectively. However, the number of operations in mini-batch GD and Adam can be higher based on their variants [73, 74].

5.2.2 Memory

Memory system plays the most crucial role and the biggest challenges in the hardware design of the LSTM network. There are different memory options available in hardware design (on-chip and off-chip memory) to support a specific mode of operation (training or inference) or both. The various aspects of the memory system are analyzed in the following section to choose the right options in hardware design and accelerate training and inference for both batch or non-batch operation.

Memory Size

The connectivity between layers and also within a layer of an LSTM network and scaling LSTM network into much larger sizes requires massive storage for weight parameters and intermediate activations. For example, the 34 LSTM layers and 24 fully-connected (FC) layers of Tensor Processing Unit (TPU) requires 52 million weight parameters to perform forward propagation only [27]. The 32-bit floating-point representation of each connection needs approximately 1.6 Gb of storage for weight parameters. In addition to the weight parameters, LSTM requires a huge number of samples or examples to perform training. In [14], the four LSTM layers and one fully-connected (FC) layer have total 384 million parameters ($\sim$12 Gb in single-precision), where 240 million ($\sim$7 Gb) parameters are for the input and output vocabulary. The total storage requirement for weight parameters of a given LSTM layer can be calculated from $8mn$, where m is the dimension of input (or feature) vector and n is the dimension of hidden states (h_t). Thus the space complexity of weight parameters for an LSTM layer is $O(mn)$.

The on-chip memory is expensive in term of area per storage and will not be practical to add a large amount of on-chip memory in the hardware. The off-chip memory, dynamic random-access memory (DRAM), is the only solution to overcome this limiting factor. However, the interface between DRAM and processor incurs various access latencies and adds a considerable amount of power overhead due to access the off-chip data. Also, DRAM has limited memory bandwidth (especially non-batch operations) to support a large amount of data during training and inference of an LSTM network.

A small on-chip memory is useful for holding partial input or weight parameters and intermediate activations during training and inference. The use of appropriate sizing on-chip memory can potentially relax the bandwidth demand of off-chip memory and reduce the power overhead significantly. During training and inference, the network needs to perform a large size matrix-matrix, matrix-vector, matrix-scalar multiplication, and matrix-matrix elementwise addition. The block matrix mechanism can be used to break weight and input matrices into a collection of submatrices especially compliant with the size of on-chip memory and perform the computation of submatrices [75]. The block matrix mechanism increases the data reuse, decreases the off-chip memory bandwidth demand, and the power accordingly. Therefore, the flexible hardware is required to partition different matrices and vectors efficiently especially for different network dimensions. In addition, a second piece of on-chip memory can potentially help to prefetch the next submatrices for subsequent block operation, which can increase the overall computations. The details mechanism of block operation of this hardware design is shown in Figure 5.1.

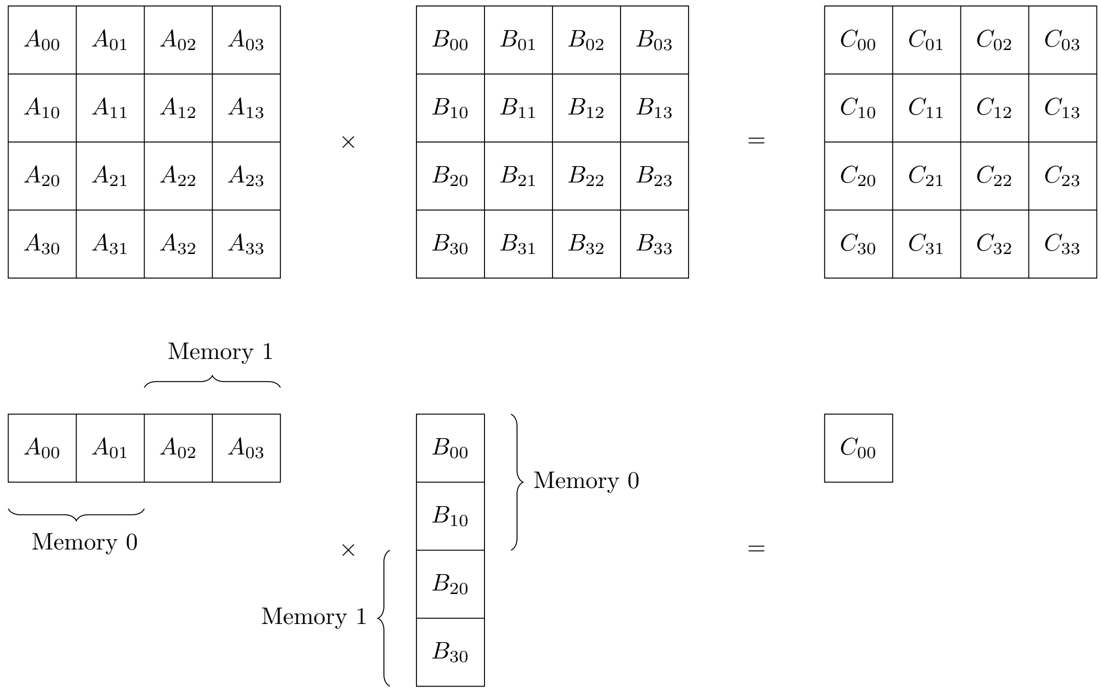

Figure 5.1: Matrix multiplication using block matrix

Memory Bandwidth

A network with multiple LSTM layers and fully-connected (FC) layers requires very high memory bandwidth and interface speed to run on hardware at peak speed. In other words, the memory bandwidth is crucial to accelerating LSTM compared to many other deep learning applications. For instance, CNNs usually deal with sparse data and achieve very high computation (86 TOPS) with limited DRAM bandwidth (32 GB/s), whereas MLPs and LSTMs are memory bound (worst-case weight stall time is 62.1% of the total processing) [27]. Moreover, the number of samples in input matrices also determines whether block matrix computation (with the help of on-chip memory) can relax the off-chip bandwidth demands or not. The bandwidth requirement increases with the decrease of samples in the input matrix. The worst-case bandwidth demands occurred when there is a single example in the input matrix (i.e. online or edge training and inference). The computational complexity and space complexity are equal at worst-case processing ($O(mn)$) and the off-chip memory bandwidth should be close to the peak processing speed (operations/second) for the highest degree of acceleration. Thus, the on-chip memory has a limited role in improving the demand for off-chip memory bandwidth during worst-case processing (online training and inference).

Memory Access Pattern

Unlike inference mode, the training of an LSTM network has a pattern of irregular data access from the off-chip memory instead of contiguous behavior. During gradient and error calculation through the back-propagation algorithm (Algorithm 2.4), the hardware needs to perform matrix-matrix multiplication of input and weight parameters with a transposed form of matrices. Such an irregular form of off-chip memory access lead to memory bandwidth saturation even though the processor is not [52]. In addition, the processor can have the maximum achievable off-chip memory bandwidth only if the off-chip memory transactions are on burst mode. The irregular pattern of memory accesses makes the off-chip memory much harder to operate on burst mode, which degrades the effective bandwidth during training. Moreover, the recent emerging 3D-DRAMs (DiRAM4) works only on burst mode and random access could potentially reduce the memory bandwidth to one-third (16-bit data from 64-bit wide cache line) of the maximum achievable bandwidth during training [76].

5.2.3 Mapping Networks onto Hardware

Different levels of inherent parallelism (operation and data) are available in the processing of LSTM networks. The highest level of concurrency and acceleration is possible when computations are mapped to specific hardware analyzing its inherent parallelisms. The concurrency analysis of an entire LSTM network is required along different dimensions in order to harness its parallelism. The most common types of parallelism and partitioning mechanism of LSTM network are discussed in the following section.

Model Parallelism

The computation of a single layer is independent for a specific sample or mini-batch of samples. The straightforward approach to accelerate a layer is to split computation among available compute units (or processing elements). Three different processing (forward propagation, backward propagation, and weight update using mini-batch GD or Adam) of an LSTM layer are also independent for a sample or mini-batch of samples. The input operands of each processing are weight parameters and examples. The weight parameters are partitioned across memories associated with all PEs, which operate on the data in parallel. A sample or mini-batch of samples, stored in memory, should be accessible from all PEs and duplicated among memories of all PEs. The use of model parallelism of a layer requires communication for sharing the output activations among PEs. The communication involves in MLP and LSTM can be achieved allocating off-chip dedicated memory associated with each PE and accessing those memories with proper coordination. The direction of model parallelism is in the direction of the breath of layers in a DNN. The concept of model parallelism is shown in Figure 5.2.

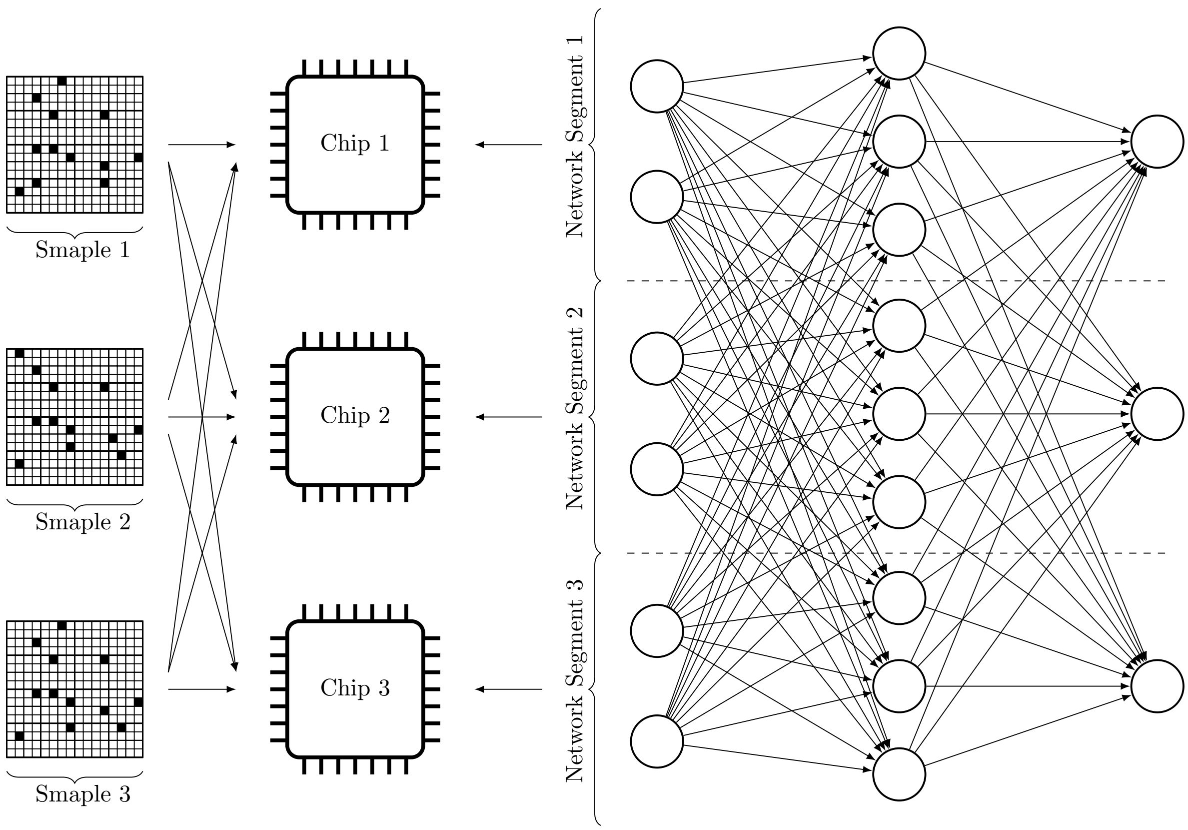

Figure 5.2: Model parallelism of network in hardware

Data Parallelism

Data parallelism seeks to partition the computation of a layer for the size of samples or mini-batch and it is only possible during batch processing. In mini-batch processing, layers process a single or subset of examples from a mini-batch at a time and the processing (forward propagation, backward propagation, and weight update) is independent of each other. Thus the processing samples of a mini-batch can be partitioned among all PEs where each PE will have access to the entire LSTM weight parameters and a fraction of samples from the mini-batch. The weight parameters of LSTM network are usually replicated to the memories associated with PEs. The mini-batch sizes determine the scaling of data parallelism and it is limited if the mini-batch size is less than the number of PEs. For example, if there is a mini-batch of 128 samples and the total number of PEs is 32, then a single PE is needed to process the entire LSTM network for four samples only. Synchronization among PEs is essential especially when an LSTM network is mapped to the hardware in a data parallelism fashion. Figure 5.3 shows the network mapping to hardware as data parallelism fashion.

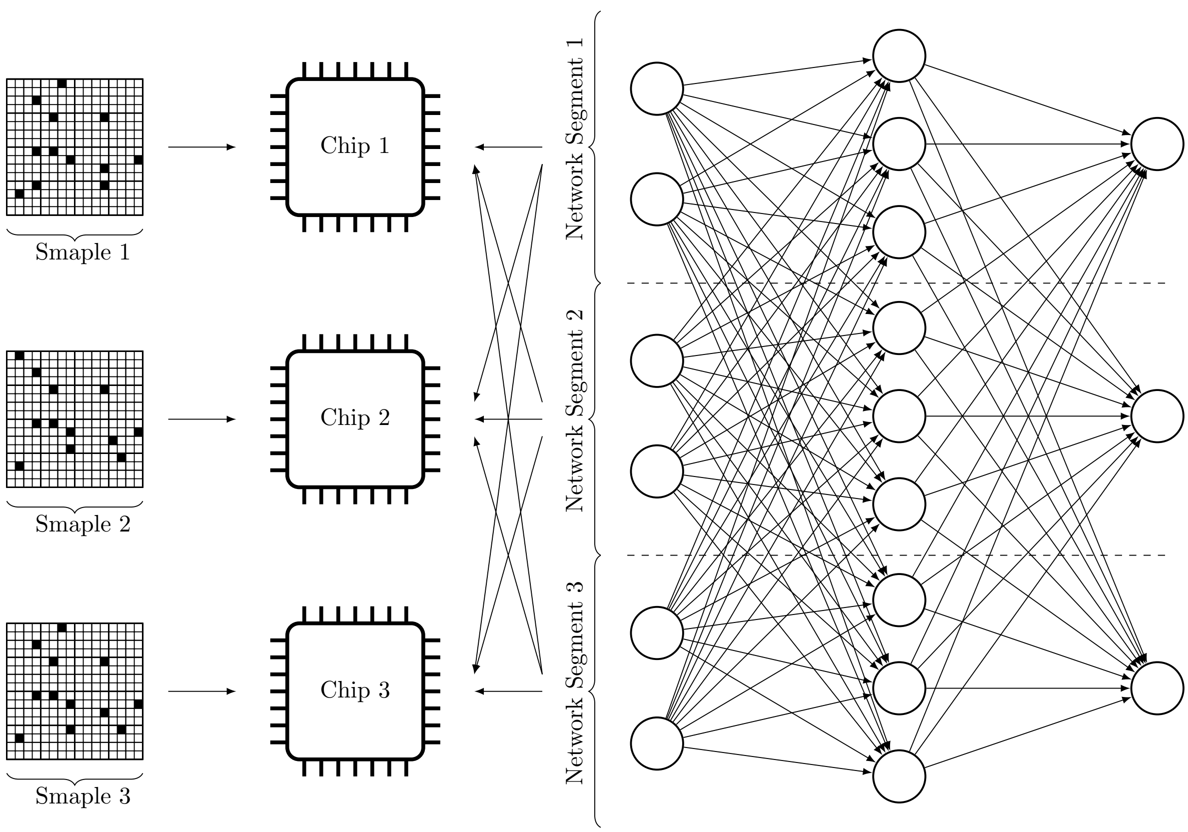

Figure 5.3: Data parallelism of network in hardware

Pipelining

In the direction to the depth of an LSTM network, the computation of layers is sequential but can have overlapping computations between layers if data is available. An LSTM network can be partitioned along depth and assign data and processing of each segment to specific PE. The combination of model and data parallelism is similar to the pipelining in the sense that samples are processed through the depth are parallel and the length of pipeline is related to the depth of LSTM network. In pipelining, the overlapping computations of forward propagation, backward propagation, and weight update are scheduled instead of synchronizing computation layer by layer. Pipelining of LSTM network increases the utilization of hardware and the bandwidth of off-chip memory (i.e. decreases the processing time).

As the mini-batch sizes are fixed, there is no tradeoff between model and data parallelism. Data parallelism is suitable when data and weight parameters are fit into the memory associated with a PE, and the mini-batch sizes are large. The off-chip memory sizes of hardware should be huge for mapping an LSTM network into hardware as data parallelism fashion. However, partitioning an LSTM network based on pipelining is better considering that a single PE does not need to store the weight parameters of the entire the network and the communication is deterministic (source and destination PEs are known even before processing). Overall, the hardware should have sufficient off-chip memory and flexibility to support different parallelism suitable for different LSTM networks. This work provides accelerators with high-density off-chip memory as well as instructions to enable flexible mapping of different LSTM networks into the hardware.

5.2.4 Precision

Fixed-point (quantization of floating-point number) representation of weight parameters for a trained LSTM network and implementation that fixed-point network into hardware is usually good for inference accelerator. The area of 8-bit fixed-point adder and multiplier are 38× and 5.8× less, and the energy advantage are 13× and 5.5× higher than that of 16-bit floating-point adder and multiplier [26]. As discussed in the state-of-the-art chapter, there is a number of recent publications on training algorithms of RNN, where it either uses few numbers of bits for weights and activations or binarize the network [63, 51, 77, 78]. However, some of these techniques either suffer from small accuracy loss, perform on small size network, and involve floating-point computation in GPUs.

The training of LSTM using half-precision (instead of single-precision) floating-point can decrease the amount of memory, the pressure of memory bandwidth, and increase the performance of arithmetic units. The weight gradients of half-precision training are sometimes too small ($<2^{-24}$) to represent in 16-bit and drop the relative training accuracy significantly [30]. The state-of-the-art mixed precision training can overcome this limitation and achieve the training accuracy as much as the single-precision training [30]. In mixed precision training, the entire storage of weight parameters, gradients, and intermediate activations are in half-precision and a copy of single-precision weight parameters are maintained. The forward and backward propagation are performed using half-precision storage (halving the storage and bandwidth) and update the weight parameters in single-precision. Also, the mixed precision training requires

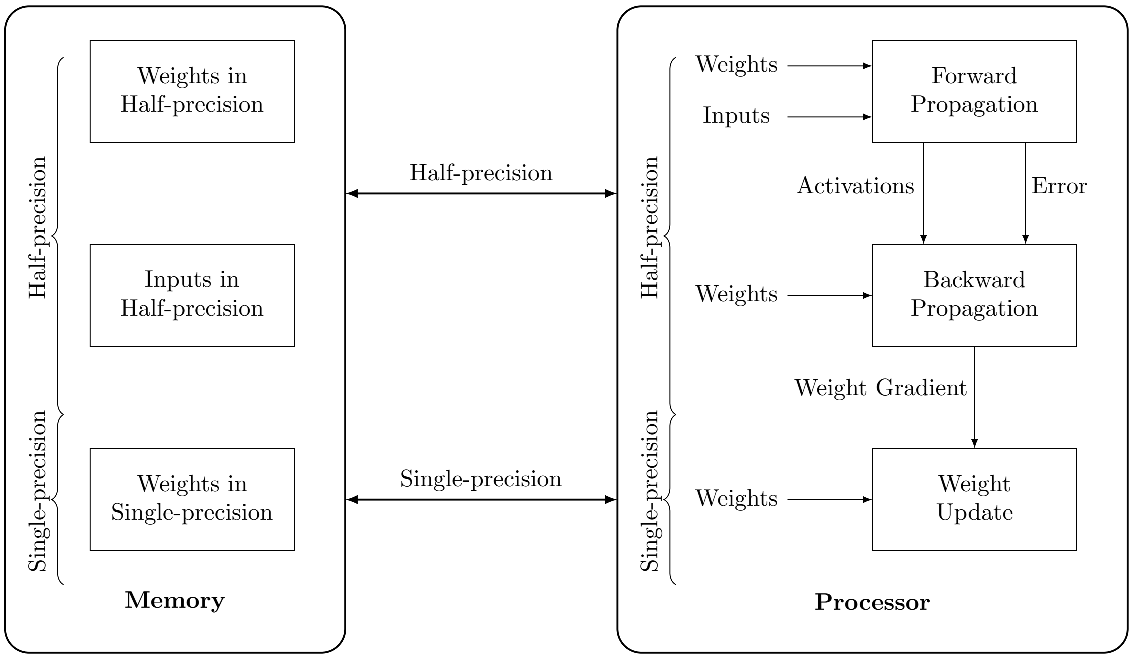

Figure 5.4: The top-level concept of mixed precision training

loss scaling before backward propagation due to shift the gradient values into a representable range in half-precision. In other words, the multiplication (or dot product) is performed using half-precision and accumulation in single-precision and converted back to half-precision before written back to the memory. In [30], the mixed precision training of most state-of-the-art DNNs (AlexNet, VGG-D, GoogleLeNet, Inception v1, Inception v2, Inception v3, Resnet50, R-CNN, Multibox SSD, variants of LSTM, and also GANs) maintain the similar accuracy of the single-precision training for state-of-the-art datasets. This work offers special instructions and also hardware to support the state-of-the-art mixed precision training for maintaining the training accuracy and, at the same time, relaxing the pressure on memory bandwidth which improves the hardware performance. The overall concept of the mixed precision training is shown in Figure 5.4.

5.3 Hardware Implementation Details

The details of proposed ASIP accelerator for training and inference of RNNs are presented in this section. This section also presents the features of ASIP architecture to address the challenges discussed in the previous section.

5.3.1 System Overview

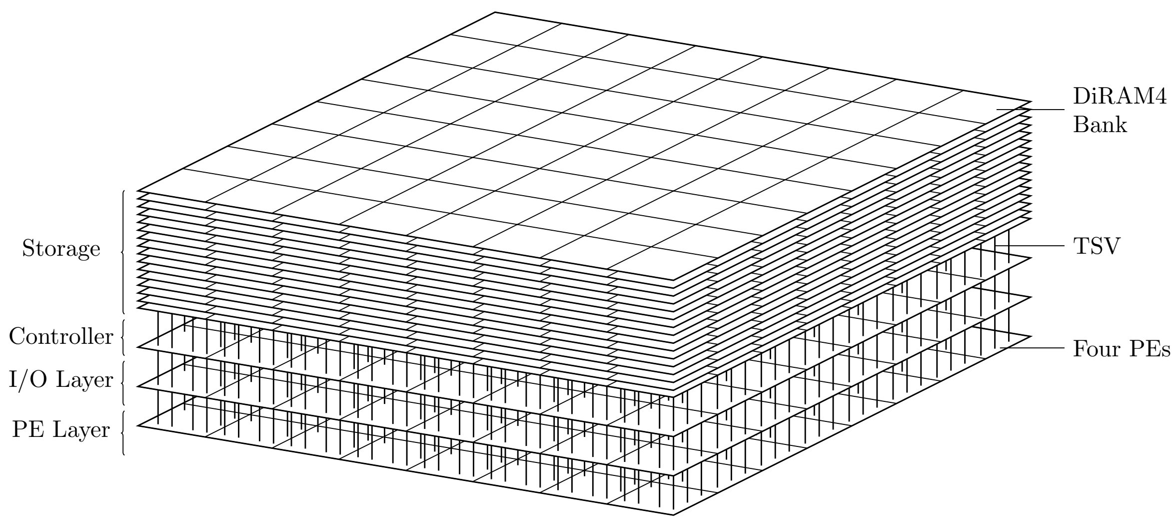

Figure 5.5: The overall architecture of proposed system

A schematic representation of the overall proposed architecture is shown in Figure 5.5 and Figure 5.6. It is a 3D architecture to accelerate the training and inference of LSTM, MLP, and potentially other DNNs. The entire system can be divided into two different sub-systems: 1) the accelerator (multiple parallel PEs); and 2) the 3D memories are connected through the on-chip crossbar network. The host with CPU and host memory communicates to the accelerator and 3D memory using host interface of this accelerator. The overall concept of this design is that the host will off-load the entire computations or the hotspot of LSTM computations to this accelerator. The host transfers the instructions, samples of input, weight parameters and hyperparameters to the 3D memory and passes the initial configurations and memory allocations to the accelerator at the beginning of the computation. The high bandwidth Tezzaron DiRAM4 is used as 3D memory in this architecture. A C++ program is written to analyze the LSTM network and generate sequence of VLIW instructions (for memory transaction and also computation) of this accelerator. The accelerator performs forward propagation, and weight update

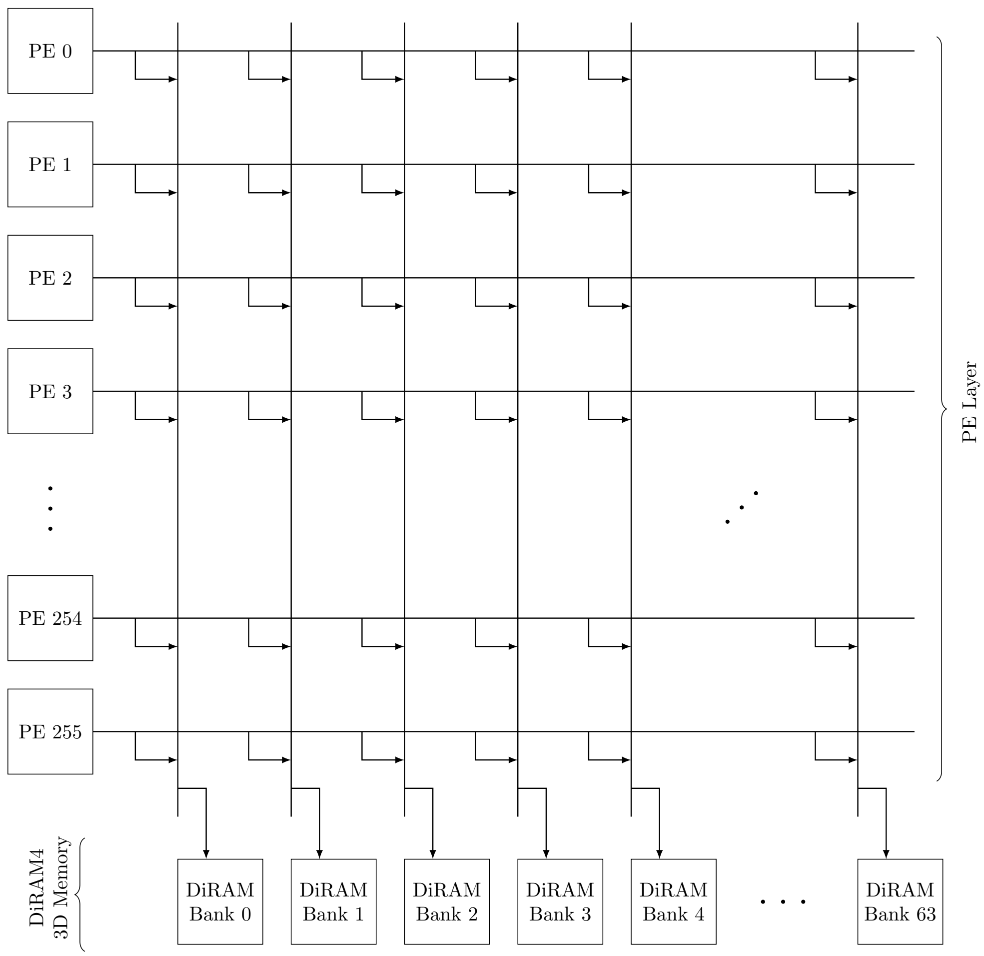

Figure 5.6: PEs connected to DiRAM4 banks using crossbar

stages for an LSTM network using different memory accesses operations to bring data inside PE from DiRAM4, load-store operations for data movement inside a PE, conduct computations for different matrix-matrix, matrix-vector, special function operations (Sigmoid, Exponential, and Tanh) for different activations and at the end, store the output or intermediate results to the DiRAM4 using DMA operations. In case of training, the backward propagation is performed using a smart DMA controller to bring weight or intermediate parameters in the reverse order and perform different operations (load-store, matrix-matrix, special function operations, and others) inside PEs accordingly. The operations involved at three different stages (forward propagation, backward propagation, and mini-batch GD or Adam) are performed using custom instructions designed for this accelerator.

5.3.2 Memory and Data Allocation

DiRAM4 3D Memory

The memory system has a significant impact on the overall performance of this accelerator. A banked and multi-ported 3D memory system (DiRAM4) is used with multiple identical PEs to perform the parallel computation for training and inference. It is a heterogeneous 3D stacked memory system with 18 dies (16 layers contain memories, 1 layer containing sense amps, drivers, decoders, and 1 layer containing I/O buffer and other support circuits), where it has 64 DiRAM4 banks associated with 64 ports [76]. The storage of each DiRAM4 bank is 1 Gb with a total of 64 Gb storage in this memory system. The storage of individual DiRAM4 bank is arranged in 64 banks, in which 4096 pages per bank and 4096 bits per page reduce bank conflict significantly. The 20-bit unidirectional address and control bus, a 32-bit unidirectional input data bus, and a 32-bit unidirectional output bus can transfer data to/from host or PEs through the on-chip crossbar network. The DiRAM4 has Dual Single Data Rate (DSDR) interface on 64 external ports with two channels to the users. Each channel operates on its own clock which is derived from the port clock (90 degrees out of phase from each other). Thus, the data rate per channel is single data rate, but the net data rate from/to the DiRAM4 port is two times the single channel (DSDR). In addition, DiRAM4 can be configured either as a burst of 2 version (64-bit quantities) or burst of 8 version (256-bit quantities). Each port operates on a 1 GHz clock and maximum achievable bandwidth can be as high as 8 Tb/s (read bandwidth 4 Tb/s and write bandwidth 4 Tb/s) from 64 disparate DiRAM4 banks. The host CPU transfers the network parameters and instructions from host memory to DiRAM4 banks using host interface. The proposed accelerators have 256 identical PEs connected to 64 DiRAM4 banks. The overall architecture of a single DiRAM4 bank is shown in Figure 5.7 and the access latency and energy per operation are listed in Table 5.1 and Table 5.2.

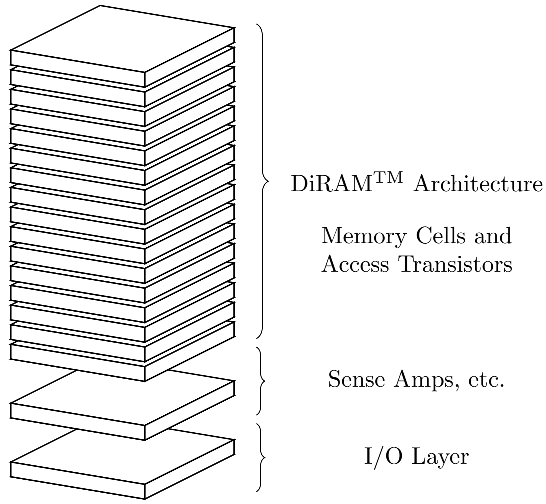

Figure 5.7: A single bank of DiRAM4 3D memory

Table 5.1: The access latency per operation

Operation	Time (ns)
Page Open	22
Cache Access	5

Table 5.2: The energy comsumption per operation

Operation	Energy (pJ)
Page Open	100
Page Refresh	320
Page Close	320
Cache Read or Write	64
No Operation	20

Data Allocation

The access of DiRAM4 has high data locality during forward propagation considering that parameters are stored in row-major order. However, the transpose of a matrix/vector is required during backward propagation where the data access pattern is irregular and locality is less. The irregular access of data reduces the effective data bandwidth which makes the training of LSTM very difficult. The multi-ported DiRAM4 containing multiple banks enables multiple pages (up to 4096) to be opened simultaneously and provides low latency data access (maintain bandwidth) after careful mapping of the network parameters into the DiRAM4 and scatter-gather operation inside PEs. The LSTM network is split into 64 identical segments along the depth of network as per model parallelism of the network and is stored into DiRAM4 banks in an interleaved fashion. The primary purpose of interleaved memory instead of flat memory is to compensate for high latency memory access during forward and backward propagation. In interleaved memory, the next piece of data is ready immediately after the current piece of data is used. As a result, memory reads or writes from contiguous addresses always alternate the bank address inside a DiRAM4 bank and increase the throughput of access by reducing the bank conflict. For instance, the logical address 0 of DiRAM4 belongs to the bank 0 and the logical address 1 belongs to bank 1 and so on.

A single DiRAM4 bank is 64 way interleaved and can continuously open pages in different banks instead of waiting to open a page in the same bank. As discussed in the previous section, DiRAM4 works either on burst 2 or burst 4 mode. In burst 2 mode, the single request (read/write) to memory will be on 64-bit data from the cache line. The details of "storage (or address mapping) scheme targeting" to minimize the bank conflicts during both forward and backward propagation of the LSTM network for a single DiRAM4 bank are discussed here.

The close-page address mapping scheme is followed when a single DiRAM4 bank has two channels and consecutive cache line addresses are mapped to these two channels. The subsequent addresses are then mapped to different banks, and then to a different page. The address mapping scheme of a DiRAM4 bank can be denoted by $P : N : B : C : D$, where D is the data bits per cache line, C is the number of channels per DiRAM4 bank, B is the number of banks per channel, N is the number of cache line per page, and P is the number of pages in a bank. The upper bound of P, N, B, C, and D of a single DiRAM4 used in this implementation are 4096, 64, 32, 2, and 64, respectively. The storage capacity of a single DiRAM4 bank is 1 Gb for DiRAM4-64C64 product family [76]. Figure 5.8 shows a 64-way interleaved, and 64-bit word addressable memory.

The parameters, inputs, and intermediated activations of the LSTM network are stored in row-major order, where a single address stores four consecutive parameters (in 16-bit precision). A DiRAM4 bank can keep open 64 pages simultaneously and can make 16384 parameters

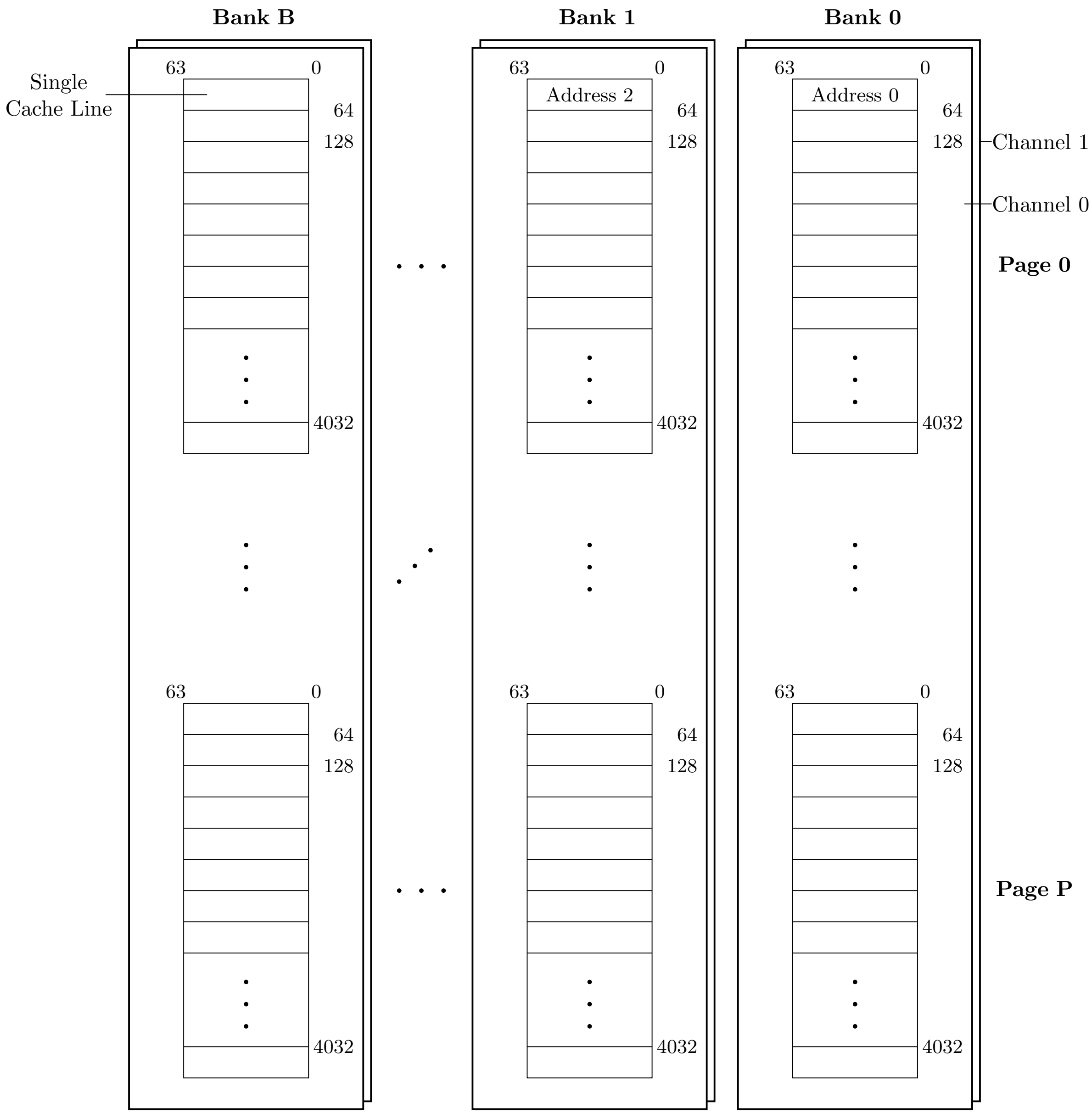

Figure 5.8: The configuration of DiRAM4 bank for version burst of 2

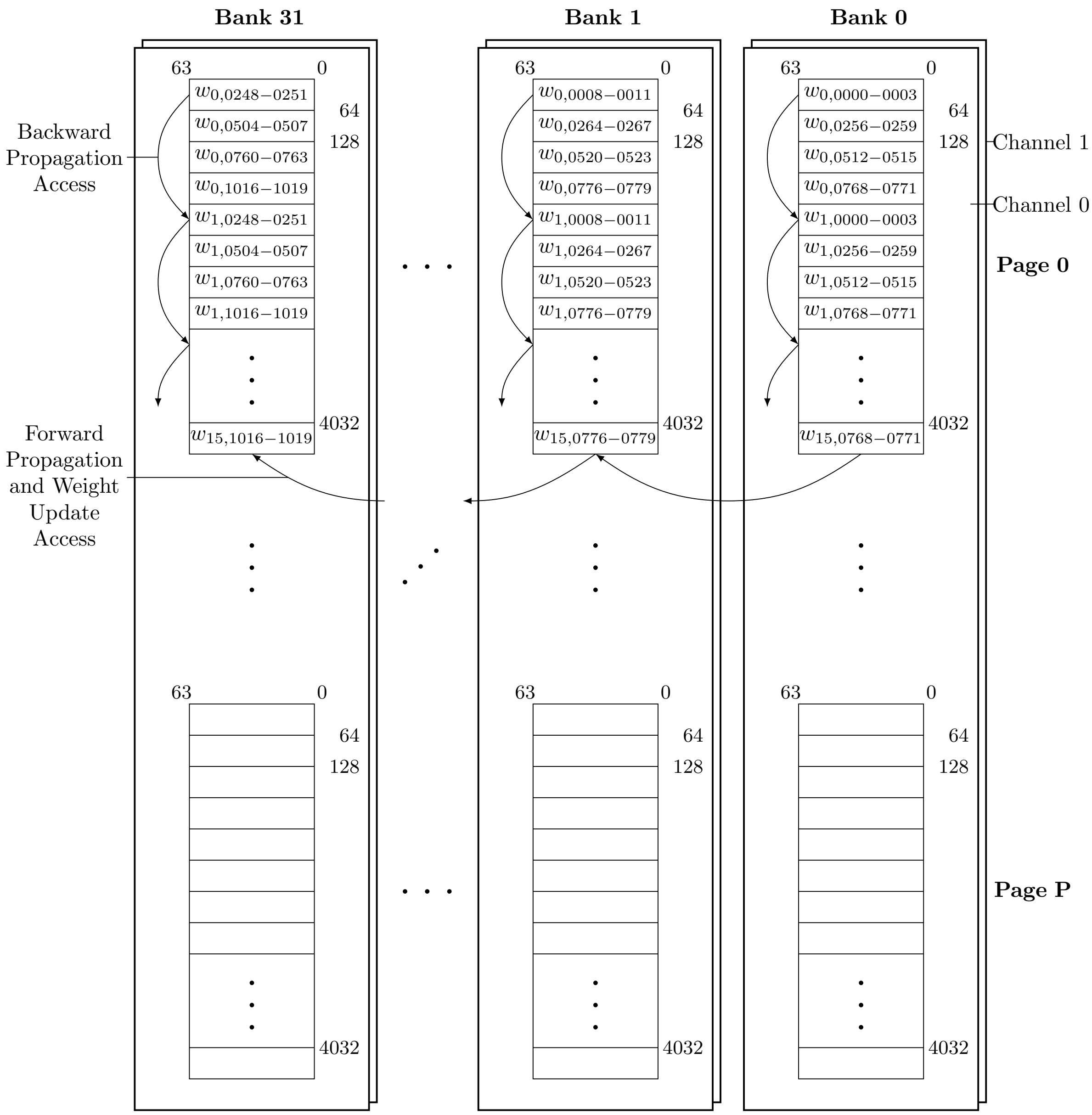

Figure 5.9: The sorage and access pattern of a DiRAM4 bank

available, which is equivalent for 8 LSTM units of an LSTM layer or 16 units of an output layer for an LSTM network with 1024 LSTM units and 16384 output units. In this storage mechanism, the forward propagation and weight update stages read and write the parameters from contiguous address location without any bank conflict. However, the backward propagation needs to read the parameters from increased or decreased memory address location by 512 and 256 for an LSTM layer and an output layer, respectively. A single bank conflict occurred during parameter read of every 8 LSTM units of the LSTM layer and 16 units of the output layer. A programmable DMA controller increases (or decreases) addresses and unique functionality of DMA can scatter and then gather the cache line data for backward propagation, which are discussed in the next section. Figure 5.9 explains the parameters stored in a DiRAM4 bank and the memory access pattern for three different stages. The $w_{i,j-j+3}$ represents the connection/weight value from neurons j, $j+1$, $j+2$, $j+3$ to i. For instance, the weight $w_{0,0-3}$ stored in cache line 0 of bank 0 and page 0 associated with channel 0 represents the weight values from neurons 0, 1, 2, 3 of a layer to neuron 0 of another layer or same layer (in case of recurrent weights). The forward propagation and weight update stages always access consecutive weights ($w_{i,j-j+3}$) from different banks and the backward propagation access weights ($w_{i,j-j+3}$) from the same bank (i.e. in the direction of the page in a bank).

5.3.3 Processing Element

An individual layer of LSTM processes either one (edge learning or inference) or multiple inputs (batch learning or inference) and generates one or batch of outputs accordingly. The processing element (PE) is designed in a way that it can either process one or multiple inputs. There are 256 PEs in this architecture that runs independently (i.e. each PE perform computation a subset of entire processing). For example, an LSTM network (two hidden layers with 1024 units each and an output layers with 16348 output units) maps to this architecture in a model parallelism fashion where an individual PE processes 4 units of each hidden layer and 64 units of the output layer. The PE reads and writes from DiRAM4 complying with instructions as well as memory allocations from the host. A single PE has a five-stage pipeline (*fetch*, *issue*, *read*, *execute*, *writeback*) and that can fetch four instructions (each instruction 32-bit wide) at every cycle. In addition, PE has five different lanes which allow it to execute five different types of instruction (direct memory access, load-store, vector, scalar, and special function) simultaneously. The flow control mechanism of PE allows the back-to-back issue of independent instructions in each lane. A single PE contains a direct memory access (DMA), a load-store unit (LSU), a vector functional unit (VFU), a scalar arithmetic unit (SAU), a special functional unit (SFU), and memories (SRAM and register files). The overall architecture of a PE is shown in Figure 5.10 and different key features are discussed in the subsequent subsections.

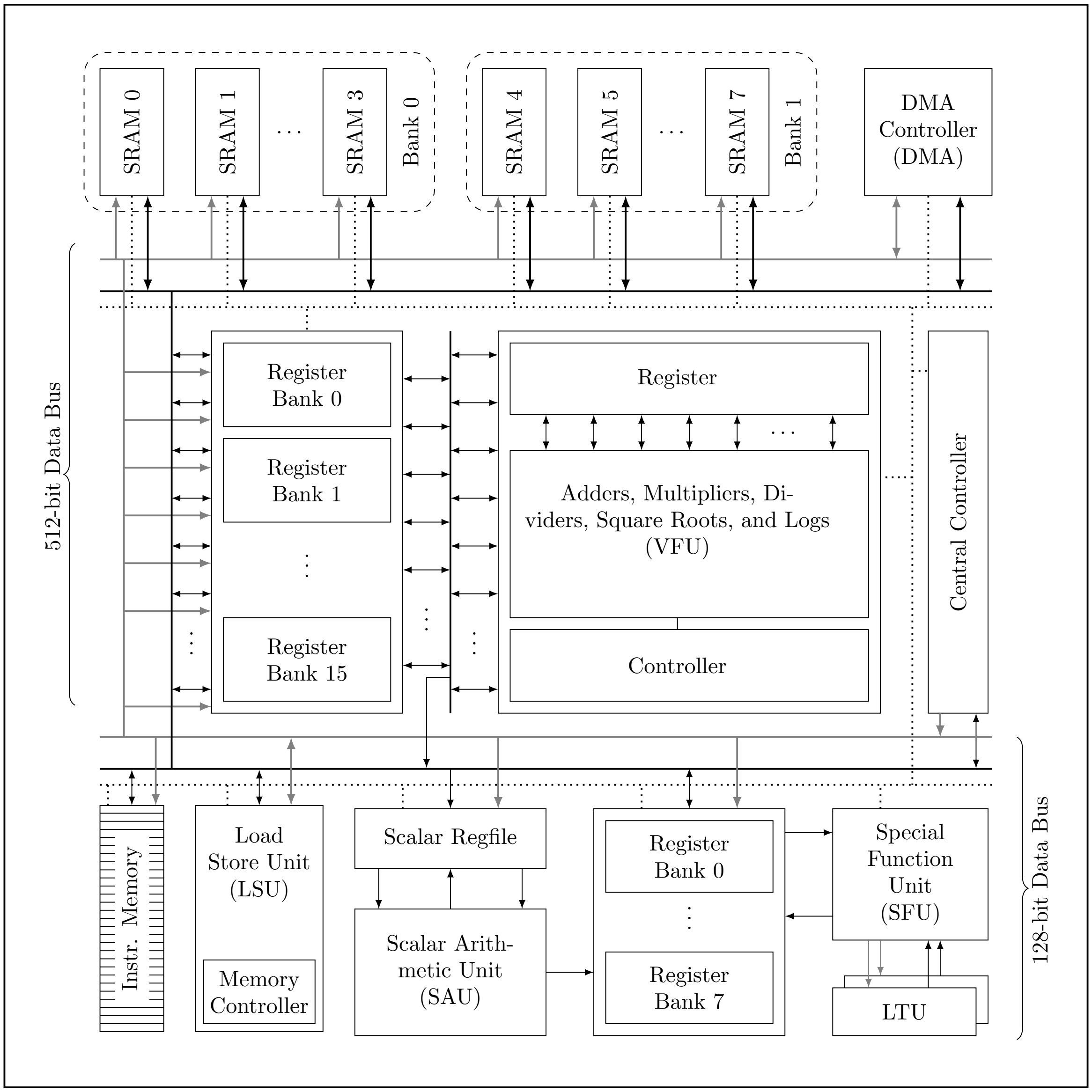

Figure 5.10: The overall architecture of a PE

Programmable DMA Controller

A fully programmable DMA is designed to transfer bulk data between DiRAM4 and PEs in different fashions (serial-in-parallel-out, parallel-in-serial-out, 16-bit IEEE 754 to 32-bit IEEE 754 conversion and vice versa, and also scatter-gather). A set of instructions has been implemented in DMA to access data efficiently from DiRAM4 for different computational steps of training and inference. Instructions of DMA communicate its status with instructions of other execution lanes through the central control unit and the dependent instructions are scheduled accordingly. The total on-chip SRAM (2 Mb) of a PE is decomposed into two banks (each 2048×512), where DMA can access one of the banks for read/write operation from/to the DiRAM4 via crossbar network. The DMA is bidirectional and can read/write 64-bits of data word from/to the DiRAM4 in every clock cycle. During mixed precision training, each parameter can be either 16-bit or 32-bit where 64-bit data between DMA and DiRAM4 can have either 4 or 2 parameters. In addition, the interface between DMA from/to SRAM bank is 512-bit wide which requires DMA to have data gathering mechanism before it is written to the SRAM banks. A DMA operation to read/write data from/to DiRAM4 is summarized below.

- The configuration instructions program the DMA controller for different addresses (read and write addresses) as well as burst length of the transfer.
- The configuration instruction will also set SRAM bank flag to busy so that instructions dependent on data of that SRAM bank will be waiting until the DMA operation is finished.
- The channel setup instructions configure the read and write channel (i.e. source and destination) to transfer data between SRAM bank and DiRAM4.
- The start instructions configure the mode (data conversion, scatter-gather, etc.) of operation and transfer data through a FIFO between the read and write channel. The central control unit is notified and the SRAM bank is freed at the end of the DMA operation. As a result, PE starts executing instructions waiting for data in that SRAM bank. The overall hardware architecture of the DMA with various operation modes is shown in Figure 5.11.

The DMA controller has data path, control unit, crossbar, FIFO (512-bit wide and depth of 8), six serial-in-parallel-out (SIPO), and three parallel-in-serial-out (PISO) with different widths and depths. Also, DMA has 16 internal registers (each 16-bit wide) to program it for different channel and data (addresses and sizes of the transaction) configurations. The write operation (DiRAM4 to SRAM) reads 64-bit data word from DiRAM4, stores to serial-in-parallel-out until serial-in-parallel-out is full (512-bit), transfers 512-bit data to the FIFO, and then transfers the data from FIFO to the SRAM bank using the SRAM controller. Similarly, the general read operation reads 512-bit word of SRAM and stores the data to the parallel-in-serial-out through

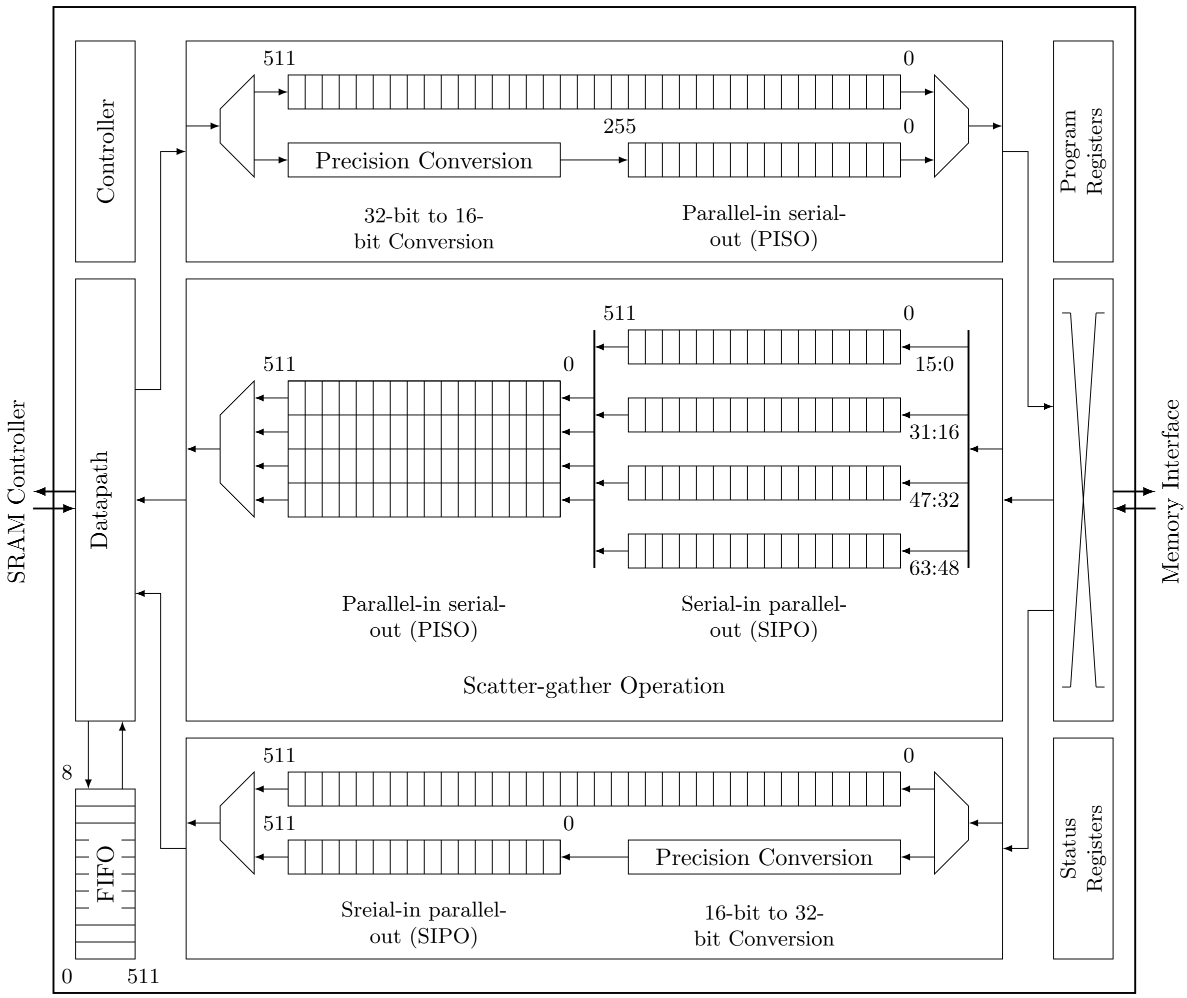

Figure 5.11: The architecture of programmable DMA controller

Table 5.3: The instruction format of DMA lane

31–29	28–25	24	23–22	21–20	19–16	15–0
DMA	Opcode	Unused	Source	Destination	Register Number	Register Value

Table 5.4: The address of DMA lane

Name	Address
DMA	001

the FIFO. The data of PISO is read serially and written as 64-bit data word to the DiRAM4 bank.

Furthermore, DMA uses precision conversion (half/single to single/half) mechanism (required for the mixed precision training) during data transfer between DiRAM4 and SRAM bank through buffers. It can read 64-bit data (4 parameters in half-precision) from DiRAM4 and convert the data to 128-bit data (4 parameters in single-precision) before writing to the SRAM bank. Similarly, it can read 32-bit parameters from the SRAM bank and write as 16-bit parameters to the DiRAM4. In the scatter-gather operation, the 64-bit data from DiRAM4 is scattered into four 16-bit words and stored (i.e. gather) into four different serial-in-parallel-out (SIPO) buffers before written (when the four SIPOs are full) to the FIFO. The data of FIFO is then written to the SRAM bank at four different memory locations based on the program registers of DMA. The scatter-gather mechanism is usually used during backward propagation

Table 5.5: The available opcodes of DMA instruction

Opcode	Description
0000	No operation
0001	Program DMA register
0010	Request DMA channel
0011	Start DMA in normal mode
0100	Start DMA in scatter-gather mode
0101	Release SRAM bank
0110	Start DMA in precision conversion mode

Table 5.6: The source and destination address

Name	Address
DiRAM	01
SRAM Bank 0	10
SRAM Bank 1	11

Table 5.7: The register mapping of DMA instruction

Register	Register Name	Description
Register 0	Read Start Address	Start read address of DiRAM4 or SRAM Bank
Register 1	Read Word Total	Total number of words to be read
Register 2	Read Word Count	Counter for number of reads
Register 3	Write Address Start	Start write address of DiRAM4 or SRAM Bank
Register 4	Write Word Total	Total number of words to be write
Register 5	Write Word Count	Counter for number of writes
Register 6	Read Address Increment	Increment read address after every read
Register 7	Write Address Increment	Increment write address after every read
Register 8	Write Address 0	SRAM write address 0 during scatter-gather
Register 9	Write Address 1	SRAM write address 1 during scatter-gather
Register 10	Write Address 2	SRAM write address 2 during scatter-gather
Register 11	Write Address 3	SRAM write address 3 during scatter-gather
Register 12	Not Used	Not Used
Register 13	Not Used	Not Used
Register 14	Not Used	Not Used
Register 15	Not Used	Not Used

to enhance the utilization of DiRAM4 memory bandwidth. The concept of a scatter-gather mechanism of this architecture is shown in Figure 5.12.

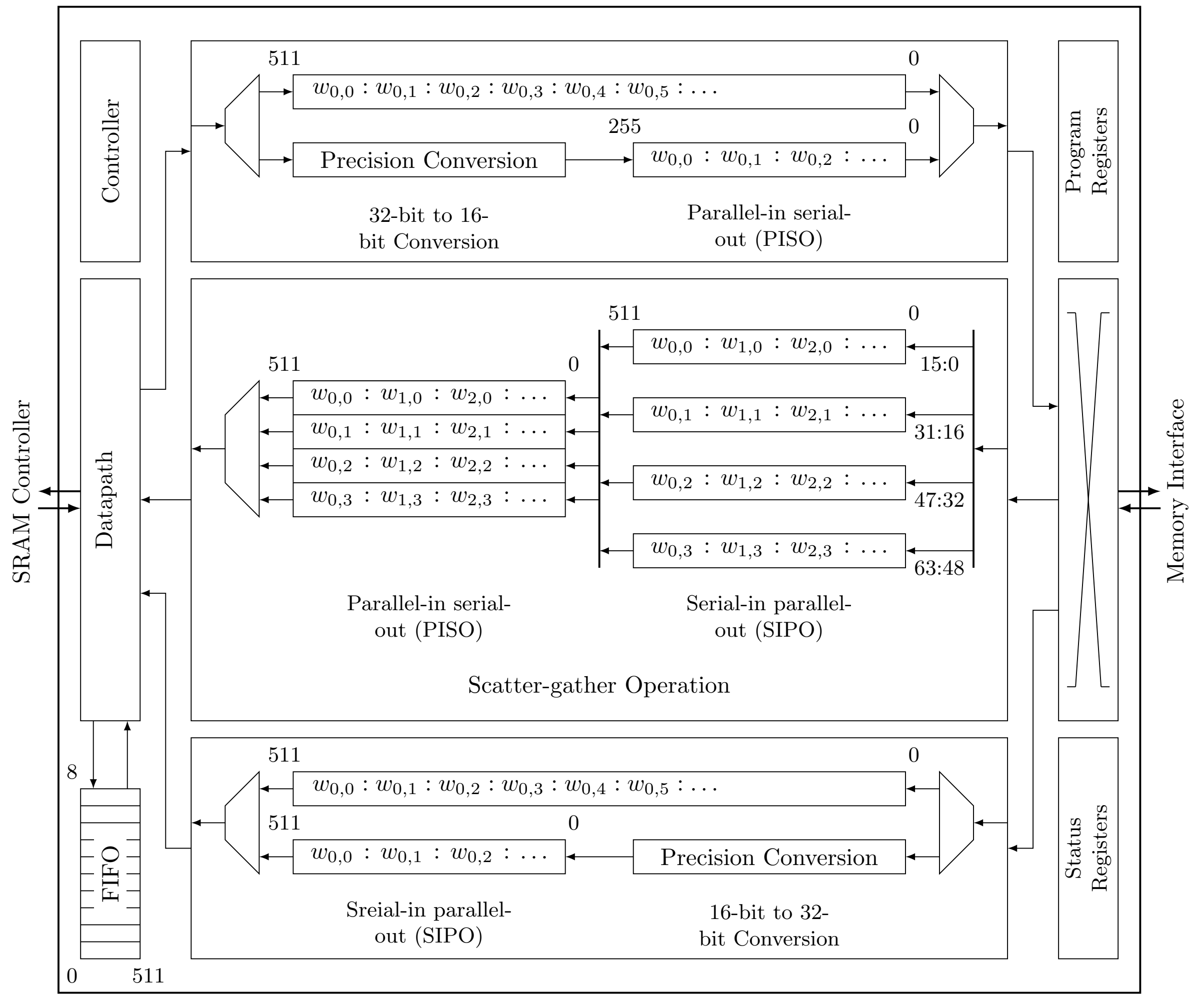

Figure 5.12: The dataflow of DMA operations

At the end of DMA transfer, there is a special instruction to release the SRAM bank and inform the central controller that the data of SRAM is ready to execute other dependent instructions. The 32-bit instruction format of the DMA and the description of its operands are given in Table 5.3 to Table 5.7.

On-chip Memory and Data Movement

There are three different register files (two vector and one scalar) with multiple ports and eight small on-chip SRAM (each 2048×128) available in a single PE. The eight SRAM banks group into two SRAM banks (four small SRAM banks in each group) with a data bus width of 512-bit; the address bus of SRAM banks remains the same throughout. The rationale for using two SRAM banks is to perform computation on data of one SRAM bank and prefetch data for next set of computations into another SRAM bank simultaneously.

The PE accommodates one multi-ported vector register file (*vreg*) size of 64 Kb and one multi-ported hybrid register file (*hreg*) of size 2 Kb. In addition, it has a multi-ported scalar register file (*sreg*) size of 4 Kb. The multiple register files with multiple ports are designed to support instruction-level parallelism within five lanes of this architecture. For instance, the *vreg* register file has three read ports and two write ports. The first read and write port are dedicated to the load-store unit and support continuous 512-bit data streaming from SRAM to the *vreg*. The other two read ports are dedicated to the vector function unit (VFU) for two operands of vector operations. In addition, the write port is assigned to write vector outputs (from VFU) to the *vreg*. The scalar register file (*sreg*) has two read and one write ports dedicated to the scalar arithmetic logic unit (SAU) and another write port is assigned to write the scalar output from VFU to the *sreg*. There is a scalar-to-vector converter associated with SAU, where it reads scalar data from the *sreg* and outputs a vector. In addition, the *sreg* has one read port that can read 128-bit vector data from the *sreg* and load to one of the SRAM banks. The *hreg* register file has three read ports and three write ports in which one read and one write port is dedicated to the load-store unit for 128-bit data transfer between *hreg* and SRAM. The next two read and one write ports are assigned to provide operands of the vector operation in the special function unit (SFU) and write output back to the *hreg*. The last write port is used to write vector output from the scalar-to-vector converter.

The load-store unit (LSU) of this architecture supports different load-store instructions and performs data movements in different fashions among different on-chip memories (SRAM, *vreg*, *hreg*, and *sreg*). The LSU can load/store 512-bit data from/to SRAM to/from *vreg* through the 512-bit data bus. It can also load/store 128-bit data from/to SRAM to/from *hreg*. A special store instruction can read 128-bit data from the *sreg* to one of two SRAM banks. In addition to the vector load/store, LSU can read 16-bit data word from SRAM and fan it out to 512-bit or 128-bit before loading it to the *vreg* and *hreg*, respectively. This type of load (scalar read followed by vector load) is necessary to load different hyperparameters (learning rate, momentum) of the network before performing scalar-vector operations. Moreover, the mixed precision training required to scale the loss function, which is also a scalar-vector multiplication. The details of 32-bit LSU instructions are listed in Table 5.8, Table 5.9, and Table 5.10.

Table 5.8: The instruction format of LSU lane

31–29	28–25	24–0
LSU	Opcode	Source and Destination Addresses

Table 5.9: The address of LSU lane

Name	Address
LSU	101

Table 5.10: The available opcodes of LSU instruction

Opcode	Description
0000	No operation
0001	Load SRAM to vreg
0010	Store vreg to SRAM
0011	Load SRAM to hreg
0100	Store hreg to SRAM
0101	Load SRAM to vreg (Data fanout)
0110	Load SRAM to hreg (Data fanout)
0111	Store sreg to SRAM

Table 5.11: The instruction format of VFU lane

31–29	28–25	24	23–16	15–8	7–0
VFU	Opcode	Unused	Source 0	Source 1	Destination

Table 5.12: The address of VFU lane

Name	Address
VFU	100

Vector Function Unit (VFU)

There is a VFU in each PE, where it has sixty-four (64) IEEE 754 compliant half-precision multiples, sixty-eight (68) IEEE 754 compliant single-precision adders, sixteen (16) single-precision multipliers, sixteen (16) single-precision dividers, sixteen (16) single-precision square roots, and sixteen (16) half-precision logarithms hardware. A Wallace tree multiplier using 3:2 compression technique was used during designing the 16-bit and 32-bit floating-point multiplier. A very small lookup-table has been used during designing the single-precision divider and square root hardware [79, 80]. The ICSILog algorithm with small lookup-table has been used to implement the half-precision logarithm hardware [81]. The multiplier, adder, divider, square root and logarithm hardware of this design are fully pipelined and throughput in every clock cycle. The precision of adder is higher to support the mixed precision training as per [30]. The single-precision fully-pipelined multipliers, dividers, and square root hardware are used to support the optimization process (using GD or Adam) during mixed precision training. The logarithm hardware is mainly used to calculate the loss ($Logsoftmax$ operation) during forward propagation. The entire training flow in half-precision usually suffers the accuracy loss because of the small gradient values, which is not possible to capture using 16-bit precision. However, increasing the precision of adder and performing optimization steps of training in single-precision (mixed precision training) can maintain training accuracy as good as single-precision training [30].

The operands are continuously streamed from the *vreg* and made available to the VFU during *read* stage of the pipeline. Based on different operation codes during the execute stage, the output of VFU can be scalar or vector and written as output to the *sreg* or *vreg* register file, respectively. A single instruction of VFU performs on vector operands of size 64 with a

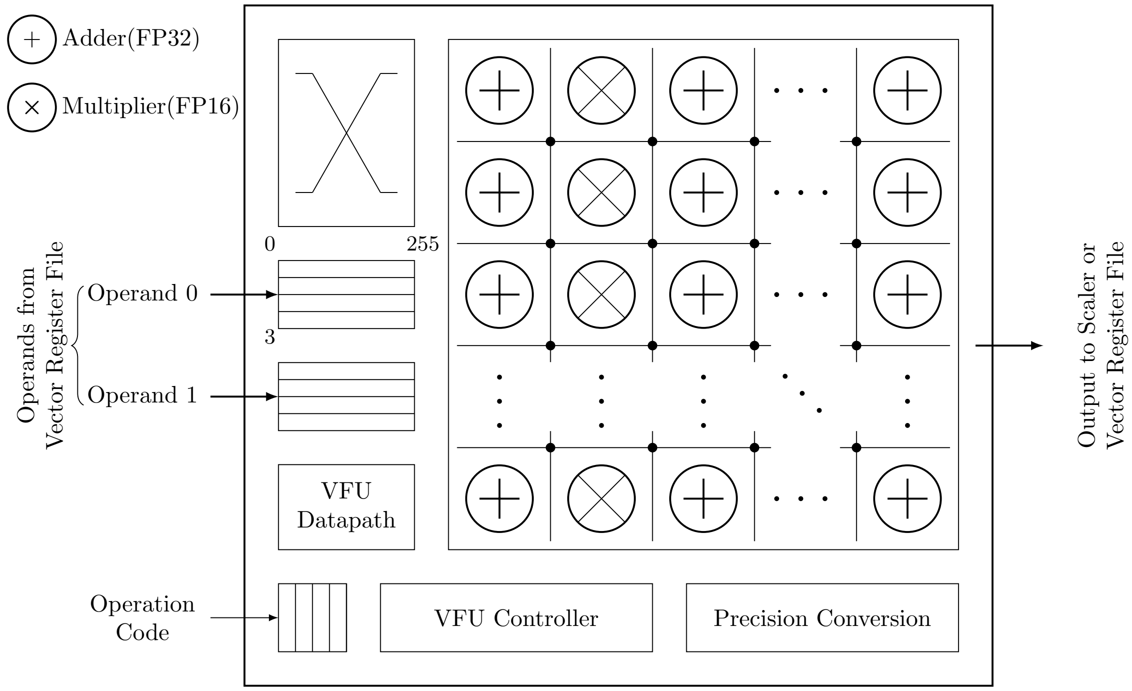

Figure 5.13: The architecture of Vector Functional Unit (VFU)

Table 5.13: The available opcodes of VFU instruction

Opcode	Description
0000	No operation
0001	Multiply-accumulate operation
0010	Element-wise multiplication
0011	Element-wise addition in half-precision
0100	Reduction of a vector
0101	Differentiation of Tanh
0110	Substract a vector from one
0111	Element-wise addition in single-precision
1000	Element-wise multiplication in single-precision
1001	Element-wise division in single-precision
1010	Square root in single-precision
1011	Log in half-precision

required precision conversion. For example, the MAC operation code (0001) mainly performs the dot product of 64 elements in half-precision multipliers and does the necessary conversion (half-precision to single-precision) before performing the reduction in single-precision adders. A set of special instructions have also been added to support the loss scaling and weight update steps in single-precision required for the mixed precision training. The VFU has a 3-bit address and 4-bit operation code to support twelve different operations for various computation steps of training and inference. The 4-bit operation code of VFU offers the possibility of future extension for other vector operations. Figure 5.13 shows the underlying architecture of the VFU. The instruction format and details of its operand are listed in Table 5.11, Table 5.12, and Table 5.13.

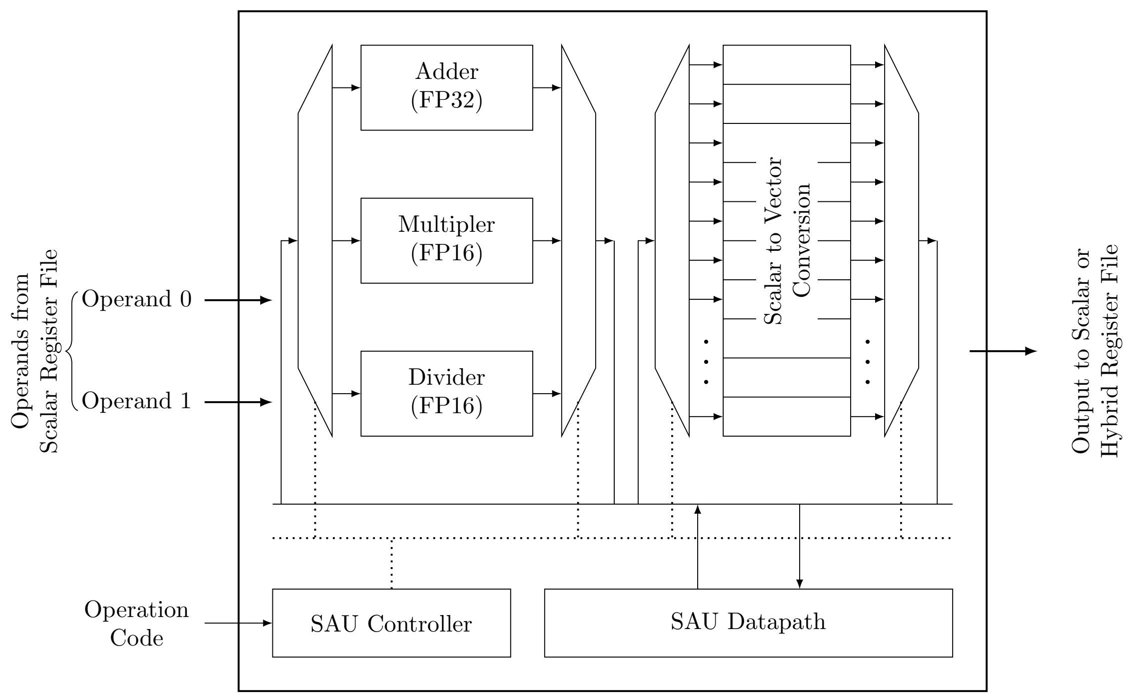

Figure 5.14: The architecture of Scalar Arithmetic Unit (SAU)

Scalar Arithmetic Unit (SAU)

A single Scalar Arithmetic Unit (SAU) in each PE performs different scalar operations as well as scalar-to-vector conversion and writes the vector to the *hreg* register file. The SAU consists of a single-precision adder, a half-precision multiplier, a half-precision multiplier-based divider, and a buffer. The scalar outputs from VFU are usually passed to the *sreg* for different scalar

Table 5.14: The instruction format of SAU lane

31–29	28–25	24	23–16	15–8	7–0
SAU	Opcode	Unused	Source 0	Source 1	Destination

Table 5.15: The address of SAU lane

Name	Address
SAU	011

operations and/or generates a vector output of length 4 (64-bit) and writes that vector to the *hreg* register file. The SAU is also used to compute different scaling parameters (scaling factor or shanking factor) during the backward propagation of training. For example, the scaling factor is defined based on the scalar operation (division) between the maximum normalized gradient (hyperparameter) and the current normalized gradient to scale the parameter gradients of the network before the weight update rule. The instruction format and currently supported scalar instructions in SAU are listed in Table 5.14, Table 5.15, and Table 5.16. The SAU also open to add new instructions (4-bit operation codes) for future extension.

Table 5.16: The available opcodes of SAU instruction

Opcode	Description
0000	No operation
0001	Addition
0010	Division
0011	Reciprocal
0100	Read sreg and load to the buffer
0101	Read sreg and fanout to the buffer
0110	Transfer buffer content to hreg

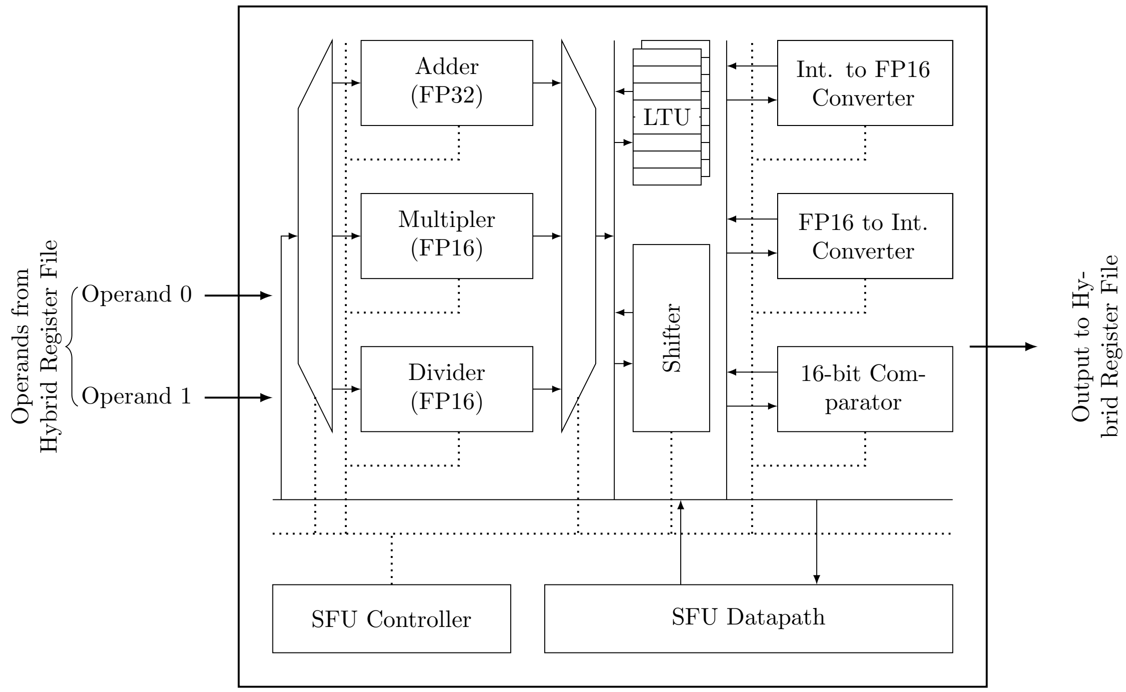

Figure 5.15: The architecture of Special Function Unit (SFU)

Special Function Unit (SFU)

There are four special function units (SFUs) in a PE to support computation of various non-linear activation functions as well as other vector operations (addition/subtraction, multiplication). The input vector size of SFU is 4 (unlike VFU, where its vector size is 64) and it reads/writes operands/output from/to the *hreg* register file. The frequently used non-linear activation functions in DNNs include the parameterized sigmoid, exponential, rectifier linear unit (ReLU) and hyperbolic tangent. The accuracy and efficiency of the non-linear functions have a significant role in the hardware performance as well as the accuracy of learning and inference for an LSTM network. The fixed-point implementations (based on PieceWise Linear (PWL), CORDIC and Taylor series expansion) are the most common approaches for implementing the non-linear functions, which always suffer small to medium accuracy loses for different input ranges. To maintain computation in IEEE 754 standard and efficiency of the algorithm, a combination of the look-up table (LUT) and the numerical method is implemented in hardware to improve hardware performance and maintain the learning/inference accuracy [82]. The SFU also has its a 3-bit address and seven instructions to perform different operations. A single instruction is assigned against each non-linear function to accelerate the non-linear function and at the same time reduce the number of instructions. For example, the opcode 2 (0010) is dedicated to performing the sigmoid operation. A single SFU hardware has one single-precision adder,

Table 5.17: The instruction format of SFU lane

31–29	28–25	24	23–16	15–8	7–0
SFU	Opcode	Unused	Source 0	Source 1	Destination

Table 5.18: The address of SFU lane

Name	Address
SFU	010

one half-precision multiplier, a multiplier-based half-precision divider, and two very small (each 32×32) LUTs. The hardware implementation of exponential function only in single-precision was done in [83]. The hardware architecture of an SFU is shown in Figure 5.15. The supporting instructions and details are listed in Table 5.17, Table 5.18, and Table 5.19.

Central Controller

Multi-ported register files are designed in the central controller to monitor the status of hardware resources and the data dependencies due to eliminating the structural and data hazards accordingly. During the *issue* stage of pipeline, the five instructions for five different lanes go to the central controller and simultaneously check the register file (multi-ported register file) to ensure the execution unit and/or destination register (applicable only for VFU, SAU, and

Table 5.19: The available opcodes of SFU instruction

Opcode	Description
0000	No operation
0001	exponential
0010	Sigmoid
0011	Tangent hyperbolic
0100	Substract vector from one
0101	Addition
0110	Multiplication

Table 5.20: The status of functional unit

Functional Unit	Opcode	Status
DMA	0100	Doing scatter-gather operation
LSU	0001	Loading data from SRAM to vreg
VFU	0001	Performing Mac operation
SAU	0010	Doing division operation
SFU	0011	Doing tanh function

Table 5.21: The status of SRAM bank

SRAM	Functional Unit	Status
Bank 0	DMA	DMA performing scatter-gather on SRAM bank 0
Bank 1	LSU	LSU loading data from Bank 1 to vreg

SFU) of these 5 instructions are available (avoid structural and write-after-write (WAW) hazard). The successful *issue* of one/more instructions also updates the register files of the central controller. During the *read* stage, five instructions check the status of their operands simultaneously from the register file database and eliminate the read-after-write (RAW) hazard. As soon as the execution unit finishes the execution of an instruction, the central control updates its database (free the function unit) and inform waiting instructions that the execution unit is ready to execute the next instruction. In the *writeback* stage, instructions update the status of its destination registers in the central control unit. The central control unit then informs the instructions waiting for the same destination registers (avoid WAR hazard). The unique concept of the central controller is to design multi-ported register files in a way to support multiple reads and writes during *issue*, *read*, *execute*, *writeback* stages and serve the maximum degree of parallelism of the pipeline stages. A register-based central control named scoreboarding also used in the out-of-order processor, but a scoreboard mechanism can issue only one instruction at a time where this architecture can issue a maximum of five instructions at a time. Table 5.20, Table 5.21, and Table 5.22 represent an example of the register file content in the central controller.

Table 5.22: The status of register files

Name	Address	Status
vreg	0	Busy
vreg	1	Available
vreg	2	Available
vreg	3	Busy
⋮	⋮	⋮
hreg	0	Available
hreg	1	Busy
hreg	2	Busy
hreg	3	Available
⋮	⋮	⋮
sreg	0	Busy
sreg	1	Busy
sreg	2	Available
sreg	3	Available
⋮	⋮	⋮

5.3.4 Instructions

Each PE consists of five execution lanes (DMA, LSU, VFU, SFU, SAU) and each execution lane consists of five pipeline stages ($fetch$, *read*, *issue*, *execute*, and *writeback*). An address is assigned against each execution lane of a PE. There are also five groups of different instructions corresponding to these execution lanes, which were presented in the previous section. In addition, each execution lane supports "no operation" for the pipeline bubble for introducing delay inside the pipeline. The "no operation" is needed for C++ compiler due to insert a delay (usually 4 cycles delay to update register files of the central controller) in the execution lane for maintaining the register-based flow control among all lanes and avoid different hazards (structural and data hazard). For instance, the VFU execution lane is fully pipelined and supports eleven different datapaths for eleven different instructions. The central controller has four cycle delays in updating its register when issuing an instruction. There should be at least four cycles delay to eliminate the datapath conflict (i.e. structural hazard) if a different type of VFU instruction required to issue back-to-back. The compiler usually introduces four cycles delay between two different types of VFU instruction with the help of "no operation". However, issuing the same type of VFU instructions back-to-back does not require any delay (no datapath conflict) inside the pipeline. In addition, there is also a "synchronous" instruction in each execution lane to synchronize executions among different lanes. The "synchronous" instructions can be used especially to make sure finishing up the entire computation of a DNN layer and before starting to execute the instructions for the next layer.

5.3.5 Data Reuse

The off-chip memory access (DiRAM4) is always expensive from latency (22 ns delay for worst-case page open and 14 ns delay for worst-case access) and power point of view (page open and cache read/write energies are 100 pJ and 64 pJ, respectively, at 1 GHz). Due to speedup of the overall performance, the data reuse technique has been used at various steps of learning and inference for an LSTM network. The LSTM network offers very small data reusability. However, in batch processing, the multiple examples or samples are applied to the same weight parameters and the weight reusability is proportional to the number of samples or the batch sizes. These operations (matrix-matrix or matrix-vector multiplication) are performed using a block matrix where submatrices of weight and input matrix are kept inside PE and the other submatrices are continuously streamed from the DiRAM4 memory. The data reuse is done at various stages inside a PE. The on-chip SRAM holds the partial weight parameters and at the same time, *vreg* can hold a segment of those parameters during batch processing. In addition, the state of the LSTM unit, the error of current timestamp (usually used in the previous timestamp during backward propagation) and intermediate activations (activations of gates, cells, etc.) are stored in on-chip SRAM and used once or multiple times in subsequent computations.

5.3.6 Compiler

There could be several approaches to design compiler or generate instructions for an LSTM network. In this implementation, the data flow of an entire LSTM network has been analyzed during compile time and allocated memories for efficient memory management for both training and inference. The high-level C++ functions have been designed to generate the low-level VLIW instructions into a binary text file to perform DiRAM4 memory transaction at various forms, block matrix operations, data movement, and also various special function operations. The current version of C++ implantations is designed to target the LSTM and MLP networks. However, the implementation can be extended for other DNNs in future. The sample C++ functions and the generated instructions for five different functional units (lanes) are explained in Table 5.23.

The first high-level function (DLDR) generates a sequence of instructions to pull data out from the DiRAM4 and load to the SRAM. The configure instructions (first nine instructions) program the DMA registers and channel assignment for this transfer. Then, the start instruction transfers (16×64) 128 bytes of DiRAM4 data (between address 0 and 15) to the SRAM address

Table 5.23: The example of high level functions and corresponding instructions

Functional Unit	High-Level Function	Generated Instruction	Description
DMA	DLDR(0, 1, 0, 15, 1, 1)	22600000	Config: Register 0 $\Leftarrow$ 0
		22610010	Config: Register 1 $\Leftarrow$ 15
		22620000	Config: Register 2 $\Leftarrow$ 0
		22630000	Config: Register 3 $\Leftarrow$ 0
		22640002	Config: Register 4 $\Leftarrow$ 1
		22650000	Config: Register 5 $\Leftarrow$ 0
		22660001	Config: Register 6 $\Leftarrow$ 1
		22670001	Config: Register 7 $\Leftarrow$ 1
		24600006	Config: Channel request
		26600000	Start DMA operation
LSU	VLDR(0, 2)	a2000002	Load: vreg[2] $\Leftarrow$ SRAM[0]
VFU	VMAC(0, 1, 0)	82000100	sreg[0] $\Leftarrow$ Mac(vreg[0], vreg[1])
SAU	SADD(0, 1, 2)	62000102	sreg[2] $\Leftarrow$ sreg[0] + sreg[1]
SFU	VTAN(0, 1)	46000001	hreg[1] $\Leftarrow$ Tanh(hreg[0])

0 and 1 (SRAM bus is 512-bit wide). The VLDR function generates instructions to load 512-bit data from SRAM address 0 to the *vreg* address 2. The VMAC function generates an instruction to perform MAC operations on data located at address 0 and 1 of the *vreg* and the write scalar output to the *sreg* at 0 address. Similarly, the SADD and VTAN functions generate an instruction to perform the scalar addition and tangent hyperbolic, respectively. Based on these instructions, the next level high-level functions were also written in the compiler to generate VLIW instructions for forward propagation, backward propagation, and weight update stages of the target LSTM network for both batch and non-batch processing.

5.4 Experimental Setup

This section discusses the experimental setup of this implementation as well as software implementations in other hardware solution. The performance estimation processes of this work and comparable hardware platform are also discussed.

The Verilog RTL was used to model and implement the core of a single PE and the memory generator was used to generate the on-chip memory (SRAM). The behavioral model of a DiRAM4 was also implemented in Verilog and used as external 1 Gb DiRAM4 bank. As per

Table 5.24: The hardware specification of Tesla P100 GPU

Name	Tesla P100 GPU
Multiprocessors	56
Core Count	3584
Die Area	610 mm^2
Technology	16 nm
Max Clock Rate	1480 MHz
Memory Size	16 GB
L2 Cache Size	4096 KB
Memory Bus Type	4096-bit HBM2
Memory BW	732 GB/s
Bus Interface	PCIe Gen3
Board Power	300 W

datasheet, the DiRAM4 bank has 32-bit Dual Single Data Rate (DSDR) I/O data bus run at 1 GHz while the peak data bandwidth of a single PE is 32 Gbps (64-bit I/O data bus run at 500 MHz). The storage of parameters, WMT16 English-German dataset, and dataflow of Google's Neural Machine Translation System was analyzed during compile time and assigned DiRAM4 memory accordingly [84, 7]. The Google's Neural Machine Translation System has three different networks: 1) Encoder LSTM network; 2) Attention network; 3) Decoder LSTM network. The encoder network has one bi-directional LSTM layer with 1024 nodes, one LSTM layer with 1024 nodes but 2048 input feature vectors and two LSTM layers with 1024 nodes. The attention network has one LSTM layer with 1024 nodes and two FC layers with 1024 input feature vectors. Finally, the decoder network has three LSTM layers with 1024 nodes but 2048 input feature vectors. There is also a classifier and Logsoftmax layer with total vocabulary size equal to 32320. The popular regularization mechanism (Dropout) also enabled before each LSTM layer in this benchmark system. The high-level C++ function was called to generate instructions for processing a segment of the target network and dump binary instructions into a text file. The instructions were loaded to the instruction buffer. However, the original concept of this implementation is to store instructions to the DiRAM4, and DMA instructions will be used to fill the instruction buffer.

The ModelSim simulation was performed to calculate the number of cycles required to process a segment of the benchmark network. The C++ environments have been developed for functional verifications of high-level function as well as the benchmark network. The spreadsheet calculation was performed to estimate the overall computation time. The synthesis and place-

Table 5.25: The key hardware results of a PE

Name	Processing Element
Technology	28 nm
Clock Frequency	500 MHz
Core Voltage	0.8 V
Total Area	4.64 mm^2
Total Cell Area	4.17 mm^2
On-chip SRAM Area	0.47 mm^2
On-chip Register	46 KB
On-chip SRAM	256 KB
SRAM Area/Total Area	11 %
Core Power	479.29 mW

and-route of RTL was done using Synopsys DesignWare and Synopsys ICC, respectively, in 28nm standard cell library at 500 MHz. The overall area of a PE with on-chip SRAM was calculated from the core area of the routed netlist from Synopsys ICC. The total area of this architecture was calculated based on the area of a single PE. The power of a PE was calculated based on the routed netlist after place-and-route of that PE using Synopsys ICC followed by switching activity (VCD) processed by Synopsys Primetime. In addition, the access pattern of DiRAM4 has been captured in ModelSim simulation and power calculated based on access (page open/close, cache read/write, page refresh, etc.) energy from the Tezzaron DiRAM4 datasheet. The spreadsheet calculation was performed to calculate the overall performance for comparison with Tesla P100 GPU (hardware specification of Tesla P100 GPU is given in Table 5.24). The 16nm scaling factor (ITRS) was used to estimate the area and power results of this accelerator in 16nm technology [70]. To profile a Tesla P100 GPU with HBM2, the baseline network (also implemented in mixed precision) as well as profiling script was written in PyTorch (deep learning framework) using deep learning APIs (cunn, cuDNN, cutorch) for CUDA backend acceleration. The NVIDIA GPU profiler was invoked to measure the energy and the compute time of GPU. The die area was taken from the datasheet of respective Tesla P100 GPU [85].

5.5 Results

This section summarizes the area, power, and timing analysis of the proposed accelerator with the performance of Tesla P100 GPU with HBM2 high-speed memory [85]. The performance comparison of this accelerator is evaluated using Google's Neural Machine Translation System

Table 5.26: The area comparison

Name	Technology	Tesla P100 GPU	This Architecture	Inprovement
Die Area	16nm	610 mm^2	407 mm^2	1.5×

Table 5.27: The speed and power comparison for batch processing

Name	Tesla P100 GPU		This Architecture		Improvement	
Type of Metrics	Forward Pass	Backward Pass	Forward Pass	Backward Pass	Forward Pass	Backward Pass
Processing Time (ms)	92.16	169.73	63.14	107.20	1.46×	1.58×
Power (W)	205.51	175.27	37.56	49.25	5.47×	3.56×
Memory BW (GB/s)	732	732	1024	1024	1.40×	1.40×

Notes. the power and bandwdith of GPU are from CUDA profiler and standard datasheet

for automated language translation task from English to German (i.e. sequence-to-sequence processing task). The dataset (WMT16 English-German dataset) has a total of 32320 unique words. For benchmarking purpose, the network was unrolled for 64 time steps (64 words per sentence) and evaluated training and inference performance for two different mini-batch sizes (1 and 32). The online or non-batch processing means when minibatch size is 1. The batch processing of this architecture was done for a mini-batch of size 32.

The key hardware results of a single PE is shown in Table 5.25. The PE is synthesized at 500 MHz in 28nm commercial technology followed by place-and-route for calculating the area and average power. The overall core area (after place-and-route) of a PE is 4.64 mm^2, even though the synthesis area of a PE is 2.38 mm^2. The three on-chip register files (64 Kb for *vreg*, 2 Kb for *hreg*, and 8 Kb for *sreg*), and sequential cells (176 Kb) adds up total 46 KB on-chip register of a PE. The size of on-chip SRAM is 256 KB (11% of the total area), which is usually useful for improving data locality during batch processing. Based on the area of a single PE and considering 5% of the area for the crossbar network and other logic in the PE layer, the overall area of this architecture was calculated. The ITRS scaling factor was used to estimate the die area of this accelerator in 16nm technology [70]. The overall area comparison against a P100 GPU's die area is shown in Table 5.26. The area benefit of this architecture is 1.5×. The die area of this architecture is 2× than the die area of DiRAM4 3D memory system.

The processing time, power requirement, and available memory bandwidth for batch and

Table 5.28: The speed and power comparison for online processing

Name	Tesla P100 GPU		This Architecture		Improvement	
Type of Metrics	Forward Pass	Backward Pass	Forward Pass	Backward Pass	Forward Pass	Backward Pass
Processing Time (ms)	28.12	45.50	14.23	31.34	1.98×	1.45×
Power (W)	109.20	100.17	33.43	33.74	3.27×	2.97×
Memory BW (GB/s)	732	732	1024	1024	1.40×	1.40×

Notes. the power and bandwdith of GPU are from CUDA profiler and standard datasheet

Table 5.29: The speed and power comparison for optimization using Adam

Name	Tesla P100 GPU	This Architecture	Improvement
Type of Metrics	Optimization and Weights Update	Optimization and Weights Update	Optimization and Weights Update
Processing Time (ms)	27.70	4.93	5.62×
Power (W)	96.57	13.31	7.26×
Memory BW (GB/s)	732	1024	1.40×

Notes. the power and bandwdith of GPU are from CUDA profiler and standard datasheet

Table 5.30: The operations per second comparison for batch processing

Name	Tesla P100 GPU		This Architecture		
Type of Operation	Forward Pass (GOPS)	Backward Pass (GOPS)	Forward Pass (GOPS)	Backward Pass (GOPS)	Overall Improvement ×
Multiply & accumulate	2965	2985	4328	4726	1.5×
Sigmoid, Tanh etc.	134	118	196	188	1.6×

Table 5.31: The operations per second comparison for online processing

Name	Tesla P100 GPU		This Architecture		
Type of Operation	Forward Pass (GOPS)	Backward Pass (GOPS)	Forward Pass (GOPS)	Backward Pass (GOPS)	Overall Improvement ×
Multiply & accumulate	304	348	600	505	2.0×
Sigmoid, Tanh etc.	14	14	27	20	1.5×

online processing are shown in Table 5.27, Table 5.28, and Table 5.29, respectively. The batch inference performance refers to the resource requirements due to process 32 samples unrolled up to 64 time steps for the forward propagation only. The batch training refers to the resource requirements for processing a forward propagation, a backward propagation for 32 samples unrolled up to 64 time steps, and a weight update step using Adam, a state-of-the-art optimizer. The GPU usually performs well beacsue of high-speed HBM2 memory and memory bandwidth and highly optimized software for DNNs. This architecture still speeds up the processing 1.5× – 5.6× for different steps of inference and training. The reduced address bus and higher data bus (512-bit) of this architecture, and the increased on-chip SRAM offer approximately 2.97× – 7.26× power benefit. Unlike batch processing, GPU usually struggles for data (i.e. memory bandwidth) during online processing, especially during the forward propagation of online processing and the optimization. The high bandwidth (read bandwidth 4 Tb/s and write bandwidth 4 Tb/s) DiRAM4 has helped this architecture to achieve approximately 5.62× speedup during online processing (optimization using Adam). The memory bandwidth (read bandwidth) requirement for online processing of this architecture is approximately 3.6 Tb/s.

The hotspot operation of different DNNs (especially MLPs, RNNs, and CNNs) are matrix-

Table 5.32: The number of instructions for batch processing

	Forward Propagation		Backward Propagation	
Instruction Type	Total No. of Instructions ($\times 10^3$)	Total No. of NOP Instructions ($\times 10^3$)	Total No. of Instructions ($\times 10^3$)	Total No. of NOP Instructions ($\times 10^3$)
DMA	1393	28806	790	52271
LSU	23743	6465	44843	8284
VFU	17841	12366	35554	17573
SAU	4913	25295	280	52847
SFU	178	30019	0	53127
Total Instr	48069	102949	81467	184102
Instr BW (GB/s)	2.84	6.07	2.83	6.40

matrix and matrix-vector multiplication which boils down to the multiply and accumulate operations on data in various forms. Table 5.30 and Table 5.31 shows the GFLOPS of multiply and accumulate operations for batch and non-batch processing, respectively. The MLP and LSTM are data bound rather than computations and this architecture achieved approximately 2× higher GFLOPS during online processing. The high bandwidth DiRAM4 of this architecture has helped to achieve this higher GFOPS. In addition to the multiply and accumulate, the operations involve with non-linear activations (Sigmoid, tanh, exponential, etc.) and other operations (especially element-wise operations) are also required but not significant. This architecture has also achieved 1.6× higher operations per second for different non-linear activations and other operations.

The proposed architecture has five execution lanes, and a PE can fetch 128-bit wide instructions in each cycle. During compile time, the instructions (128-bit for a VLIW instruction word) were generated in which each lane has 32-bit wide instructions. The number of instructions of this architecture was minimized to define single instruction for different high-level operations. Table 5.32 and Table 5.33 shows the number of instructions required to process one forward propagation and one backward propagation of the target LSTM network unrolled up to 64 time steps. The dedicated instructions for high-level operations (Sigmoid, Tanh, etc.) reduces the DMA and SFU instructions significantly compared to others. The amount of load-store, vector operations, and scaler arithmetic operations are almost similar. These three types of instructions are mostly related to the different matrix-matrix and matrix-vector operations. The instruction bandwidths of this architecture are 2.8 GB/s and 0.5 GB/s for batch and on-

Table 5.33: The number of instructions for online processing

	Forward Propagation		Backward Propagation	
Instruction Type	Total No. of Instructions ($\times 10^3$)	Total No. of NOP Instructions ($\times 10^3$)	Total No. of Instructions ($\times 10^3$)	Total No. of NOP Instructions ($\times 10^3$)
DMA	57	1515	83	4151
LSU	1053	519	3391	845
VFU	559	1013	899	3337
SAU	519	1053	14	4222
SFU	8	1565	2	4234
Total Instr	2195	5665	4389	16788
Instr BW (GB/s)	0.57	1.48	0.52	2.00

Table 5.34: The number of instructions for Adam optimizer

	Adam Optimization	
Instruction Type	Total No. of Instructions ($\times 10^3$)	Total No. of NOP Instructions ($\times 10^3$)
DMA	7	1444
LSU	1218	232
VFU	203	1247
SAU	0	1450
SFU	0	1450
Total Instr	1428	5823
Instr BW (GB/s)	1.08	4.40

line processing, respectively. The compiler of this architecture also inserted the "no operation" during generate instructions. The variable length VLIW instructions could potentially save the instruction bandwidth (1.5 GB/s to 6.5 GB/s for non-batch and batch processing) for "no operation", which remain to be examined as future work. The number of instructions and instruction bandwidth for Adam optimizer are also shown in Table 5.34. The entire optimization process of Adam mainly uses the LSU and VFU instructions.

5.6 Discussion

This section discusses two essential aspects (flexibility and scalability) that an accelerator should have for efficient deep learning applications. It also discusses flexibility of mapping LSTM into different ways in this accelerator.

5.6.1 Flexibility

There are different flexibilities that a hardware should have to process different LSTM networks. The first one is to support different model variant of RNNs. RNN has many variants and LSTM itself has at least eight different variants for speech processing, handwriting recognition, and music modeling [41]. This hardware supports instructions for a wide range of RNN variants. The most popular and complex variants of LSTM networks were analyzed for defining instructions to support both training and inference. The compiler is also written in a way to support high-level function to perform different operations with multiple low-level instructions. For example, the *norm* operation of a matrix mainly decomposes into two vector instructions (elementwise multiplication of the same matrix followed by a reduction operation) and a scalar operation. The second flexibility is to map networks different way into this hardware. The batch sizes, hyperparameters, and dimensions of an LSTM network are key parameters to choose right parallelism into hardware. Due to support data parallelism and model parallelism, the high density single DiRAM4 bank has been adapted in this work to store an entire network and the "synchronous" instruction to synchronous layer computation. The compiler or high-level C++ function also offers the ability to perform pipelining for a given LSTM network.

5.6.2 Scalability

In this work, a single 3D ASIP implementation was compared to a GPU. However, to accelerate the training of a DNN, multiple GPUs are often employed. In [86], it achieved 47× and 39× training speedup in a cluster of 128 GPUs for GoogleLeNet and Network-in-Network. The computation of training and inference of an LSTM network has linear scalability with hardware resources. As a result, the speed and scalability of LSTM network are not limited by the number of accelerators rather by the communication among these accelerators. The topologies of different global interconnect networks, communication bandwidth among multiple accelerators, and routing mechanism will be examined in the future.

5.7 Conclusion

In this work, a novel application-specific instruction set architecture has been proposed to accelerate training and inference of RNNs, MLPs, and potentially other DNNs. An emerging 3D memory system has adapted in this architecture to improve the off-chip memory bandwidth. The on-chip memory has been used to increase the data locality for batch processing and improve the off-chip power and bandwidth budget. In addition, instructions were designed carefully to reduce the number of instructions and the instruction bandwidth. Multi-ported register files have been designed and implemented to support the highest degree of instruction-level parallelisms. Instructions were added to support both half-precision and mixed precision training due to maintaining the accuracy comparable to single-precision. A small lookup table and numerical method were implemented in hardware to perform special function operations and improve the hardware performance. Dedicated instructions for special function operations were also incorporated in this work. The high-level C++ functions were written to generate blocks of instructions to perform forward propagation, backward propagation, and weight update stages. A prototype of this accelerator (PE) was implemented in 28nm commercial technology, and the area and power were also measured. Future work includes adding instructions after careful evaluation of other DNNs (CNN, RBM, etc.) and benchmark accordingly. The final chip tape-out also remains an important goal to be done in future work.

Chapter 6

Conclusion and Future Work

Deep learning using deep neural networks has revolutionized many aspects of modern society through a wide range of AI applications. Deep learning uses labeled data to learn different representations from the natural dataset. However, learning and inference of representations are computationally expensive and memory intensive in deep neural networks. Therefore, these applications are hard to implement in traditional hardware within a limited area and power budget. In contrast to deep learning, cortical learning algorithm learns representations from entirely unlabeled data and works closer to the way a human brain works. A deep neural network that models cortical learning algorithm in hardware is also computation and memory intensive, but computations are mostly binary and integer rather than floating-point in deep learning and the amount of computations are much higher than that of deep learning. To address these problems, a special custom hardware was presented for a cortical learning algorithm and an application-specific instruction set processor for deep learning to improve the performance of learning and inference.

The primary purpose of this custom hardware was to accelerate learning and inference of Sparsey, a cortical learning algorithm, in configurable custom hardware and improve its power/speed performance. The acceleration and performance improvement of custom hardware focused on three aspects: bit-level data processing from distributed on-chip SRAM for synaptic weights; a custom multiply-accumulate hardware; and fixed-point arithmetic for a special function and other floating-point operations. In addition, this work also focused on improving the performance of recurrent neural networks in application-specific instruction set processor from four aspects: an emerging 3D-stacked memory system to increase the off-chip storage and memory bandwidth; a set of short instructions for different complex and time-consuming special operations; a look-up table technique for special function operations; and Very Long Instruction Word (VLIW) instructions. The summary of contributions is discussed in the following section.

6.1 Summary of Contributions

In custom hardware of Sparsey, the highest memory bandwidth and power efficiency was achieved using an on-chip SRAM in each neuron for storing and retrieving synaptic weights during the iteration of learning and inference. With proper sizing and arrangement of on-chip memory and register file, the custom hardware has achieved 92× higher memory bandwidth than Tesla P100 GPU. The processing element of Chapter 4 has explained the details of on-chip memory organizations. The bit-level storage in distributed on-chip memory and processing at custom hardware were used to reduce the storage requirement and increase the bandwidth by 8×. The read and write of on-chip memory and fixed-point arithmetic for special function has made this custom hardware 353.12× more energy efficient than a GPU. The learning and inference steps of Sparsey were also analyzed to determine the suitable fixed-point formats and storage for computations. A custom multiply-accumulate hardware complying with the peak bandwidth of on-chip SRAM and the fixed-point arithmetic has achieved 25.24× acceleration on KTH action dataset. The custom multiply-accumulate hardware and fixed-point arithmetic also improved the die area by 1.43× than Tesla P100 GPU. The local interconnection topology was implemented while considering the routing and traffic pattern of Sparsey for further improvement of the performance. Finally, a scalable global interconnection network, *Fat tree*, has used to perform the scalability analysis of proposed accelerator for exhaustive inter-cluster communication. The computation and core processing time were usually scaled linearly with the hardware resource but not for inter-cluster communication.

The recurrent neural network has many variants and the most popular and successful variant, Long Short-Term Memory, itself has at least eight variants for three different applications. An application-specific instruction set processor was designed after careful evaluation of data flows and computations of popular RNNs to support as many variants as possible. A set of short instructions were designed to accelerate different complex, time-consuming, and special operations for training and inference. In addition, it also supports different data flow mapping of LSTM networks in this accelerator. The processing of LSTM networks are data bound instead of computation bound and an emerging multi-pored 3D memory system was used to improve the bandwidth. The training of LSTM networks is usually performed in batch mode and has reasonable data locality inside the processor. Thus on-chip SRAM banks were used in processing element to hold data for batch computation as well as intermediate storage. The on-chip SRAM has relaxed the bandwidth pressure and saved the access energy for off-chip memory access. Two banks of on-chip SRAM have also increased the computation and resource utilization by computing and prefetching data simultaneously. The data allocation of parameters and intermediate activations in DiRAM4 have been designed for both training and inference. The scatter-gather operation and an instruction associated with it have also been implemented in the DMA controller to improve the memory bandwidth for an irregular pattern of memory access during training. The 16-bit floating-point multiplier has been used to save area, storage, and memory bandwidth. The 32-bit floating-point adder has been used for accumulation to support state-of-the-art mixed precision training. The lookup table based 32-bit floating-point divider, multiplier, and square root hardware have been used for optimization (using Adam) during mixed precision training. The mixed precision computation has maintained the training accuracy at a level equivalent to single-precision training. The special instructions have also been designed and implemented to support the mixed precision training. A combination of the lookup table and numerical technique has been implemented in hardware for different special function operations which has improved the performance. The high-level short instructions have also been designed to be associated with each special function operation. A register-based central controller was designed to support multiple instructions (up to five issues and executes) simultaneously. In addition, the 128-bit wide VLIW instruction has accelerated the training and inference by executing multiple instructions concurrently (instruction-level parallelism). Finally, a high-level programming environment has been developed to generate Very Long Instruction Word (VLIW) instructions for training and inference using this accelerator. This ASIP accelerator has achieved 1.5×–5.6× speedup for training and inference compared to Tesla P100 GPU for different batch and non-batch processing. The accelerator has also yielded 1.5× less die area and approximately 4.3×–40.8× less energy.

6.2 Future Work

Sparsey is based on sparse distributed representation (SDR) and has very deterministic memory locality and memory access pattern. Also, the levels of local memory and memory bandwidth are the crucial elements required to achieve higher performance. The Processor-in-memory (PIM) and the custom version of 3D-stacked high bandwidth memory could potentially enhance the memory bandwidth. The latencies of global interconnection network limit the scalability after a certain point. The design of a high bandwidth scalable interconnection network and exploration of different communication algorithms for low latency inter-cluster communication would also be interesting topics for future research.

The ASIP accelerator offers instructions to cover the acceleration of popular recurrent neuron networks where the characteristics of computation are dense. However, the computational characteristics of CNNs are sparse and have very high data locality. In the future, it will be important to add new instructions to accelerate the training and inference of CNN and other deep neural networks or at least the hotspot of those networks. In addition, the optimization of software to generate efficient VLIW instructions and support a broad range of high-level functions, not limited to deep neural networks but other application domains, would be interesting for the future exploration.

There is also an opportunity to generate fractions of total instructions (especially "no operation") from hardware instead of software and generate variable length VLIW instructions from the software. An instruction set that supports variable width and variable VLIW instructions would potentially save 50% of the total instruction bandwidth in this architecture. Also, research on interconnection network that connects multiple accelerators and maps deep neural networks differently into that system would be interesting for future work. Finally, the tape-out of the chip also remains an important goal for future work.

REFERENCES

[1] M. Chen, Y. Hao, K. Hwang, L. Wang, and L. Wang, "Disease prediction by machine learning over big data from healthcare communities," *IEEE Access*, vol. 5, pp. 8869–8879, 2017.

[2] S. Ghoshal and S. S. Ghoshal, "Product review platform based on social connections," January 26 2018. US Patent App. 10/007,936.

[3] B. Hidasi, A. Karatzoglou, L. Baltrunas, and D. Tikk, "Session-based recommendations with recurrent neural networks," *arXiv preprint arXiv:1511.06939*, 2015.

[4] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 701–710, ACM, 2014.

[5] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2625–2634, 2015.

[6] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pp. 6645–6649, IEEE, 2013.

[7] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[8] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[9] A. Abeshu and N. Chilamkurti, "Deep learning: the frontier for distributed attack detection in fog-to-things computing," *IEEE Communications Magazine*, vol. 56, no. 2, pp. 169–175, 2018.

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

[11] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning.," in *AAAI*, vol. 4, p. 12, 2017.

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

[13] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

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

[15] S. Ahmad, D. George, J. L. Edwards, W. C. Saphir, F. Astier, and R. Marianetti, "Hierarchical temporal memory (htm) system deployed as web service," May 20 2014. US Patent 8,732,098.

[16] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio, "Robust object recognition with cortex-like mechanisms," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 3, pp. 411–426, 2007.

[17] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[18] J. Appleyard, T. Kocisky, and P. Blunsom, "Optimizing performance of recurrent neural networks on gpus," *arXiv preprint arXiv:1604.01946*, 2016.

[19] R. Rinkus and J. Leveille, "Superposed episodic and semantic memory via sparse distributed representation," *arXiv preprint arXiv:1710.07829*, 2017.

[20] J. Hawkins, S. Ahmad, and D. Dubinsky, "Hierarchical temporal memory including htm cortical learning algorithms," *Techical report, Numenta, Inc, Palto Alto http://www. numenta. com/htmoverview/education/HTM_CorticalLearningAlgorithms. pdf*, 2010.

[21] I. Laptev, M. Marszalek, C. Schmid, and B. Rozenfeld, "Learning realistic human actions from movies," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1–8, IEEE, 2008.

[22] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[23] G. J. Rinkus, "Overcoding-and-paring: a bufferless chunking process and uses thereof," March 17 2015. US Patent 8,983,884.

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

[25] G. J. Rinkus, "Current best performance on mnist," Sept. 22 2016. [Online]. Available: http://www.sparsey.com/MNIST_Results.html. [Accessed Oct. 9, 2018].

[26] W. Dally, "High-performance hardware for machine learning," *NIPS Tutorial*, 2015.

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

[28] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[29] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *arXiv preprint arXiv:1802.09941*, 2018.

[30] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh, *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[31] G. J. Rinkus, "A cortical sparse distributed coding model linking mini-and macrocolumn-scale functionality," *Frontiers in neuroanatomy*, vol. 4, p. 17, 2010.

[32] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[33] W. Shang, K. Sohn, D. Almeida, and H. Lee, "Understanding and improving convolutional neural networks via concatenated rectified linear units," in *International Conference on Machine Learning*, pp. 2217–2225, 2016.

[34] V. Turchenko, E. Chalmers, and A. Luczak, "A deep convolutional auto-encoder with pooling-unpooling layers in caffe," *arXiv preprint arXiv:1701.04949*, 2017.

[35] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

[36] M. Klapper-Rybicka, N. N. Schraudolph, and J. Schmidhuber, "Unsupervised learning in lstm recurrent neural networks," in *International Conference on Artificial Neural Networks*, pp. 684–691, Springer, 2001.

[37] G. J. Rinkus, "Sparsey: event recognition via deep hierarchical sparse distributed codes," *Frontiers in computational neuroscience*, vol. 8, p. 160, 2014.

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

[39] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[40] F. Gers, *Long short-term memory in recurrent neural networks.* PhD thesis, Verlag nicht ermittelbar, 2001.

[41] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[42] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of caffe, neon, theano, and torch for deep learning," *CoRR*, vol. abs/1511.06435, 2015.

[43] B. J. Erickson, P. Korfiatis, Z. Akkus, T. Kline, and K. Philbrick, "Toolkits and libraries for deep learning," *Journal of digital imaging*, vol. 30, no. 4, pp. 400–405, 2017.

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

[45] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

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

[47] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. K. Kim, A. Mishra, and H. Esmaeilzadeh, "Dnnweaver: From high-level deep network models to fpga acceleration," in *the Workshop on Cognitive Architectures*, 2016.

[48] P. D. Franzon, L. Baker, S. Dey, W. Li, and J. Schabel, "Hardware acceleration of sparse cognitive algorithms," tech. rep., NORTH CAROLINA STATE UNIV AT RALEIGH, 2016.

[49] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Tbd: Benchmarking and analyzing deep neural network training," *arXiv preprint arXiv:1803.06905*, 2018.

[50] A. Rush, A. Sirasao, and M. Ignatowski, "Unified deep learning with cpu, gpu, and fpga technologies," tech. rep., Advanced Micro Devices, 2017.

[51] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on fpga," *arXiv preprint arXiv:1511.05552*, 2015.

[52] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 111–118, IEEE, 2015.

[53] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pp. 1–4, IEEE, 2016.

[54] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, "Fpga-based low-power speech recognition with recurrent neural networks," in *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*, pp. 230–235, IEEE, 2016.

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

[56] Z. Wang, J. Lin, and Z. Wang, "Accelerating recurrent neural networks: A memory-efficient approach," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2763–2775, 2017.

[57] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 243–254, IEEE, 2016.

[58] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*, pp. 240–241, IEEE, 2017.

[59] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 393–405, IEEE Press, 2016.

[60] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[61] S. Shin, K. Hwang, and W. Sung, "Fixed-point performance analysis of recurrent neural networks," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 976–980, IEEE, 2016.

[62] J. Ott, Z. Lin, Y. Zhang, S.-C. Liu, and Y. Bengio, "Recurrent neural networks with limited numerical precision," *arXiv preprint arXiv:1608.06902*, 2016.

[63] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[64] X. Wu, Y. Wu, and Y. Zhao, "Binarized neural networks on the imagenet classification task," *arXiv preprint arXiv:1604.03058*, 2016.

[65] G. J. Rinkus, "64x64 natural snippets," September 22 2016. (Accessed on 10/08/2018).

[66] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[67] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[68] S.-A. Reinemo, F. Sem-Jacobsen, and T. Skeie, "Fat-trees and dragonflies: a perspective on topologies," in *Contributed talk at the HPC Advisory Council Workshop, Lugano, Switzerland*, 2012.

[69] C. Nvidia, "Nvidia cuda c programming guide," *Nvidia Corporation*, vol. 120, no. 18, p. 8, 2015.

[70] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.

[71] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 86–96, IEEE, 2013.

[72] A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pp. 273–278, IEEE, 2013.

[73] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[74] D. Kinga and J. B. Adam, "A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, vol. 5, 2015.

[75] R. Lim, Y. Lee, R. Kim, and J. Choi, "An implementation of matrix–matrix multiplication on the intel knl processor with avx-512," *Cluster Computing*, pp. 1–11, 2018.

[76] T. Semiconductor, "Tezzaron diram4 3d memory," Jan. 06 2016. [Online]. Available: https://www.tezzaron.com/applications/diram4-3d-memory/. [Accessed Oct. 9, 2018].

[77] Z. Lu, V. Sindhwani, and T. N. Sainath, "Learning compact recurrent neural networks," *arXiv preprint arXiv:1604.02594*, 2016.

[78] Q. He, H. Wen, S. Zhou, Y. Wu, C. Yao, X. Zhou, and Y. Zou, "Effective quantization methods for recurrent neural networks," *arXiv preprint arXiv:1611.10176*, 2016.

[79] J. C. Jeong, W. Jeong, H. J. Woo, S. H. Kwak, W. C. Park, M. K. Lee, and T. D. Han, "A new pipelined divider with a small lookup table," in *3rd IEEE Asia-Pacific Conference on ASIC, AP-ASIC 2002*, pp. 33–36, Institute of Electrical and Electronics Engineers Inc., 2002.

[80] M. Leeser and X. Wang, "Variable precision floating point division and square root," tech. rep., NORTHEASTERN UNIV BOSTON MA DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, 2005.

[81] O. Vinyals and G. Friedland, "A hardware-independent fast logarithm approximation with adjustable accuracy," in *2008 Tenth IEEE International Symposium on Multimedia*, pp. 61–65, IEEE, 2008.

[82] P.-T. P. Tang, "Table-driven implementation of the exponential function in ieee floating-point arithmetic," *ACM Transactions on Mathematical Software (TOMS)*, vol. 15, no. 2, pp. 144–157, 1989.

[83] H. Bui and S. Tahar, "Design and synthesis of an ieee-754 exponential function," in *Electrical and Computer Engineering, 1999 IEEE Canadian Conference on*, vol. 1, pp. 450–455, IEEE, 1999.

[84] C. Buck, K. Heafield, and B. Van Ooyen, "N-gram counts and language models from the common crawl.," in *LREC*, vol. 2, p. 4, Citeseer, 2014.

[85] T. NVIDIA, "P100-the most advanced datacenter accelerator ever built," *Featuring Pascal P*, vol. 100.

[86] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2592–2600, 2016.

[87] J. Harrison, "Floating point verification in hol light: the exponential function," in *International Conference on Algebraic Methodology and Software Technology*, pp. 246–260, Springer, 1997.

APPENDICES

Appendix A

Sparsey

A.1 Mathematical Steps of CSA

The learning mode of CSA has two phases. In phase I, it calculates the familiarity of the overall inputs (U, H, and D) to the Mac and in phase II, it selects the final set of Q winners per Mac (i.e. the SDC code of Mac for the current input or G-modulated neuron's activation). The computation steps of phase I learning mode are shown below.

A Mac becomes active if either the number of active feature vectors (FVs) of bottom-up receptive field (RF_u) lies within specified limits or it is already active for the number of frames which is less than its persistence.

$$Active(m) = \begin{cases} true & \text{if } \gamma(m) < \delta(m) \\ true & \text{if } \pi_u^- \leq \pi_u(m) \leq \pi_u^+ \\ false & \text{otherwise} \end{cases} \tag{A.1}$$

The signal $Active(m)$ determines whether Mac, m, will be activated or not based on the number of frames, $\gamma(m)$, its persistence, $\delta(m)$, and the user-defined limits of active RF_u, $\pi_u(m)$.

The persistence is activation duration of Mac of each layer, which usually increases along the depth of the network. Every neuron, i in a Mac computes its U, H, and D input summations, $u(i)$, $h(i)$, and $d(i)$. The RF_u, RF_h, and RF_d are the receptive fields from U, H, and D input source of evidence.

$$
\begin{aligned}
u(i) &= \sum_{j \in RF_u} a(j,t) \times w(j,i) \\
h(i) &= \sum_{j \in RF_h} a(j,t-1) \times w(j,i) \\
d(i) &= \sum_{j \in RF_d} a(j,t-1) \times w(j,i)
\end{aligned} \tag{A.2}
$$

The pre-synaptic activation of a neuron, j, at time t and $t-1$ is $a(j,t)$ and $a(j,t-1)$, respectively, and the weight between neuron, i, and pre-synaptic input, j, is $w(j,i)$. Each neuron normalizes every summation to [0, 1] interval independently, yielding $U(i)$, $H(i)$, and $D(i)$, respectively.

$$
\begin{aligned}
U(i) &= \begin{cases} \dfrac{u(i)}{S} & \text{if } L = 1 \\ \dfrac{u(i)}{f_u(RF_u)} & \text{if } L \geq 2 \end{cases} \\
H(i) &= \frac{h(i)}{f_h(RF_h)} \\
D(i) &= \frac{d(i)}{f_d(RF_d)}
\end{aligned} \tag{A.3}
$$

The S represents the number of active bits in the U input matrix for the first layer (L) and the function, $f(.)$, is the placeholder for the summation of neuron's activation of subjacent, same, and top layer's neurons. Each neuron, i, computes the three-way match or the overall local measure of support, $V(i)$, from different input sources. An exponent (λ) is used to modulate the relative sensitivity of various input sources (U, H, and D).

$$
V(i) = \begin{cases} U(i)^{\lambda_u} \times H(i)^{\lambda_h} \times D(i)^{\lambda_d} & \text{if } t \geq 1 \\ U(i)^{\lambda_u} & \text{if } t = 0 \end{cases} \tag{A.4}
$$

At this point, CM calculates $\hat{V}_j$, the neuron with the highest local support or likelihood in each CM. The C_j is the j^{th} CM and i is the neuron within that CM.

$$
\hat{V}_j = \max_{j \in C_q} \{V(i)\} \tag{A.5}
$$

At the end of phase I, each Mac computes G, a global measure of the familiarity of its current overall (U, H, and D) inputs, as the average of the maximum local measure of support over the Q CMs.

$$G = \sum_{q=0}^{Q-1} \frac{\hat{V}_q}{Q} \tag{A.6}$$

At the beginning of phase II, CSA determines the range of expansivity (η) of the activation function based on the global familiarity (G), the number of neurons (K) in a CM, the expansion factor (π), and the threshold value (G_T). The G_T is the user-defined threshold value of global familiarity (G) below which the input is considered entirely unfamiliar.

$$\eta = K \times \pi \times \left[\frac{G - G_T}{1 - G_T}\right]^+ \tag{A.7}$$

The local measure ($V(i)$) of all neurons and the expansivity (η) are passed through a non-linear activation function, yielding the relative probability of winning, $\psi(i)$, of all neurons. The λ is the controlling parameters of overall sigmoid shape.

$$\psi(i) = \frac{\eta}{1 + e^{-\lambda V(i)}} + 1 \tag{A.8}$$

The relative probability of winning, $\psi(i)$, is normalized within each CM to get the final probability of winning, $\rho(i)$, of each neuron in its respective CM. This final probability is used in the WTA contest.

$$\rho(i) = \frac{\psi(i)}{\sum\limits_{j \in CM} \psi(j)} \tag{A.9}$$

The final step of phase II is to select a winner neuron in each CM from the probability distribution. Once a winner is chosen in all Q CMs, the learning rule is applied to update the weights leading to and from the winning neurons. In Sparsey, freezing of learning based on a threshold is applied on Mac to limit the saturation. The detail aspects of the freezing mechanism at different levels are still under exploration [37].

There are two types of retrieval modes in Sparsey's recognition phase, the modified G-computation retrieval mode, and the pure retrieval mode. The modified G retrieval mode is basically based on *back-off* policy where the G is computed from a permissive 2-way match (U and D) instead of more stringent 3-way (U, H, and D) match. In the modified G retrieval mode, the model back off to the 2-way G-computation (G_{ud} is computed from U and D source of evidence) if the 3-way G (G_{hud} is from U, H, and D) falls below a threshold value. Note that in this retrieval mode, all stored SDC codes first compete using highest-order G and then progressively do so using lower-order G based on a user-defined threshold. The computation complexity of Sparsey is still constant i.e. does not change with the *back-off* policy. The details aspect of the *back-off* policy is still under exploration. In the pure retrieval mode, each Mac activates SDC code of the learning moment based on the most closely matched with its current moment. In this retrieval mode, a simple winner choice method is used in which, in each CM, the neuron with the highest local likelihood value is chosen as the winner. In this mode, the model is asked to find the closest match and not asked to learn anything new. The optimal pure retrieval mode could be maximum likelihood estimation in conjunction with the back-off policy. The deterministic algorithm called "maximum likelihood" method is much faster since it obviates the latter steps of the CSA and for many applications, it yields much better performance and higher expected accuracy.

Table A.1: The fundamental differences between Sparsey and DNN

Aspect	Sparsey	Traditional DNN
Connection Number	Very high	Relatively less
Connection Type	Binary	Floating-point
Weight Matrix	Sparse binary	Dense floating-point
Neuron Number	Extremely high	Relatively less
Training	Unsupervised/semi-supervised	Mostly supervised
Learning Mechanism	Continuous	Batch
Learning Method	A memory-predication	Representations of data
Memory Bandwidth	Extremely high	High
Model Development	Easy	Hard
Biological Nature	Realistic	Simple
Application	Anomaly detection	Classification

A.2 Comparision with DNN

The fundamental differences between Sparsey and traditional deep neural networks (i.e. MLPs, CNNs, RNNs, etc.) has analyzed from architectural, computational, and storage point of view and listed in Table A.1.

Appendix B

LSU of ASIP

B.1 Instruction of Load and Store

The eight different load-store instructions and their address formats are discussed in this section. Table B.1 lists the "no operation" for load-store execution lane in a PE. In Table B.2 and Table B.3, data load from SRAM to *vreg* and data store to SRAM from *vreg* are explained. The 24^{th} and 11^{th} bit of load-store instructions represent the bank address of SRAM, respectively. Similarly, Table B.4 and Table B.5 explain the load-store instructions between SRAM and *hreg* register file.

Table B.6 represents the load instruction for loading 16-bit data from SRAM to *vreg* register file as data fanout manner. As discussed in this dissertation, this type of loading is necessary for scaler-matrix multiplication, which is required for adjusting the loss of mixed precision training, scaling the weight gradients of backward propagation, etc. Table B.7 shows the similar load instruction for SRAM and *hreg*. Table B.8 shows the load instruction from *sreg* to SRAM.

Table B.1: The no operation instruction

31–29	28–25	24–0
101	0000	XX–XX

Table B.2: The load instruction from SRAM to vreg

31–29	28–25	24	23–13	12–7	6–0
101	0001	SRAM Bank	SRAM Address	XXXXXX	vreg Address

Table B.3: The store instruction from vreg to SRAM

31–29	28–25	24–20	19–13	12	11	10–0
101	0010	XXXXX	vreg Address	X	SRAM Bank	SRAM Address

Table B.4: The load instruction from SRAM to hreg

31–29	28–25	24	23–11	10–4	3–0
101	0011	SRAM Bank	SRAM Address	XXXXXXX	hreg Address

Table B.5: The store instruction from hreg to SRAM

31–29	28–25	24–18	17–14	13	12–0
101	0100	XXXXX	hreg Address	SRAM Bank	SRAM Address

Table B.6: The load instruction from SRAM to vreg

31–29	28–25	24	23–8	7–0
101	0101	SRAM Bank	SRAM Address	vreg Address

Table B.7: The load instruction from SRAM to hreg

31–29	28–25	24	23–8	7–4	3–0
101	0110	SRAM Bank	SRAM Address	XXXX	hreg Address

Table B.8: The load instruction from sreg to SRAM

31–29	28–25	24–19	18–14	13	12–0
101	0111	XXXX	sreg Address	SRAM Bank	SRAM Address

Appendix C

SFU of ASIP

C.1 Pseudocode of Special Function

The pseudocode of the algorithm implemented in special function unit (SFU) is given in Algorithm C.1. This is a lookup table based method to calculate the floating-point exponential function [87]. Algorithms use to calculate the sigmoid and tangent hyperbolic function are give in Algorithm C.2 and Algorithm C.3, respectively.

C.2 Lookup Table Content

The constants (A1 and A2) are calculated from the polynomial approximation given in equation (C.1). The constants and contents of LUT are listed in Table C.1 and Table C.2 using single-precision. In equation (C.1), A1 and A2 are the coefficients of the Taylor series. The content of LUT and constant are used from [82]. The details calculation of LUT content are also described in [82], which is the values of $2^{j/32}$ where $j = 0, 1, ..., 31$. These values are broken into two parts (the content of LUT1 and LUT2) considering the six trailing bits of the LUT2's content are zero.

$$p(t) = t + A_1 t^2 + A_2 t^3 \tag{C.1}$$

Algorithm C.1: The Algorithm of Exponential Function

$Int_{32} \leftarrow 32$; $Int_{2e9} \leftarrow 512$; $Plus_{one} \leftarrow 1$
$THRESHOLD\_1 \leftarrow 220.421$
$THRESHOLD\_2 \leftarrow 2.98023e^{-08}$
$Inv_L \leftarrow 46.1662$; $L1 \leftarrow 0.0216608$; $L2 \leftarrow 4.4644e^{-08}$
$A1 \leftarrow 0.500004$; $A2 \leftarrow 0.166668$
if $x = NaN$ **then**
  $f(x) \leftarrow x$
end if
else if $x = +infinity$ **then**
  $f(x) \leftarrow +infinity$
end if
else if $x = -infinity$ **then**
  $f(x) \leftarrow 0$
end if
else if $abs(x) > THRESHOLD\_1$ **then**
  if $x > 0$ **then**
    $f(x) \leftarrow +infinity$
  end if
  else
    $f(x) \leftarrow 0$
  end if
end if
else if $abs(x) < THRESHOLD\_2$ **then**
  $f(x) \leftarrow x + 1$
end if
else
  $N \leftarrow (int)(x.Inv\_L)$
  $N2 \leftarrow N\%Int\_32$
  $N1 \leftarrow N - N2$
  if $abs(x) >= Int_{2e9}$ **then**
    $R1 \leftarrow (x - N1.L1) - N2.L1$
  end if
  else
    $R1 \leftarrow x - N.L1$
  end if
  $R2 \leftarrow -N.L1$
  $M \leftarrow N1/Int_{32}$
  $R \leftarrow R1 + R2$
  $Q \leftarrow (A1 + R.A2).R.R$ $P \leftarrow R + Q$
  $S \leftarrow LUT1[N2] + LUT2[N2]$
  $E \leftarrow S + S.P$
  $f(x) \leftarrow E.2^M$
end if

Algorithm C.2: The Algorithm of Sigmoid Function

$N \leftarrow$Algorithm C.1(x)
$D \leftarrow (1 + N)$
$f(x) \leftarrow N/D$

Algorithm C.3: The Algorithm of Tangent Hyperbolic Function

$x \leftarrow 2.x$
$N \leftarrow$Algorithm C.1(x)
$N \leftarrow N - 1$
$D \leftarrow N + 1$
$f(x) \leftarrow N/D$

Table C.1: The constants of polynomial

Name	Value
A1	3F000044
A2	3E2AAAEC

Table C.2: The content of lookup tables

Index	LUT1	LUT2	Index	LUT1	LUT2
0	3F800000	00000000	16	3FB504C0	36CCCFE7
1	3F82CD80	35531585	17	3FB8FB80	36BD1D8C
2	3F85AAC0	34D9F312	18	3FBD0880	368E7D60
3	3F889800	35E8092E	19	3FC12C40	35CCA667
4	3F8B95C0	3471F546	20	3FC56700	36A84554
5	3F8EA400	36E62D17	21	3FC9B980	36F619B9
6	3F91C3C0	361B9D59	22	3FCE2480	35C151F8
7	3F94F4C0	36BEA3FC	23	3FD2A800	366C8F89
8	3F9837C0	36C14637	24	3FD744C0	36F32B5A
9	3F9BBD00	36E6E755	25	3FDBFB80	36DE5f6C
10	3F9EF500	36C98247	26	3FE0CCC0	36776155
11	3FA27040	34C0C312	27	3FE5B900	355CEF90
12	3FA5FEC0	36354D8B	28	3FEAC0C0	355CFBA5
13	3FA9A140	3655A754	29	3FEFE480	36E66F73
14	3FAD5800	36FBA90B	30	3FF52540	36F45492
15	3FB123C0	36D6074B	31	3FFA8380	36CB6DC9