Addressing the algorithmic gap in low-precision neural network substrates

By

Rohit Shukla

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: 8/30/2018

The dissertation is approved by the following members of the Final Oral Committee:

Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Yu Hen Hu, Professor, Electrical and Computer Engineering
Jing Li, Assistant Professor, Electrical and Computer Engineering
Dimitris Papailiopoulos, Assistant Professor, Electrical and Computer Engineering
Stephen Wright, Professor, Computer Sciences

ACKNOWLEDGMENTS

First and foremost I would like to sincerely thank my parents, Surendra Nath Shukla and Neerja Shukla, and my wife Meatrayi Sharma. Their unwavering support, motivation and guidance, was the most important contribution towards achieving my goal of receiving a Ph.D.

Second, I would like to express my deepest gratitude to my advisors Prof. Mikko Lipasti for his mentorship and continuous support throughout the course of my graduate studies contributed in achieving this goal. Words fail in expressing my appreciation for Prof. Mikko Lipasti, because of his guidance I got an opportunity to work on the research projects which I completely enjoyed and his passion for exploring novel and practical solutions for challenging problems has been inductive and I feel fortunate to be his student.

I?d like to thank my undergraduate advisor, Dr. Kailash Chandra Ray for introducing me to the research area of VLSI architecture and signal processing. It was because of his motivation I got the courage to apply for graduate school and pursue this PhD.

I also want to acknowledge Prof. Stephen Wright and Dr. Jing Li, for sharing their knowledge and experience. Their commitment to several events and productive discussions helped in understanding some obscure concepts and kindled new ideas.

I would also like to thank my friends and colleagues who have provided me with insightful discussions and fellowship. Special acknowledgements to Dibakar Gope, David Palframan, Gokul Subraminian, Ravi Raju, Carly Schulz, Michael Mishkin, David Schlais, Heng Zhou and Kyle Daruwalla for their friendship, collaboration, discussions and support. Special thanks go to Erik Jorgensen for his help with implementing the linear solver

prototype on TrueNorth, Soroosh Khoram for his expertise and analysis on error analysis, and Kyle Daruwalla for further extending this work to SVD implementation of the algorithm and providing insights on how to make stochastic computing based implementations better.

I also thank the staff in the Electrical and Computer Engineering department for making my academic life at University ofWisconsin-Madison a great experience. I also thank the professors from systems, computer architecture and machine learning groups for all the things that I have learned from their respective courses.

This work was supported in part by NSF Award CCF-1628384.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With the recent popularity of machine learning and increase in demand for low power computing, researchers are investigating alternative architectures that can operate on streaming input data for real-time applications. These constraints put up a challenge for existing microarchitects to come up with novel computing techniques that can perform a variety of computations with limited resources. One such alternative computing technique is stochastic computing where the input data is represented as single bitstreams. By reframing algorithms under the stochastic computing paradigm, designers can also take advantage of the energy efficiency of ultra low-power FPGAs and IBM's TrueNorth Neurosynaptic System. For example, a recurrent Hopfield neural network can be used to find the Moore-Penrose generalized inverse of a matrix, thus enabling a broad class of linear optimizations to be solved efficiently, at low energy cost. However, deploying numerical algorithms on hardware platforms that severely limit the range and precision of representation for numeric quantities can be quite challenging. This dissertation discusses these challenges and proposes a rigorous mathematical framework for reasoning about range and precision on such substrates. The dissertation derives techniques for normalizing inputs so that solvers for those systems can be implemented in a provably correct manner on hardware-constrained neural substrates. The analytical model is empirically validated on the IBM TrueNorth platform, and results show that the guarantees provided by the framework for range analysis. The Hopfield linear solver model is empirically validated on the IBM TrueNorth and stochastic computing platform, and results show promising potential for deploying an accurate and energy-efficient generalized matrix inverse engine

calculator, with compelling real-time applications including target tracking (object localization), optical flow, and inverse kinematics. Experiments with optical flow demonstrate the energy benefits of deploying a reduced-precision and energy-efficient generalized matrix inverse engine on the IBM TrueNorth platform, reflecting $10\times$ to $100\times$ improvement over FPGA and ARM core baselines. Moreover, we combine designs from SC with a biological encoding scheme called population coding to alleviate the long latency associated with SC. Using the techniques proposed, we achieve up to 25.56x speedup with population coding scheme, 7x reduction in area and 275x reduction in energy consumption.

1 INTRODUCTION

This chapter serves as an introduction for the readers to look at and understand the problem of learning invariant transformations in visual cortex. In this chapter we present readers with the motivation to understand how the visual cortex is able to learn different affine transformations using approximate computing units called spiking neurons and why this problem would be useful for various real-time applications. Here readers will go through different research ideas that we have published and proposed that looks at mathematically formulating unsupervised learning of affine transformations and later using these formulations for low-power hardware constrained applications such as Micro Aerial Vehicles (MAVs).

1.1 Motivation: Learning invariant transformations in visual cortex

The human visual system is very adept at recognizing objects. Even before a human infant first opens its eyes, spontaneous retinal activations are driving development of the visual system [3, 14]. From the very beginning the infant is able to recognize its mother's face. An early developmental task is to organize this input into objects and learn to recognize them despite variations in scale, rotation, and position in the visual field. As humans grow they start recognizing objects moving in complex scenarios. How is it that our eye is able to learn these transformations and be able to store them so that we can use them later for object recognition?

The human visual system is efficient in recognizing and classifying objects, but computers are still not robust enough to process the visual information in the same way as humans do [105]. A considerable number of articles have been published that discuss about how humans are able to recognize objects based on their invariant features and later extend the concepts of human visual system to computer visions for robust object recognition and classification [82], [66], [34] and [45]. One question that remains to be addressed is how a human being is able to learn invariant transformations that map the seen object to the reference object present in the memory. Prior work such as [48], [51], and [83] have addressed how a human brain is able to store invariant features of an object, and proposed unsupervised learning mechanisms such as Hebbian learning, that are able to store these invariant features. To the best of our knowledge, there has been no other work that has proposed a computational model which explains how the mammalian visual system learns invariant transformations such as translation, rotation and scaling, and organizes them into independent layers.

We proposed an unsupervised learning method that demonstrated with simple matrix algebra how mappings may emerge in human visual system to distinguish among independent image transformation functions and is also able to group similar transformations together in the context of map-seeking-circuits [6](please refer to [92]). Based on recognition of object permanence, the proposed algorithm matches reference pattern with the input pattern by comparing an ordered list of interesting or invariant features present in the two images and later learn the necessary affine transformation to match the two features. The work done in [92] have proposed that the human brain is able to learn and retain invariant transformations using spontaneous activation feature of the eye and its

ability to recognize temporal invariant features of the image, through object permanence and temporal association, .

In this dissertation we focus on computational framework that proposes how human cortex is able to learn to recognize invariant transformation mappings between the information appearing on the retina and the visual model present in the memory, without any prior information. Our goal is to have a mathematical framework using which we can reason about how spiking neurons are able to estimate the affine transformations using an unsupervised learning approach. Results suggest that the proposed computational model may be a key to understanding the way that the primate visual system is able to identify various affine transformations with its remarkable processing speed and its low energy consumption. These mechanisms are also interesting for artificial vision systems and robotics systems, particularly for hardware solutions.

1.2 Motivation: Neuromorphic computing

Recent advances in neuromorphic engineering [88] have motivated the development of neural hardware substrates that are tailored to loosely emulate computations that happen in a human brain with extremely low power and efficiency. Examples include IBM TrueNorth Neurosynaptic System [67], NeuroFlow [16], Neurogrid [9], SpiNNaker [29], and the BrainScaleS project [87], all of which are implemented using Si CMOS. While Si CMOS is the prevailing technology, the slowdown in transistor scaling has led to broad interest in spiking neural network substrates that exploit the unique properties of emerging nonvolatile memory such as memristor crossbar [72] and RRAM [32]. Due to the close match between

the algorithmic requirements and the underlying hardware architecture, such designs have the potential to achieve much better computational efficiency than the conventional Si-CMOS based designs.

In spite of the radically differing hardware implementations of these neural network substrates, many of them share an inherent design principle: converting input signal amplitude information into a rate-coded spike train and performing parallel operations of dot-product computations on these spike trains, based on synaptic weights stored in the memory array. These similarities also result in a set of common challenges during practical implementation, especially when using them as computing substrates for applications with a mathematical algorithmic basis. These challenges include a restricted range of input values and the limited precision of synaptic weights and inputs. Since a value is encoded in unary spikes over time (i.e. as a firing rate), each individual input and variable must take a value in the range $[0, 1]$. Furthermore, the precision of the encoded value is directly proportional to the size of the evaluation window, which, for reasons of efficiency, is typically limited to a few hundred spikes. Finally, because of hardware cost, synaptic weights can be implemented only by a limited number of memory bits, resulting in limited precision. For instance, IBM's TrueNorth supports 9-bit signed weight values.

Mapping existing algorithms to these substrates requires the designer to choose a strategy for quantizing inputs and weights carefully, so that the range limitations are not violated (i.e. values represented by firing rates do not saturate), while maintaining sufficient precision. Prior work notes these challenges, but typically presents only ad hoc solutions that choose scaling factors and quantization strategies based on empirical measurements that can guarantee correct operation for the tested scenarios, but provides no guarantees

in the general case [46], [91]. Error analysis for feedforward networks appear in [37], but omits recurrent networks and range analysis.

1.3 Low-power computing for robotics applications

The recent popularity of machine learning (ML) has lead to several advancements in architectures designed to accelerate evaluation in applications such as artificial neural networks (ANNs). At the same time, utilizing these complex ML models comes at an increased energy cost, motivating researchers to consider low power computing paradigms such as stochastic computing (SC) and neuromorphic computing. But the focus so far has been to emulate the functionality of ANNs using stochastic computing. While this trend has helped alleviate the energy cost of ANNs, it fails to extract the full potential of stochastic bitstreams.

While prior work regarded stochastic bitstreams as a low power implementation of ANNs, we consider stochastic computing as its own emerging paradigm. Recent advancements in control systems and MEMS fabrication has enabled new energy constrained applications such as micro aerial vehicles (MAVs) [27]. One such popular MAV is RoboBee [26] [22]. Due to the limited energy budget in MAV based applications, microcontrollers are not an option for the control system, and current work uses ASICs instead. Unfortunately, these ASICs can only perform a limited set of algorithms (e.g. optical flow) that are required to keep the robotic bee in flight. While the control of a robotic flapping wing insect is technically impressive within the robotics community, the overall utility of the robot is low if it cannot perform any extra functions. Indeed, the prior work indicates

that a RoboBee should be able to perform the duties of biological bees, perform aerial surveillance of crops, collect weather data, and assist search and rescue teams [26]. But current implementations do not support these functions, because they lack a computing platform capable of performing these complex algorithms while consuming less than 10% of the available power budget [26].

In addition to performing optical flow for stable flight control, a robotic bee should be able to perform object recognition and tracking, inverse kinematics, and navigation and path planning. First, object tracking and inverse kinematics allows the bee to detect and avoid static and moving obstacles or to identify targets in search and rescue missions. Second, optical flow and navigation allows the bee to mimic a biological bee's duties such as cross-pollination or to automate pathing during surveillance. All these functions can be implemented with basic matrix operations such as least squares minimization. These algorithms can be potentially implemented using stochastic computing hardware which consumes few enough resources to be mapped to ultra low-power FPGAs and at the same time the proposed methods can be extended to digital neuromorphic computing hardware like IBM TrueNorth.

1.4 Objectives and Contributions

This dissertation has focused on considerations that have to be kept in mind when mapping algorithms on a low-precision neural network hardware. In this dissertation we present the theoretical similarities between performing matrix computations on bitstreams using stochastic computing hardware and neuromorphic computing hardware. We show that

arithmetic computing theories that have been developed for stochastic computing can be applied to spiking neural networks, and similarly, theories that have been developed in neuroscience such as population coding, can be used to improve performance of stochastic computing hardware. The mathematical framework has been developed rigorously to reason about various computation steps and different hardware implementation styles have been explored to understand strength and limitations of each approach.

1.4.1 Range analysis

This first major contribution of this dissertation develops a rigorous mathematical model that enables a designer to map numerical algorithms to these substrates and to reason quantitatively about the range. Unlike prior research [13] [100] [37] our mathematical framework can be applied to a wide range of problems in linear optimization running on neural substrates with diverse constraints. The model is validated empirically by constructing input matrices with random values and computing matrix inverse using a recurrent Hopfield neural-network-based linear solver. Our results show that the scaling factor derived by the mathematical model hold for this application under a broad range of input conditions. We report the computing resources and power numbers for real-time applications, and quantify how the errors and inefficiencies can be addressed to enable practical deployment of the Hopfield linear solver.

1.4.2 Implementation

Later in chapter 5 we will discuss how the proposed Hopfield neural network algorithm is implemented for stochastic computing platform. Initially we will look at a stochastic computing datapath for implementing the proposed linear solver, followed by how the TrueNorth neuron parameters have to be configured so that their calculation behavior is the same as stochastic computing elements. This system will be end-to-end setup that will guarantee all of the computations are happening on input random bitstreams.

1.4.3 Population coding

The next major contribution of this dissertation is implementing a population coding approach to reduce the computing latency of Hopfield linear solver. Prior research on stochastic computing has suffered from long latency, that is, the proposed algorithms need longer duration of compute to achieve the desired accuracy. We propose that borrowing ideas of population coding from neuroscience, we can apply it to stochastic computing, as a result, reducing the computation latency considerably. To the best of author's knowledge this is the first literature within stochastic computing where an architecture has been proposed that addresses the issue of latency by having multiple similar stochastic computing units operating in parallel.

1.4.4 Adaptive scaling technique

In this topic we have proposed a stochastic computing architecture that can scale the input values adaptively to ensure the intermediate computing results never saturate. Unlike the

range analysis proposal that we had presented in subsection 1.4.1, the scaling factor will be computed on the stochastic computing hardware itself. We won't need any additional mechanism to calculate the scale factor offline, rather the scaling factor can be estimated online for continuous input bitstreams.

1.4.5 Architectural benefits of stochastic computing

We will conclude this dissertation with understanding the architectural benefits of implementing the proposed iterative algorithm on an ultra low-power lattice FPGA and compare it with more standard implementation approaches. In this part we evaluate how the proposed architecture can be extended to robotics applications such as Micro-Aerial Vehicles which have stringent performance and power requirements.

1.5 Related published work

This dissertation encompasses these previously published work

- **A self-learning map-seeking circuit for visual object recognition (IJCNN 2015)** This work describes a self-learning approach to identify orthogonal affine transformations and group similar affine transformations into various layers of Map-Seeking Circuits. [92]. This work was co-authored with Mikko Lipasti.

- **Evaluating Hopfield-network-based linear solvers for hardware constrained neural substrates (IJCNN 2017)** In this work evaluate the TrueNorth based implementation of proposed Hopfield linear solver for three different real-time robotics applica-

tions such as object tracking, inverse kinematics and optical flow [91]. This work was co-authored with Erik Jorgensen and Mikko Lipasti.

- **Computing Generalized Matrix Inverse on Spiking Neural Substrate (Frontiers in Neuroscience: Neuromorphic Engineering 2018)** This work presented the algorithm to implement the Hopfield neural network structure on a stochastic computing platforms as well as on IBM TrueNorth neurosynaptic system [93]. This publication also presented the mathematical formulation for range analysis. This work was co-authored with Soorosh Khoram, Erik Jorgensen, Mikko Lipasti, Jing Li and Stephen Wright.

- **A Case for Hardware/Software Codesign of Bitstream Computing (ASPLOS 2019 - under review)** This work presented the benefits of population coding approach for stochastic applications and compared benefits of FPGA based SC implementation against more standard approaches. This work was co-authored with Kyle Daruwalla, Heng Zhou and Mikko Lipasti.

1.6 Dissertation structure

The rest of this dissertation is organized as: Chapter 2 provides background material relating to artificial neuron models, recurrent neural networks, and a brief discussion on the iterative algorithm to compute matrix pseudoinverse. Chapter 3 presents how to perform various arithmetic calculations using stochastic computing based digital logic and this chapter also talks about how to extend these stochastic computing logic to digital

spiking neural network hardware such as IBM TrueNorth Neurosynaptic system. Chapter 4 presents the mathematical framework for range analysis that would guarantee that the intermediate computations would never saturate. Chapter 5 presents the algorithm to implement the proposed iterative algorithm on any stochastic computing substrate, including IBM TrueNorth. Chapter 6 proposes the architecture for population coding approach and chapter 7 proposes adaptive scaling architecture. Both of these chapters together are meant to address the issue of long latency that occurs when doing computations with stochastic computing units. Chapter 8 provides the details of experimental setup that was used to evaluate the proposed architecture and algorithm. Chapter 9 shows the results of how the implemented Hopfield neural network based linear solver performs when tested with the different experimental setup that were proposed in chapter 8. Finally, chapter 10 concludes the dissertation and discusses potential future directions for this research.

2 BACKGROUND

Purpose of this chapter is to familiarize the readers with the challenges and opportunities present in solving inverse problems using the computation scheme of the human brain. This chapter is divided into three subsections. First, we present the motivation behind considering artificial and spiking neural networks. Second, we look at the prior work that has been done in the context of solving systems of linear equations using biological neural networks. Third, we will look at the Hopfield neural network iterative algorithm that we implement to compute generalized matrix inverse.

2.1 Artificial Neural Network

Artificial neural networks are a collection of compute nodes, also called neurons that represent the mathematical abstraction of the way calculations happen in a biological neuron. With the recent success in the domain of deep learning, ANNs have shown unprecedented results in the domain of object recognition, localization and detection. Modeled after the biological neuron, ANNs have shown promising applications in the areas of computer vision, natural language processing, control systems, financial analysis, etc.

Fig. 2.1, shows the image of a biological neuron. A biological neuron receives input electrical signals (input values) from other neurons through an axon. These input signals are integrated with the neuron's membrane potential. The value by which the input signals are integrated depends on synaptic strength (or synaptic weight) between the axons and the neurons. Once the membrane potential of the neuron reaches a certain value, it will produce

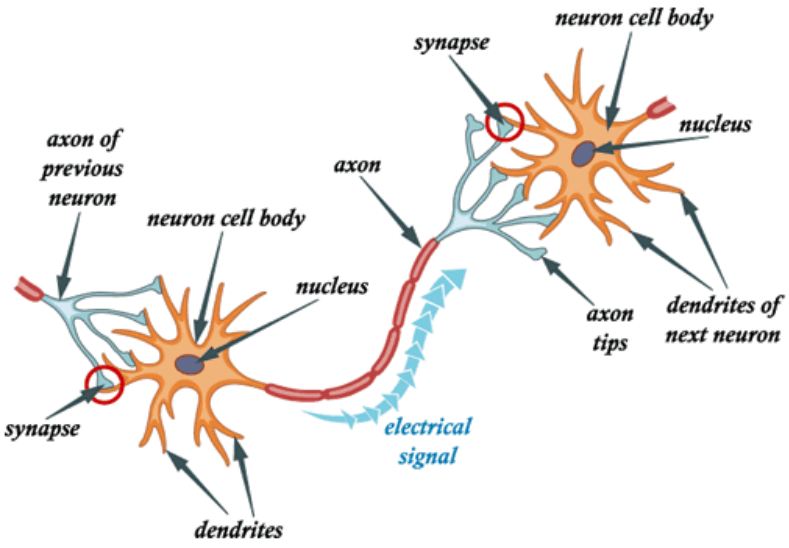


Figure 2.1: Image of a biological neuron. This image has been taken from [12].

an output electrical signal (output value) that gets sent down to the neuron's dendrites and this output signal is later transmitted to other neurons through their respective axon.

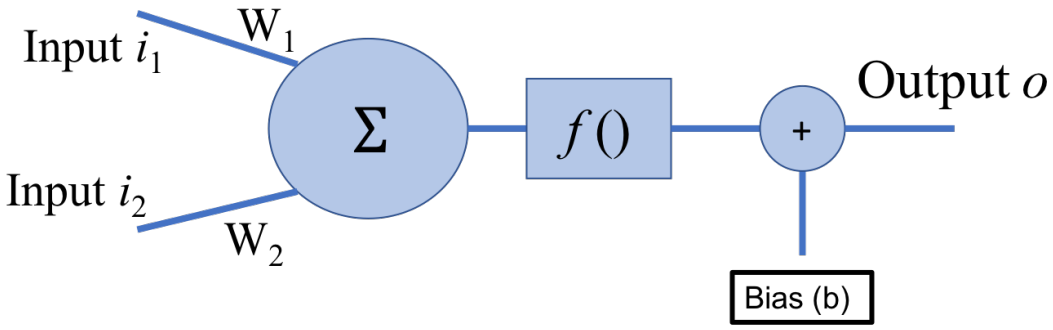


Figure 2.2: Mathematical model of a single neuron in an artificial neural network.

Fig. 2.2 shows a simple representation of an artificial neuron. The input values (i_1 and i_2) and the output value (o) are the rate-encoded representations of spike based calculations that happen in a biological neuron. Synaptic strength between the axons and the neurons is represented with the parameters W_1 and W_2. After the integration operation a linear or non-linear activation function f() is applied to the summation result followed by addition

of a bias term b. The resultant output value is sent to other neurons for processing.

$$Y = f(W^T X) + b \tag{2.1}$$

Due to the huge demand of ANN based applications, this has motivated the hardware architecture community to propose hardware implementations of neural networks that have low latency, high throughput and are energy efficient . Even though there have been considerable number of design proposals from hardware community with various energy-efficient based implementations, one of the common themes among these proposals is to train the neural network off chip, usually using a GPU, then perform operations such as quantization or pruning to cut down on the number of operations and finally map these trained neural networks onto the proposed hardware platforms. Although there have been numerous variants of neural networks such as Hopfield network, Restricted Boltzman machine, autoencoders, etc., majority of the hardware neural network models have only considered convolutional neural networks, fully connected networks or long-short term memory networks. As a result, researchers that are considering low-power neural network based solutions for robotics applications have to either propose a custom ASIC that can solve a subset of problems or consider off-the shelf CPUs or GPUs that end up consuming a considerable amount of power.

2.2 Spiking neural networks

Spiking neural networks are third generation of neural networks that closely mimic the computing features of the human brain [62] . Unlike ANNs, these spiking neurons closely

resemble the computing features of a biological neuron. Many models of spiking neurons have been proposed, ranging from simple integrate and fire models to complex synaptic-conductance based models which use a large set of differential equations to describe the behavior of the neuron and its synapses [42]. However, the most common aspect between these different models is that neurons communicate via spikes (as opposed to rates) and integrate these spikes over time, giving spiking neurons a concept of time that is absent from more traditional rate-coded models or ANNs.

Spiking neural networks have gained interest among the neuroscientific community, because researchers can use these models to better understand the computational behavior of the human brain which is involved in the process of learning and complex decision making. At the same time, the hardware architecture community is looking at deploying these biological solutions for low-power computing based commercial applications.

2.2.1 Leaky-Integrate and Fire model

The leaky integrate and fire model (LIF) proposes the basic computing properties of biological spiking neurons. As per the LIF model, neurons communicate through spikes and neurons integrate spikes over time. Eqn 2.2 shows the differential equation of LIF model.

$$\frac{dV}{dt} = \frac{1}{\tau_m}(-V + IR_m) \tag{2.2}$$

In the eqn. 2.2, V is the current membrane potential of the neuron, R_m is the membrane resistance, I is the input current to the neuron, and m is the time constant of the membrane.

If the membrane potential reaches a firing threshold, the neuron emits a spike, and is reset to a resting membrane potential.

In LIF model, neurons are considered as the basic compute unit. As the neuron receives input spikes (the term I in eqn. 2.2), its internal voltage (the term V in eqn. 2.2) is changed. If the inputs are excitatory in nature, the neuron membrane potential will increase; on the other hand, if the neuron inputs are inhibitory , the neuron membrane potential will decrease. This membrane potential decays as a function of time in the absence of inputs (referred to as the membrane leak), and eventually stabilizes at a resting voltage (the term R_m in eqn. 2.2). However, if a neuron receives many strong excitatory inputs (at once, or across time at a rate greater than the membrane leak), the membrane reaches a critical firing threshold, produces a spike, and is set to a reset voltage. This spike travels down the neurons axon (its output), which synapses with the dendrites of other neurons. It should be noted that the communication between neurons (that is, the communication between the output of one neuron and the input of the other) is typically considered to be a chemical, rather than electrical, process. The axon of the presynaptic neuron releases neurotransmitters after an output spike, which in turn, are absorbed by the neuroreceptors on the dendrites of the postsynaptic neuron.

LIF neuron models typically have a set of parametrizable variables, such as the synaptic weights corresponding to each of the dendrites, the reset and a resting membrane potential values, the firing threshold, the leak of the membrane potential, and other stochastic elements and transcendental operations [13]. Figure 2.3 also highlights the role of each of these parameters during the LIF neuron's computation. At each time step, the neuron integrates its inputs by evaluating the dot-product of the neuron's synaptic weights (W1,

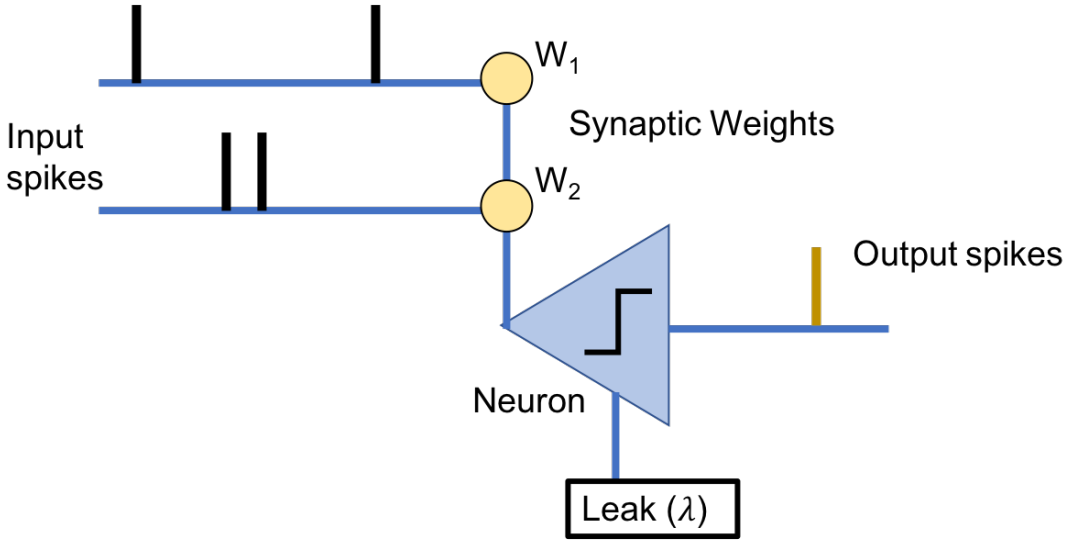


Figure 2.3: Mathematical model of a single neuron in a spiking neural network. Unlike ANNs, computations in a spiking neuron happens using spikes.

and W2)). This value is then added to the membrane potential of the LLIF neuron. Next, a leak parameter is added to the updated membrane potential and afterwards the result is compared with the threshold value. If the resultant membrane potential exceeds the threshold, the neuron generates a spike that propagates down the axon to other neurons. The membrane potential of the neuron is then set to a pre-specified reset potential. If the neuron does not generate a spike, the membrane potential leaks, which models the tendency of biological neurons to drift towards a resting potential.

The LLIF neuron can be further extended to perform operations such as stochastic computing, which is a very popular theory in the context of digital VLSI. The relationship between spiking neural networks and stochastic computing will be discussed in detail in chapter 3.

2.3 Biological models to solve the inverse problem

Systems of linear equations can be used to describe a broad class of computational problems with compelling practical applications. For example, the mammalian visual system is continuously attempting to map visual stimuli to objects stored in memory in spite of variations in scale, rotation, and position in the visual field. Numerous prior works have addressed the mammalian visual system's ability to recognize objects based on their invariant features, and have extended these concepts from the mammalian visual system to computer vision algorithms that enable robust object recognition and classification [82], [28], [6] and [83]. One question that remains to be addressed is how a human being is able to learn the invariant transformations that map the seen object to the reference object present in the memory. And once this mechanism has been understood, can it be mapped onto low precision spiking neural network platform such as IBM TrueNorth?

Recently deep learning communities have proposed neural network architectures in which affine transformations are stored as a separate layer of abstraction [43] [84]. These biologically inspired deep learning models have shown the benefits of having affine transformations stored as a separate abstraction layer would result in reduced training time without any accuracy loss. Prior biological models such as HMAX[69], and Map-Seeking Circuits [6] have addressed the problem of transformation discovery and proposed a computational model which explains how the mammalian visual system learns invariant transformations such as translation, rotation and scaling, and re-organizes itself as a new input stimuli is presented so that the input object can be mapped to the object stored in the

memory. In the following subsections we give a brief overview of how these two biological models are able to solve for system of linear equations.

2.3.1 Solving system of linear equations using HMAX

In [77] and [5] authors have proposed a multi-layer hierarchical architecture of visual system that discounts image transformations and generates a discriminative feature vector, or signature, for learnt images. These signatures are invariant to local and global affine transformations. Sample complexity of learning models reduces significantly if these models are able to factor out image transformations during the development phase, as a result, achieving the goal of recognition with one or very few labeled examples. The proposed conjecture is to find a computational model for the ventral stream that learns to factor out image transformations. The formalization and proof of the conjecture was left as an open problem. The articles state that the ventral stream learns and stores a group of transformations (G) as seen through the aperture of receptive field. The human eye stores an image as image patches (t^k, where $k \in [1, K]$ are the K templates of image patches stored in the memory) in the memory and simple cells present in human eye stores the transformed image patch ($g_i t^k$, where $\forall i, g_i \in G$). This learning operation is performed through Hebbian learning mechanism. A one-layer architecture can store all of the transformed image patches and achieve global invariance, whereas, to achieve local invariance and robust signatures for different parts of the image, authors have proposed a multi-layer hierarchical architecture that is similar to HMAX algorithm [77]. In the implemented models authors have hardwired translation and scaling transformations.

2.3.2 Map-Seeking Circuits

Map-Seeking Circuits is a biologically inspired algorithm developed by David Arathorn [6]. It is a model-based approach that assumes that the correspondence between a model and an observation of it can be represented by a decomposition of invertible transformations, such as scaling, translation and rotation. Unlike the HMAX algorithm [82] and other similar feedforward models, which simply report the presence (or absence) of a target object in the reference image or not, MSC seeks to find an appropriate set of transforms that maps a stored template to an unknown signal, returning both the recognition/classification response as well as the set of maps used to identify the object. The algorithm uses super-position with an iterative matching process to converge on the best set of transforms that map a template to a target in an input signal.

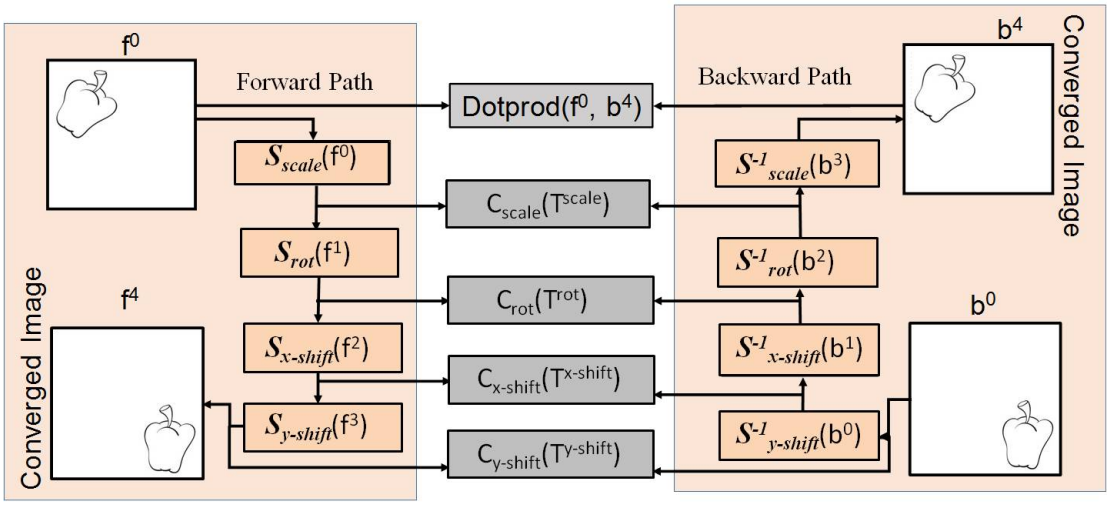


Figure 2.4: An image depicting the flow of calculations that happen in map-seeking circuits. This image has been taken from [92]

MSC is composed of one or more layers and a set of templates. Each layer represents a transformation such as translation, scale or rotation as shown in fig. 2.4. The algorithm

performs a set of transforms at each layer and sums the result. The result is then sent to the following layer where the process is repeated for another set of transformations. The algorithm depends on The Ordering Principle of Superposition [7]. The principle states that if matches are computed between a pattern, A, and a superposition of a set of patterns, the match will be greatest for the pattern within the superposition that is most like A. The use of superposition reduces the computational complexity from exponential growth to linear growth, thus making the problem tractable.

Even though MSC has a biological basis behind its mathematical framework, the work presented in [6] never proposed any kind of learning model that would explain how such an architecture would appear given a set of input examples. Work proposed in [92] and [57] formulated a learning algorithm for MSC such that a robot or visual model would be able to learn different layers and transformations of MSC, with a single view. But these learning algorithms never explained how the computations might be happening in a human brain using spiking neurons.

2.3.3 Low precision feedforward ANNs for transformation discovery

Both HMAX and MSC have proposed that the invariant transformations are stored as groups and once the input stimuli is presented, the model searches through all of the stored transformations and eventually selects the set of transformations that have the best match between input stimuli and the stored memory object. Even though both the algorithms have strong biological evidence backing up their claims, due to the structure of their framework, mapping these algorithms on the current generation of neural substrates is quite

challenging. To address this problem, this dissertation proposes a mathematical framework for a Hopfield neural-network based linear solver that discovers these transforms, and robustly matches objects with visual stimuli while revealing the corresponding affine transform that maps the input to its corresponding memory template.

Mapping such an algorithm to a low-precision hardware substrate requires the designer to carefully choose a strategy for quantizing inputs and weights. This paper examines the challenges that might come up when implementing a Hopfield network-based linear solver. In addition to affine transform-based object recognition and localization, which enables real-time object tracking, we also study two additional applications which also require finding the pseudo-inverse of a system of linear equations under real-time constraints: optical flow, which determines the direction and velocity of motion from successive frames of visual input, and inverse kinematics, which is used for motion planning of robotic arms. We report the computing resources that are required for these algorithms, and quantify how the errors or inefficiencies can be addressed to enable practical deployment of the Hopfield linear solver for these applications.

2.4 Hopfield Neural Network

In the context of recurrent neural networks, attractor states play a significant role. An attractor is a state towards which a dynamical system will converge over time. It is often the case that when a system can start many different initial points or states, eventually the system will stabilize towards a particular attractor state. The behavior of such attractor neural networks is often regarded as being similar to working memory in the neocortex,

allowing RNNs to retrieve partial or corrupted information present in the memory, at the same time controlling the motion of human body by maintaining it in a stable state.

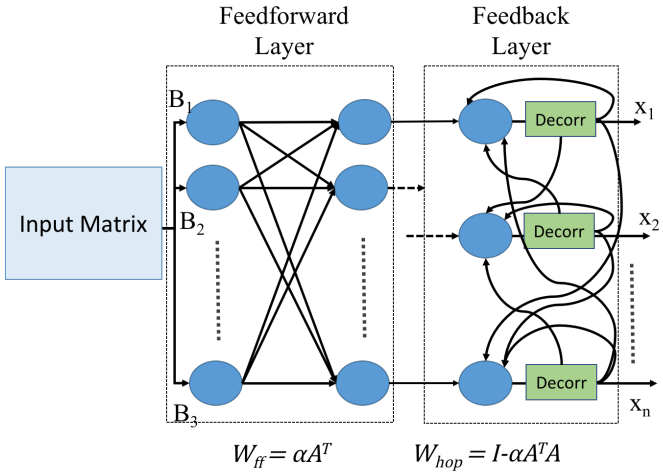


Figure 2.5: Neural network architecture of Hopfield Linear Solver

One particular implementation of RNNs is Hopfield neural networks (as shown in fig. 2.5) that was first proposed by J.J. Hopfield in 1982 [36]. They are a form of recurrent neural network that consists of a single layer where each node is an input to every other node in the network. Just like ANNs, theory of Hopfield neural networks is built around the concept of rate coded-neurons. The Hopfield network is commonly used for applications such as auto-association and optimization. We explain these two tasks more thoroughly as follows:

1. Auto-association: Hopfield neural networks or auto-associative memories have been used for storing and retrieving patterns based on partial matches and corrupted input data. For example one of the popular applications of Hopfield neural network has been in image restoration and correction [76]. Hopfield neural networks have demonstrated that even though an image might have partial information or they are

corrupted, these neural networks can reconstruct the original images using the partial information that was presented to it.

2. Optimization: Classically, the recurrent behavior of Hopfield neural networks have also shown the capabilities to tackle optimization based problems. Using the similarity between the cost function and energy function, the highly interconnected neurons can solve optimization problems. This property of Hopfield neural networks have been used to tackle applications such as the traveling salesman problem, linear dynamical systems and constrained optimization problems in robotics.

2.4.1 Calculating generalized matrix inverse with Hopfield neural network

Prior work have shown that Hopfield neural networks can be used to compute generalized matrix inverses and this property can be extended to a variety of signal processing and controls applications. For example, Zhang et.al. [106] have shown the similarities between computing generalized matrix inverse using back propagation neural networks and Hopfield neural networks. Research presented in [**Hopfield_linear_solver**] shows that Hopfield neural network can be used to compute Moore-Penrose pseudoinverse and can be applied in tasks such as estimating the "structured noise" component of a signal, adjust the parameters of an appropriate filter on-line and finally to control robotic arm through inverse kinematics.

Researchers have also considered using optical neural network hardware for implementing Hopfield neural network based matrix inversion [102] as a result bringing down

the computation time significantly. But all of these prior work have only considered implementing the algorithm on a floating point compute platform. In this dissertation we have proposed how spiking neural networks are able to compute generalized matrix inverse on random bitstream of data. This idea can also be extended to stochastic computing hardware substrates so that complex matrix operations can be done for robotics applications while consuming very low-power and at the same time using up simple logic gates.

2.5 Solving Linear Systems with a Hopfield Neural Network

Mathematically Hopfield neural networks can be written in form of iterative equations to solve a system of linear equations. This theory is explained in details as follows:

A linear equations solver is used to solve matrix equations of the form $AX = B$. More generally, to accommodate the case of infeasible systems, we obtain X from the following linear least squares problem:

$$\min_{X} \frac{1}{2}\|AX - B\|_F^2, \tag{2.3}$$

where A is an $M \times N$ matrix with $M \geqslant N$, B is a matrix of dimension $M \times P$, and $\|\cdot\|_F$ is the Frobenius norm. Note that the problem decomposes by columns of B, so we can write (2.3) equivalently as

$$\min_{[X]_{\cdot j}} \frac{1}{2}\|A[X]_{\cdot j} - B_{\cdot j}\|_2^2, \quad j = 1, 2, \ldots, P, \tag{2.4}$$

where $[X]_{\cdot j}$ denotes the jth column of the matrix X. By setting the gradient of (2.3) to zero,

we obtain the following "normal equations:"

$$A^T A X = A^T B. \tag{2.5}$$

This system can be solved by the following stationary iterative process:

$$X_{k+1} = X_k + \alpha(-A^T A X_k + A^T B) \tag{2.6}$$

$$= (I - \alpha A^T A) X_k + \alpha A^T B, \tag{2.7}$$

$$\text{where} \qquad X_0 := \alpha A^T B, \tag{2.8}$$

and α is a positive steplength. (This process can also be thought of as a steepest descent method applied to the optimization problems (2.4).) For convergence of this process, we require

$$0 < \alpha < \frac{2}{\lambda_{max}(A^T A)}, \tag{2.9}$$

where $\lambda_{max}(A^T A)$ denotes the maximum eigenvalue of $A^T A$. This condition ensures that all eigenvalues of $(I - \alpha A^T A)$ lie in the interval $(-1, 1]$. (If A is rank deficient, $A^T A$ is singular, so some eigenvalues of $(I - \alpha A^T A)$ will be 1 in this case.) See [8] for a proof of convergence.

We can map this process (2.6) to a recurrent Hopfield network cleanly by rewriting it as follows:

$$X_{k+1} = W_{hop} X_k + W_{ff} B, \quad k = 0, 1, 2, \ldots, \tag{2.10}$$

where

$$W_{\text{hop}} := \mathrm{I} - \alpha \mathrm{A}^{\mathsf{T}} \mathrm{A}, \tag{2.11a}$$

$$W_{\text{ff}} := \alpha A^{\mathsf{T}}. \tag{2.11b}$$

The Hopfield neural network architecture for implementing (2.10) is shown in Figure 2.5.

This elementary derivation shows that we can solve arbitrary systems of linear equations (which we refer to also as "matrix division") directly in a recurrent neural network by loading synaptic weight coefficients W_{ff} and W_{hop} derived from A and α into the neural network, and connecting the inputs b and recurrent outputs X_{k+1} appropriately. The weight matrix W_{ff} serves as the feedforward weight for the input matrix B, while W_{hop} serves as the weight for the recurrent part for the values X_k.

A major contribution of this dissertation is to have a mathematical formulation that will reason about how biological spiking neurons are able to perform complex tasks such as solving system of linear equation using Hopfield neural network models. In this dissertation we have proposed how the Hopfield neural network can be prototyped in two ways using the spiking neural networks computation scheme to perform least squares minimization operation. The first prototype is for applications in which Hopfield network weights could be hard-coded on a low-precision hardware, while the second prototype is for applications in which Hopfield network weights are encoded as dynamic spike trains. These implementation schemes have been discussed in chapter 5.

2.6 Summary

In this chapter we have introduced the importance of having a mathematical model that can explain how biological spiking neurons are able to solve inverse problems. The motivation of having affine transformations as a separate layer of abstraction has been proposed by prior literature that have looked at computations that happen in biological neural network. But none of these prior articles have ever presented how spiking neural network models are able to learn these affine transformations. Generally, these prior approaches have performed supervised training using a GPU or CPU and later mapped the "learned" neural network on a low-precision computation models. Additionally there have been prior work that has shown that ANNs are capable of solving inverse problems using a variant of recurrent neural network models called Hopfield neural networks. Hopfield neural network are capable of solving optimization problems without any prior offline training. The intention of this dissertation is to come up with a formulation on how such Hopfield neural network models can be deployed using spiking neural network computation scheme. Additionally this dissertation provides mathematical framework that would explain how biological neurons are able to solve a system of linear equations without any prior training for a variety of tasks such as object tracking, inverse kinematics, optical flow, etc.

3 NEUROMORPHIC HARDWARE

There has been growing interest among the researchers in hardware architecture community to look at computing platforms and calculation techniques that can be performed on streaming input bitstreams. Keeping in mind the challenges provided by the tasks that require the computations to be performed at ultra low-power hardware, architecture researchers have started exploring techniques that can do arithmetic calculations on streaming input bits using simple low-precision logic hardware. In this section we will look at two types of computing hardware that can perform tasks such as pattern recognition or matrix operations on extremely low-power hardware, viz, stochastic computing and neuromorphic computing. Aim of this section is to provide readers with the background detail about stochastic computing and neurosynaptic hardware details which we will be using extensively as part of this dissertation.

3.1 Stochastic Computing

Stochastic computing was first proposed by John von Neumann in 1952 [75] and its digital circuit realization was presented by B.R. Gaines in 1967 [30]. Stochastic computing is a compilation of rules and techniques that describe how to perform arithmetic computations on streams of random bits using simple logic gates. Stochastic computing has shown promise in domains such as machine learning [81], image processing [2], and decoding error correcting codes [31]. Since it is the time at which the bit is set that carries information about the value and not the position of this bit, a longer bit-stream would represent a more

accurate value. A single bit- flip in a long bit-stream will result in a small change in the value of the stochastic number represented. Whereas, if there is a bit-flip in floating point or fixed point representation then the magnitude of error can range from very small (if the bit flip is in lower-order bits) to very high (if the bit flip is in higher-order bits).

Researchers have looked at applications of stochastic computing in the context of signal processing [85], image processing [53], deep learning [81] [95] and fault tolerant computing [79]. If the objective is to perform filtering on the incoming analog signals [85] [56], with a stochastic computing scheme we can side step the need to perform expensive ADC-DAC conversion. Instead a simple sigma-delta modulator would perform the function of converting analog signals into bit streams and stochastic-arithmetic would perform computations on these streams of bit-pulses.

In this section we will look how computations are performed using stochastic computing and logic gate design to perform these computations.

3.1.1 Data Representation

In stochastic computing the numbers are represented using streams of random bits and this representation can be reconstructed to the corresponding value based on the frequency of occurrence of bit value 1. For example, fig. 3.1 shows a stochastic unipolar representation of the number 0.4. The bit-stream contains eight ones in a twenty-bit time tick, thus it represents $P(X = 1) = 8/20 = 0.4$.

Another way of representing numbers in stochastic computing is using bipolar representation scheme. In bipolar representation the input bitstream that might have frequency

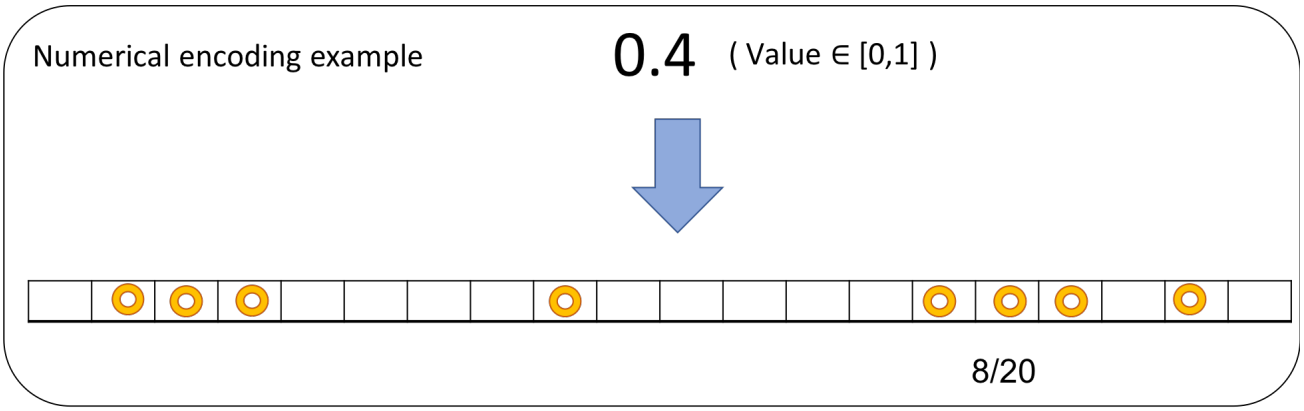


Figure 3.1: This image illustrates the stochastic computing encoding scheme. Example shown in this image illustrates how the number 0.4 will be represented as a stochastic bitstream over a period of 20 time ticks.

of bits in the range of $x \in [0, 1]$ can be converted to the range of $y \in [-1, 1]$ via the function $y = 2x - 1$. In this dissertation we will be considering only unipolar representations unless we state any other scheme specifically. The reason for selecting unipolar representation is explained thoroughly in section 3.3.

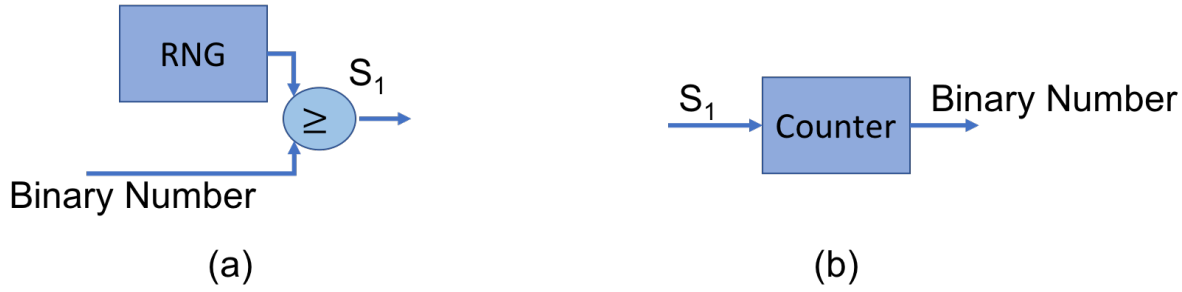


Figure 3.2: Digital logic design to encode and decode stochastic bitstreams. (a) Generates a stochastic bit stream from a binary number, (b) decodes an incoming bitstream into its binary number representation.

The stochastic bitstreams can be generated using a system that consists of a register which stores binary representation of the value, random number generator (like an LFSR) and a comparator. Fig. 3.2(a) shows the system to generate random bitstreams from a binary value. On the other hand, fig. 3.2(b) shows the system to convert a stochastic stream

into its corresponding binary value.

3.1.2 Arithmetic Computations

This subsection explains basic setup of logic gates to perform arithmetic computations and intuition behind how the calculations are carried out. We have limited our discussion to operations that are considered for the implementation of Hopfield neural network. Figures 3.3 and 3.4 show the setup of digital logic elements that perform various arithmetic operations.

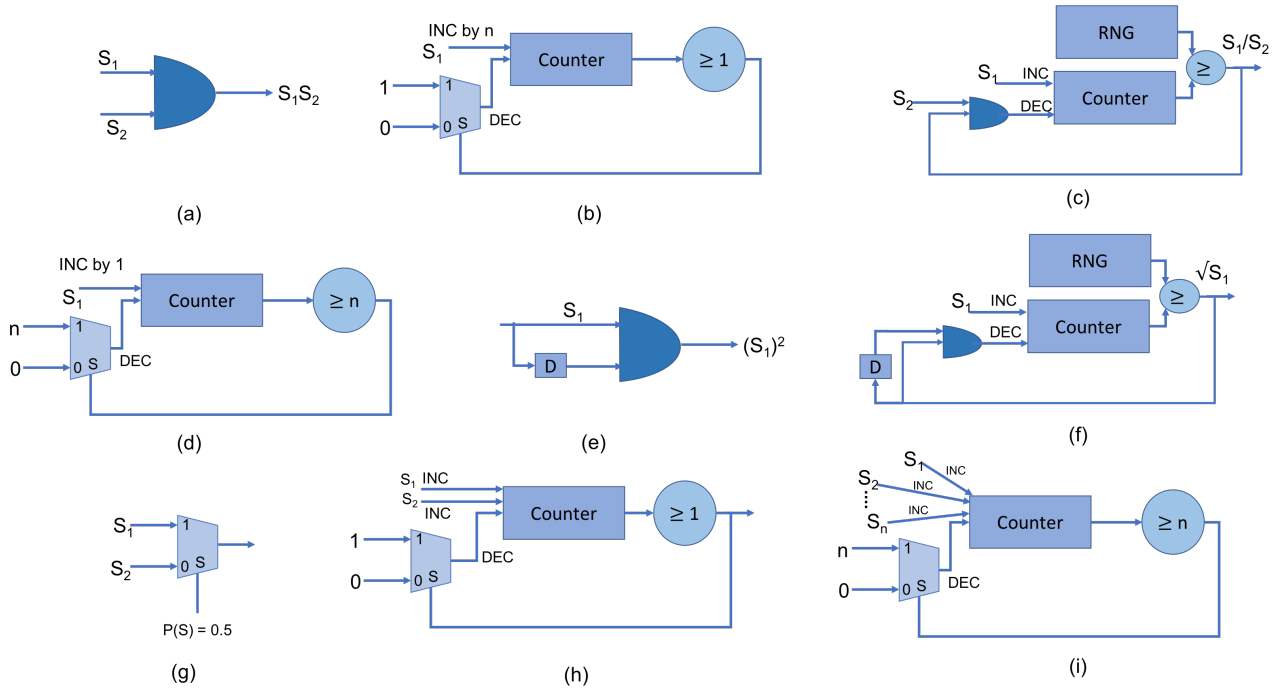


Figure 3.3: Digital logic design of different stochastic computing circuit elements. (a) AND gate based stochastic multiplier, (b) fixed gain multiplier, (c) stochastic division circuit, (d) fixed gain division circuit, (e) stochastic squarer (element D represents D-Flip Flop), (f) stochastic square-root unit (element D represents D-Flip Flop), (g) stochastic scaled adder, (h) stochastic lossless adder and (i) stochastic averaging unit

MULTIPLICATION: The major advantage of stochastic computing is that it can perform complex arithmetic computations on streaming input values with simple hardware.

For instance, values that have been encoded using a unipolar representation, can be multiplied with a single AND gate. Fig. 3.3(a) shows the setup for multiplying to stochastic bitstreams.

Eqn. 3.1 presents the mathematical intuition behind performing multiplication using an AND gate. Given two input values S_1 and S_2, these have been represented stochastically using independent random bit-streams. The AND gate can multiply these two bit streams based on the theory of expectation of random variables [30].

$$E[S_3] = E[S_1 S_2] = E[S_1]E[S_2] \tag{3.1}$$

FIXED GAIN MULTIPLICATION: For scenarios where fixed gain multiplications need to be performed on input bitstreams, these kind of computations can be done using a setup as shown in fig. 3.3(b). Each bit value in an input bitstream is multiplied with the number n and the multiplication result is later integrated to the counter register. If the counter is greater than or equal to 1, then an output bit is generated followed by decrement in counter value.

DIVISION: Fig. 3.3(c) shows the digital circuit setup for stochastic division, where S_1 and S_2 are the inputs to division unit and the goal of the circuit is to compute $\frac{S_1}{S_2}$. The logic design is same as the one that was presented by Gaines [30]. Mathematical intuition behind the digital circuit operation is that, rate at which the S_1 is coming into the system will be responsible for incrementing the counter. To negate this, the output will let the input bit S_2 pass and let it decrement the counter. Intuitively, the digital design is responsible for balancing the rate of S_1 with respect to S_2.

FIXED GAIN DIVISION: For the steps where input bitstreams have to be divided by a constant value, the digital circuit is much simpler, as shown in fig. 3.3(d). The incoming bit stream S_1, will store its value in a counter. If the value of counter reaches a threshold C (where C is a constant number), the counter will decrement its value by C and send the output bit to next set of compute logic. Because the output bits are generated by counter only after it receives C of them, this is equivalent to performing division operation by a constant term.

STOCHASTIC SQUARER: Just like multiplication, the squaring operation on an input bitstream can be performed using an AND gate. The difference between multiplication and the squarer logic is that there is an additional D Flip-Flop in the datapath, as shown in fig. 3.3(e). Since the generated input bitstream has an identical independent distribution (i.i.d.), if the generated bitstream is delayed by one-unit (using a single D-Flip Flop), statistically the delayed bitstream and non-delayed bitstream should be independent from one another. This is because an i.i.d. is only dependent on the input value and it is independent from the set of bits that were generated in previous time steps.

SQUARE ROOT: Fig. 3.3(f) shows the digital design implementation stochastic square root, where S_1 is the inputs to square-root unit and the goal of the circuit is to compute $\sqrt{S_1}$. The logic design is exactly same as the one presented in [30]. Mathematical intuition behind the digital circuit operation is that, rate at which the X_1 is coming into the system will be responsible for incrementing the counter, whereas, the counter will be decrement by a rate proportional to S_1^2.

ADDITION: Even though multiplication is easy in stochastic computing, addition is comparatively harder. Since the bitstreams represent values that are in the range of either

[0,1] (or [-1,1]), an addition of two independent bitstreams may result in a value that might be more than1, that is, the result may end up in the range of [0,2] (or [-2,2]). These design and compute issues can be addressed by using following two approaches:

1. Perform scaled addition using a multiplexer based design. Fig. 3.3(g), shows the setup of multiplexer based scaled addition of two input bitstreams. The select line of the multiplexer is a stochastic bitstream with a value of 0.5. Goal of this multiplexer line is to select either one of the two streams with a probability of 0.5 given to each stream. Given input streams that represent the value s_1 and s_2, the result of this scaled addition setup will be $\frac{s_1+s_2}{2}$.

2. Another technique to perform addition is by performing lossless addition on stochastic bitstreams, as depicted in fig. 3.3(h). This can be achieved by first scaling down the input values appropriately so that the user can guarantee that subsequent arithmetic operations will always keep the intermediate result in the range of [0,1]. In this dissertation we have chosen lossless addition technique for stochastic computing.

Scaled addition is appropriate in setup where we already know the exact number of additions that we will be performing and rescale the final result based on these predetermined number of additions. Whereas, for situations where number of addition operations may vary, for example, in iterative equations like eqn. 2.6, user would not know the number of times scaled addition was performed as a result it would be impossible to guess the value by which final result has to be rescaled to get the correct output value. Therefore, to circumvent this issue of estimating the number additions before implementation, we have selected the lossless addition scheme. For lossless addition, user would have to first reason

about the maximum and minimum values that intermediate results might take up and scale the input and output values based on the approximate range of intermediate values.

AVERAGING: The digital circuit setup shown in fig. 3.3(i) is meant to compute average of n bitstreams. Just like the lossless adder unit, the n input bitstreams integrate with the counter value. If the value of counter is greater than or equal to n, then an output bit gets generated from the unit and the value of counter gets decremented by n.

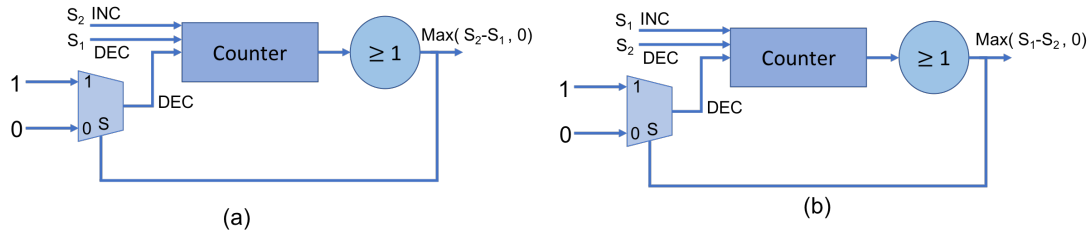


Figure 3.4: Digital logic design to perform subtraction operation between two input bitstream values. (a) computes $\max(S_1 - S_2, 0)$, and (b) computes $\max(S_2 - S_1, 0)$

SUBTRACTION: Similar to lossless addition, subtraction on stochastic bitstreams can be performed with a similar setup. But to perform subtraction, we would need two compute setups to handle results of different signs because unipolar representation does not have notion of sign. As a result, if the input values are in the range of [0,1], the subtraction results can be in the range of [-1,1]. Thus, one compute path will handle values that are in the range of [0,1], whereas, the other path will handle computations that are in the range of [-1,0]. For example, if we have two input values S_1 and S_2, their unipolar stochastic bitstream representation will lie in the range of [0,1], but unipolar stochastic representation of $(S_1 - S_2) \in [-1, 1]$. Fig. 3.4(a) will carry a positive value if $(S_2 > S_1)$ and fig. 3.4(b) will have a value 0; or fig. 3.4(b) will carry a positive value if $(S_1 > S_2)$ and fig. 3.4(a) will have a value 0; fig. 3.4(a) and fig. 3.4(b) will have a value 0 if $S_1 = S_2$.

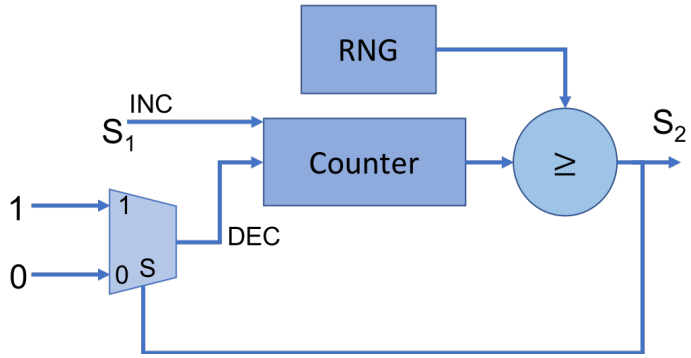


Figure 3.5: Digital design setup for implementing decorrelator operation.

DECORRELATOR: Since we will be implementing an iterative equation using stochastic computing scheme, our model will have recurrent connections (or feedback loops). To ensure that the iteration variable X_k in eqn. 2.10 is uncorrelated from the input bitstream, we pass it through a decorrelator. Function of this unit is to add additional randomness in the computation so that the multiplication between input bitstream and iteration variable bitstream will yield correct result. Fig. 3.5 shows the digital logic to implement a decorrelator. The input bitstream S_1 will increment the value of counter based on its bit value at that time tick. There is a separate random number generator (RNG) whose output will be compared with the counter value at every time tick. If the value of counter is greater than or equal to the output of RNG, the setup will output a single bit at that time tick, and in the same instant it will decrement the value of the counter; otherwise, the setup will output a zero bit at that time tick, and counter value will remain the same.

3.1.3 Pros and cons of stochastic computing

Although one of the major advantages of stochastic computing is that complex arithmetic operations could be performed on bitstreams using simple logic gates, some other advan-

tages for this computing scheme are as follows:

1. One major advantage of stochastic computing scheme is its fault tolerant computing nature. Since the value is represented with bit streams over a period of time, therefore, if there is a spurious bit-flip at a time tick, the representation error would be small.

2. Stochastic computing elements can also tolerate error that might occur due to clock skew [71]. Because of the additional delay that gets introduced due to skew, additional randomness gets added between two bitstreams. As a result, stochastic computing systems can operate using inexpensive locally generated clocks, instead of using expensive global clocks and handling the system-wide clock distribution.

3. Another major strength of stochastic computing is "progressive precision". If the calculations are performed over a long period of time, the value becomes more accurate, as a result, the output values also become more precise. Thus, accuracy of results can be improved significantly if the user operates the system for a longer duration

Even though operation on random bitstreams allow us to perform calculations on inexpensive low-power hardware systems, these types of operations also present the user with some challenges. These issues have been discussed as follows:

1. Stochastic computing systems are disadvantageous for applications where latency is primary concern. Although, precision of output values would improve significantly if the calculation duration is increased, the application would suffer from the issue of getting results after a long period of time. Therefore, for applications where

precision and accuracy is critical, stochastic computing is not yet a viable solution. This dissertation addresses this challenge using a population coding scheme in chapter 6).

2. To perform correct set of calculations, it is important to keep the bit streams uncorrelated especially for iterative algorithms. To handle feedback operations, users would have to either add additional LFSR units in the circuit or add delay units to feedback loop [15]. Additional LFSR would increase the cost of hardware consumption, whereas, delay units would slow down the computation time.

In spite of the challenges that are introduced with stochastic computing, these calculation techniques still seem promising especially because of the similarity between stochastic logic circuits and digital spiking neural network computing hardware. In the next section we will look at IBM TrueNorth neurosynaptic substrate, and we will also draw similarities between computing that happens in stochastic logic and digital spiking neurons.

3.2 IBM TrueNorth NeuroSynaptic System

In this dissertation we will focus on IBM NeuroSynaptic system for deploying least squares minimization algorithm (eqn. 2.3) on neuromorphic hardware substrate. The goal of selecting this hardware is to bridge the gap between mathematical theory that would explain how a human brain is able to solve for a system of linear equations and how are such computations performed using spiking neural network. Porting the proposed algorithms onto IBM TrueNorth helps us evaluate the benefits of using spiking neural network hardware to perform complex matrix operations when compared to using traditional architectures

such as an FPGA or an ARM CPU to perform these calculations.

3.2.1 TrueNorth Architecture

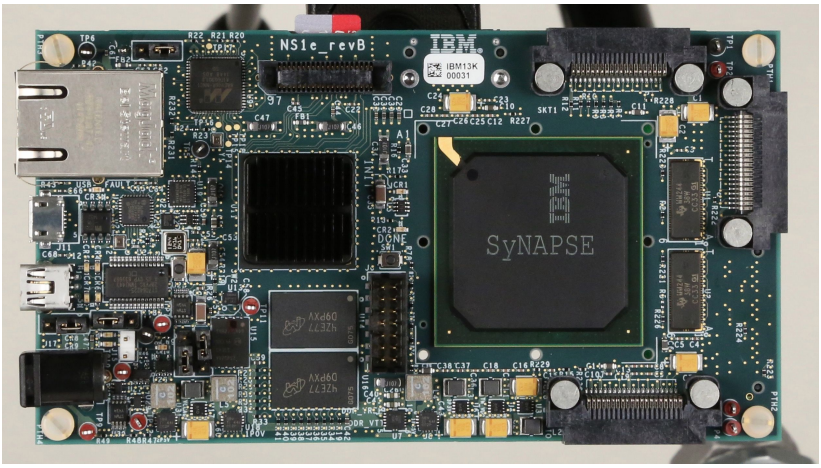


Figure 3.6: Single chip "ns1e" truenorth hardware. This image has been taken from [70]

The IBM Neurosynaptic System "TrueNorth" is a low power spiking neural network architecture that integrates 1 million programmable spiking neurons. The single chip system consists of 4096 cores where each core is composed of 256 axons (inputs) and 256 neurons (outputs), connected via a 256 x 256 crossbar of configurable synapses (about 65536 programmable synapse connections). The system operates at a rate of 1KHz, during which membrane potential processing and spike event routing occur asynchronously inside the chip. Spikes generated by a neuron can target any axon on the chip, with each neuron presenting over 20 individually programmable features (e.g. threshold, leak, and reset). Each neuron's equations and synaptic states are updated every millisecond, which is referred to as 1 tick. In this dissertation, we evaluate the proposed algorithm on IBM's ns1e single chip TrueNorth neurosynaptic hardware (as shown in fig. 3.6).

The following equations define the membrane potential (defined as $V_j(t)$) dynamics of

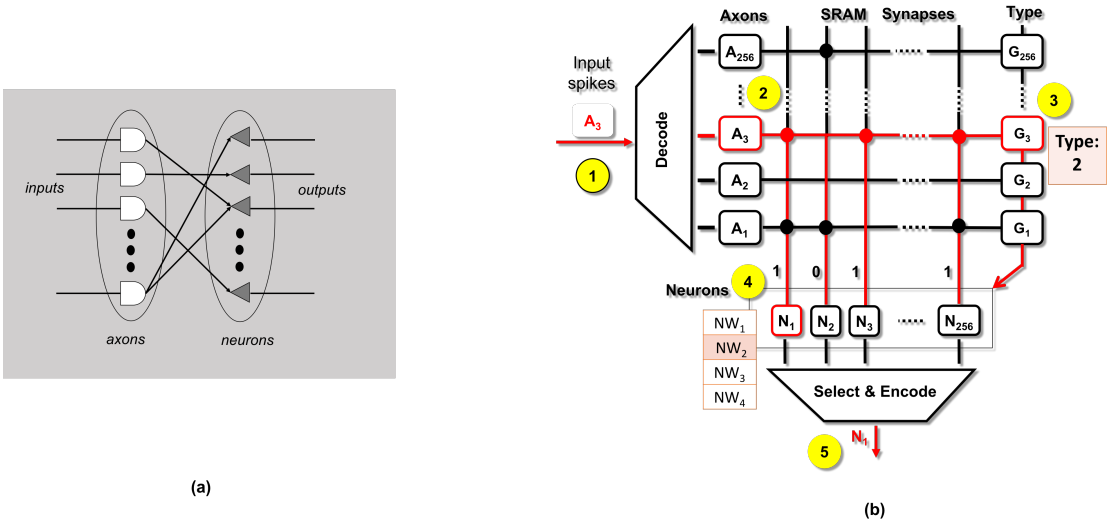


Figure 3.7: A higher level abstraction of Truenorth (from [73]) that shows, (a) Axons serve as inputs to the core. Each axon can be connected to 256 neurons. (b) Synaptic connections are programmable on a core with a weight value associated to each connection.

a j-th TrueNorth neuron at time t.

$$V_j(t) = V_j(t-1) + \sum_{i=0}^{255} A_i(t) w_{i,j} s_j^{G_i} \tag{3.2}$$

$$V_j(t) = V_j(t) + \lambda_j \tag{3.3}$$

$$\begin{aligned} &\text{if } V_j(t) \geqslant (\alpha_j + \eta_j) : \text{Spike and set } V_j(t) \leftarrow (\delta(\gamma_j) Vrst_j \\ &\qquad + \delta(\gamma_j - 1) V_j(t) - (\alpha_j + \eta_j) + \delta(\gamma_j - 2) V_j(t)) \\ &\text{else if } V_j(t) \leqslant -(\beta_j \kappa_j) V_j(t) \leftarrow -\beta_j \end{aligned} \tag{3.4}$$

Fig. 3.7, shows a high level architecture of TrueNorth's neurosynaptic core and equations 3.2, 3.3 and 3.4 formulate the basic set of operations that are performed on a TrueNorth's spiking neural model. Fig. 3.7(a) shows the idea that axons serve as inputs to the neurosy-

naptic core. Each one of these 256 axons is connected to each 256 neurons through a 256 x 256 configurable crossbar, as shown in fig. 3.7(b). Fig. 3.7(b) shows the step-by-step of computations of the neurosynaptic core. At time tick t the input spikes are first received by a decoder (as shown by ①) and are later sent to the i^{th} axon, $A_i(t)$ (marked by ②). Based on the value of binary term $w_{i,j}$ these input spikes are later redirected from i^{th} axon to j^{th} neuron (marked by ④). Next step marked by ③ represents the term $s_j^{G_i}$ which parameterizes the synaptic weight between axon i and neuron j, where G_i indicates which one of the four types was selected for i-th axon. In fig. 3.7(b) G_i has been set to 2 for i^{th} axon. As a result, any input spike that is received by the i^{th} axon will be multiplied by the second weight of any j^{th} neuron. Each neuron has configurable 9-bit signed integer weight, where most significant bit serves as the sign bit and the possible range of value that weights can take $\in [-255, 255]$.

Eqn. 3.3 shows the mathematical formulation of the Linear Leaky Integrate and Fire (LLIF) model of a TrueNorth neuron. As per eqn. 3.3, leak value λ_j gets integrated to the neuron's membrane potential $V_j(t)$ at time instant t itself. The nature of this integration is "linear", that is, TrueNorth does not support non-linear leak parameter such as exponential or transcendental neuron decay. The value of leak (λ) can be either a positive or a negative constant and it can take up any integer value in the range $\in [0, 255]$. The leak parameter can also be stochastic, that is, at every time tick t it can have any integer value in the range $\in [0, 2^{TM} - 1]$, where TM is the threshold value, which is specified by the user during design time and it can have an integer value in the range $\in [0, 8]$. In this dissertation we will not be considering the stochastic leak property of the TrueNorth neuron.

Lastly, equation 3.4 compares the updated membrane potential (after weight integration)

$V_j(t)$ with the threshold parameters α_j, η_j and $\beta_j \kappa_j$. The term α_j is an 18-bit integer parameter for positive spike threshold, η_j is also an 18-bit integer parameter which can assume any integer value in the range $\in [0, 2^{TM} - 1]$, where TM is an 18-bit positive spike threshold mask that can take integer value $\in [0, 18]$. Parameter β_j is meant for negative spike threshold, that is, to evaluate the condition whether neuron membrane potential $V_j(t) < \beta_j$ and κ_j parameter is meant to decide whether the membrane potential should saturate at the negative floor, that is, if the value of κ_j is set, the membrane potential should saturate at the negative spike threshold β_j and it should not go any lower than this value.

As per eqn. 3.4, if $V_j(t)$ is equal to or greater than the threshold $(\alpha_j + \eta_j)$, the neuron will spike and its membrane potential gets adjusted based on the selected reset mode (value of parameter γ_j). Following points explain how each one of the reset mode operates after the TrueNorth neuron j fires:

1. **Normal reset ($\gamma_j = 0$) :** In this reset mode, after the neuron fires, its potential will go back to the specified reset membrane potential, $Vres_j$, that was defined by the user during design time.

2. **Linear reset ($\gamma_j = 1$) :** In linear reset, after the neuron fires, its membrane potential is decreased linearly by the amount $(\alpha_j + \eta_j)$.

3. **No reset ($\gamma_j = 3$) :** This is the third reset mode that TrueNorth neurons can support. As per the mathematical formulation, this is the saturating reset mode, that is, even after the neuron fires, the membrane potential does not change and carries forward its membrane potential value even in the next time tick.

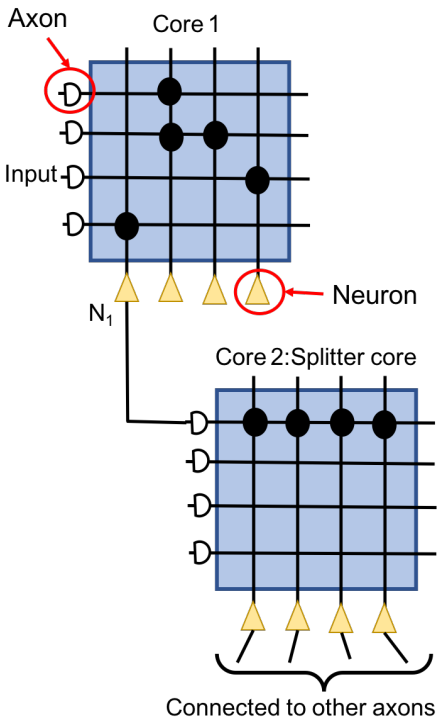


Figure 3.8: Shows a TN core being used as splitter

One of the limitations of the TrueNorth architecture is that neurons have a fan-out of one, that is, a single neuron can be either connected to only one axon or can be connected as an output port. If a neuron has to be connected to multiple axons, then we would require a set of neurons to act as "splitters" (as shown in fig. 3.8); these splitter neurons would not be performing any compute operations and their sole purpose would be to redirect the spikes received by the corresponding axon.

3.3 Mapping stochastic computing to TrueNorth

One of the major motivations for us to select IBM TrueNorth for evaluation is that digital spiking neurons have shown the ability to perform computations that are similar to stochastic computations. If the neuron parameters of TrueNorth are set appropriately, then we can

replicate the behavior of stochastic computing logic gates. For example, we can replicate an AND gate using a single TrueNorth neuron. If we have a two input TrueNorth neuron, the weights of both of these inputs can be set to 1, leak parameter to be -1 and have a no-reset mode, with this scheme a neuron can behave like an AND gate. In section 3.1.1 we had mentioned that the computations will be performed on unipolar stochastic representation scheme. This is because it is much easier to model an AND gate with a TrueNorth neuron instead of modeling an XNOR gate.

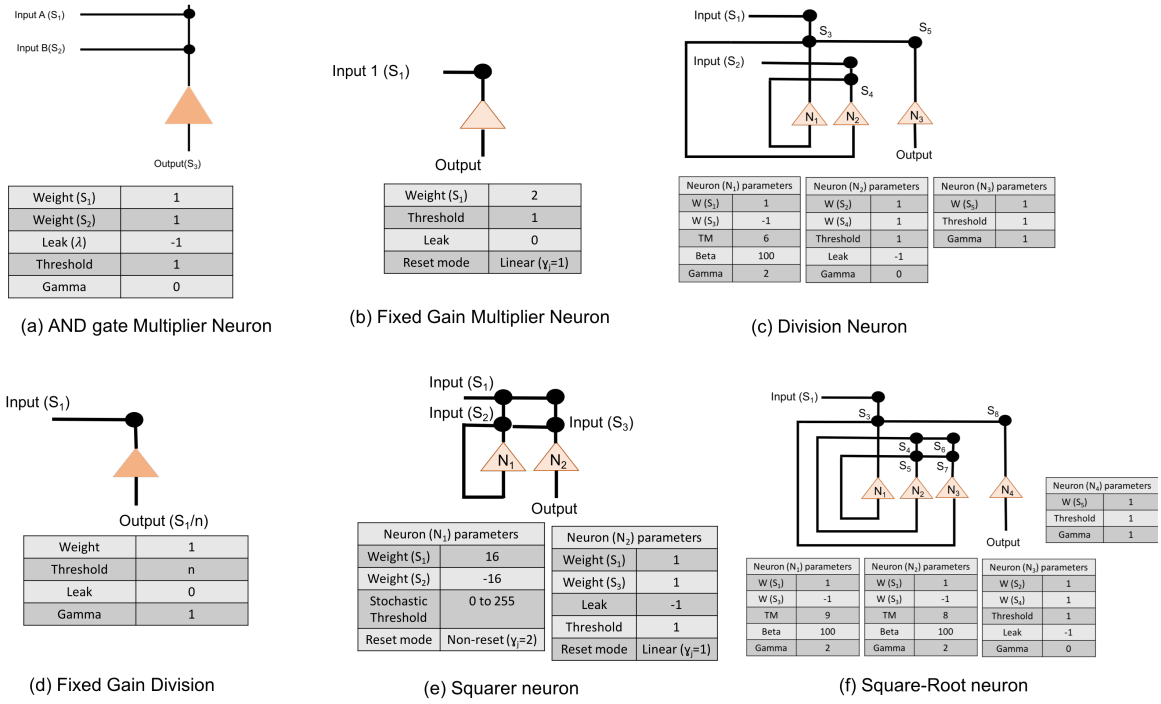


Figure 3.9: This image shows the stochastic computing arithmetic units that have been modeled using TrueNorth neurons (from [41]). Black circles in the image mean that there is connection present at that point in the crossbar setup. The compute elements refer to fig. 3.3(a)-(f)

The elements shown in fig. 3.9, show the neuron parameters and neuron connections that can perform stochastic computing operations using TrueNorth neurosynaptic system. Black circles in the image correspond to connections present at that point in the crossbar

setup. The elements in fig. 3.9 correspond to six stochastic computing calculation units of fig. 3.3(a)-(f), viz., unipolar bitstream multiplication using AND gate, fixed gain multiplication, stochastic division, fixed gain division, stochastic squarer and stochastic square root operation.

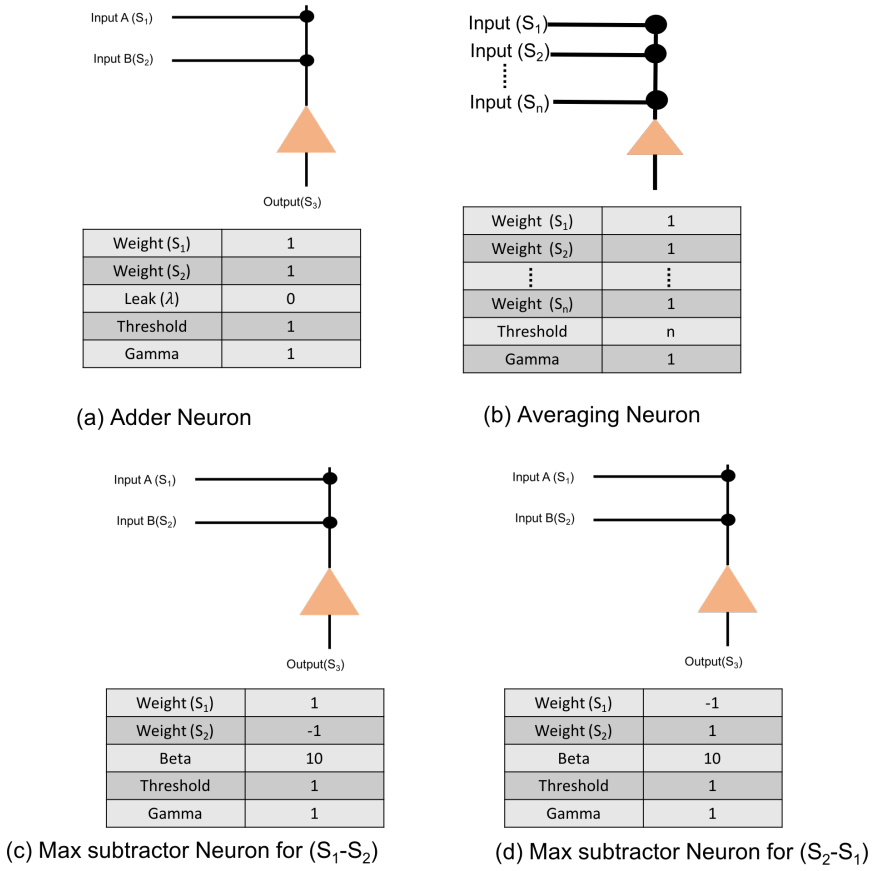
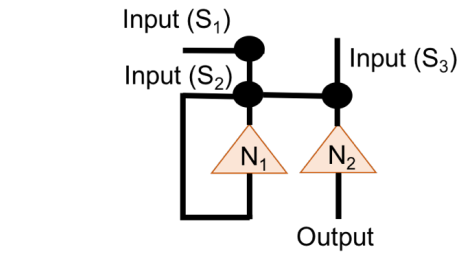


Figure 3.10: This image shows theTrueNorth neuron parameters and connections to perform lossless stochastic addition, averaging and subtraction operations (from [41]). The neuron connections and parameters replicate the behavior of stochastic elements shown fig. 3.3(h)-(i) and fig. 3.4(a)-(b)

Fig. 3.10 shows the neuron parameters and connections to perform computation tasks, viz., lossless adder, bitstream averaging (fig. 3.3(h) and (i) respectively) and stochastic subtraction (fig. 3.4(a) and (b)).

Truenorth neuron parameters and connections for implementing decorrelator is shown



Neuron (N_1) parameters	
Weight (S_1)	16
Weight (S_2)	-16
Stochastic Threshold	0 to 255
Reset mode	Non-reset (γ_j=2)

Neuron (N_2) parameters	
Weight (S_3)	1
Threshold	1
Reset mode	Linear (γ_j=1)

Decorrelator Neuron

Figure 3.11: TrueNorth neuron parameters and connections for implementing decorrelator (from [41]). The digital logic implementation of this function has been shown in fig. 3.5

in fig. 3.11. The decorrelator digital logic shown in fig. 3.5 can be implemented with a single neuron (N_1) of TrueNorth. Since TrueNorth neurons have a fan-out of 1, the second neuron N_2 is meant to forward the decorrelated spikes to other axons or output logic, similar to a splitter neuron, which is shown in fig. 3.8.

3.4 Summary

In this chapter we presented how arithmetic calculations happen using stochastic computing. Using the theory which has already been proposed for stochastic computing, we can perform complex set of computations on input bitstreams using very simple digital logic. In this chapter we also discussed about the architecture of IBM TrueNorth neurosynaptic system and basic parameters that control the computing behavior of TrueNorth's LLIF neurons. For this dissertation we selected TrueNorth as our neuromorphic hardware evaluation board because with the correct set of parameters TrueNorth neurons can behave as

stochastic computing elements. Therefore, with TrueNorth we can bridge the gap between proposed mathematical framework that we can use to solve a system of linear equations and also have a biological model to explain how such computations can be performed using spiking neurons. The arithmetic computing units introduced in this chapter serve as the basic building blocks for implementing the proposed Hopfield linear solver in chapter 5; analyzing the recurrent path for population coding in chapter 6 and implementing an adaptive scaling unit for correct operation of the linear solver in chapter 7. Finally, the accuracy analysis and hardware benefits of the proposed neuromorphic computing units is presented in chapter 9, where we have presented a thorough analysis for FPGAs and TrueNorth hardware .

4 RANGE ANALYSIS TO DETERMINE INPUT SCALING FACTOR

In chapter 3 we introduced the hardware designs for stochastic computing so that complex arithmetic calculations can be performed on random input bitstreams. In this chapter we will present the arithmetic challenges that come up when doing computations with random bitstreams. Specifically the challenge to guarantee that all of the intermediate computations will have value in the range $\in [0, 1]$. In this chapter we will discuss how such incorrect result might come up and mathematically how can we address these issues and still guarantee correct result.

4.1 Computations with Random Bitstreams : Challenges

Biologically-inspired neural network architectures, such as IBM TrueNorth, perform computations using spiking neurons or random input bitstreams. Input values are represented in a stochastic time-based coding, in which the probability of occurrence of a spike at a particular time tick is directly proportional to the input value. Since the computation values are represented as spike trains, designers are faced with two key issues in mapping algorithms to these spiking neural substrate.

1. Signed computations on spiking neural network substrates that have input values represented as rate based encoding must be performed by splitting all numbers (and intermediate results) into positive and negative parts.

2. Data representation is limited by maximum frequency of spikes. To represent different

values within a matrix, we need to scale all quantities so that no number exceeds this maximum frequency.

To repeat the argument presented in [91], Figure 4.1 illustrates the importance of selecting a correct scaling factor to represent multiple values in an input vector or an input matrix. Three values are given as inputs to TrueNorth (2, 4 and 5) and represented as spike trains. In Fig. 4.1 (a), all values have been scaled by the maximum-magnitude element 5, so all values can be represented within the available range of spiking rate. In Figure 4.1 (b), the inputs are scaled by a value smaller than the maximum magnitude, so saturation occurs: two elements (4 and 5) are represented by the same spike rate. Selecting the correct scaling factor is important when spike based arithmetic operations may produce results that are larger than any of the inputs. Fig. 4.1 (c) shows addition between two values represented as spike trains. Although both operands can be represented exactly with a scale factor of 4, the result of the addition is greater than the chosen scale factor, so the representation saturates and the result is inaccurate.

When implementing algorithms for random bitstreams, we must choose a scale factor that ensures that the intermediate computations never saturate. On the other hand, the scale factor should not be much larger than necessary, as this will result in loss of precision for the spike-train representations.

4.2 Scaling Factor

A Hopfield linear solver ([**Hopfield_linear_solver**] and [91]) can be used to compute the Moore-Penrose generalized matrix inverse based on the mathematical principles proposed

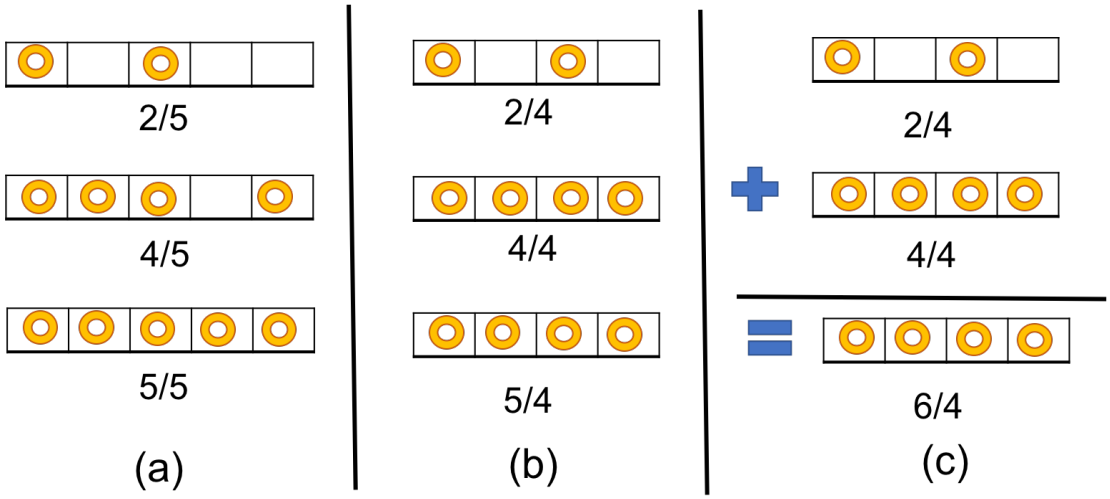


Figure 4.1: Example illustrating the importance of proper scaling for spike-based computation. (a) All three values of a vector are scaled properly. (b) Inappropriate scaling: Two values (4 and 5) are represented by the same spike rate. (c) Addition of two numbers that are scaled properly, but the scale factor is too small to allow proper storage of the result of the addition, leading to saturation.

by [8]. This section derives scaling factors that must be applied to the inputs to the system to guarantee that the vectors X_k that arise in the stationary iterative process (2.6) (equivalently, (2.10)) have no elements greater than 1 in absolute value, for all k. This requirement is achieved by means of a scaling factor η applied to the right-hand side B in (2.3).

For purposes of this section we define the max-norm of a matrix to be its largest element in absolute value, that is,

$$\|Y\|_{\max} := \max_{i,j} |[Y]_{ij}|. \tag{4.1}$$

(Note that when Y is a vector, the max-norm is the same as the ∞-norm.) Suppose that the (i,j) element of Y is the one that achieves the maximum norm. We have that

$$\|Y\|_2 \geqslant \frac{\|Ye_j\|_2}{\|e_j\|_2} \geqslant |[Y]_{ij}|, \quad \text{for any } i,$$

where e_j is the vector whose elements are all zero except for a 1 in position j. Thus

$$\|Y\|_2 \geqslant \|Y\|_{\max}. \tag{4.2}$$

We write the singular value decomposition of A as follows:

$$A = U\Sigma V^T, \tag{4.3}$$

where U is an $M \times N$ matrix with orthornormal columns, Σ is an $N \times N$ diagonal matrix with nonnegative diagonals, and V is an $N \times N$ orthogonal matrix. In fact, the diagonals of Σ are the singular values of A:

$$\Sigma = \operatorname{diag}(\sigma_1, \sigma_2, \ldots, \sigma_N), \tag{4.4}$$

where $\sigma_1 \geqslant \sigma_2 \geqslant \ldots \geqslant \sigma_N \geqslant 0$. We further use notation

$$\sigma_{\max} := \sigma_1, \quad \sigma_{\min} := \min_{\sigma_i > 0} \sigma_i. \tag{4.5}$$

In this notation, we have that $\lambda_{\max}(A^TA) = \sigma_{\max}^2$, so that condition (2.9) becomes

$$0 < \alpha < \frac{2}{\sigma_{\max}^2}. \tag{4.6}$$

Note too that $\|A\|_2 = \|A^T\|_2 = \sigma_{\max}$.

The following claim shows how we can scale the elements of B to ensure that $\|X_k\|_{\max} \leqslant 1$

for all k.

Claim 4.1. *For the iterative process defined by* (2.6)*, and supposing that condition* (4.6) *holds, we have that*

$$\|X_l\|_{\max} \leqslant \frac{2}{\sigma_{\min}}\sqrt{MN}\|B\|_{\max}, \quad \textit{for } l = 0, 1, 2, \dots . \tag{4.7}$$

Proof. By applying (2.6) recursively, we have for all l that

$$X_l = \alpha \sum_{k=0}^{l} (I - \alpha A^T A)^k A^T B. \tag{4.8}$$

From (4.3) we have that, $I - \alpha A^T A = V(I - \alpha\Sigma^2)V^T$, so that, $X_l = \alpha \sum_{k=0}^{l} V(I - \alpha\Sigma^2)^k \Sigma U^T B$. By multiplying both sides by V^T, we have that

$$V^T X_l = \alpha \left[\sum_{k=0}^{l} (I - \alpha\sigma_i^2)^k \sigma_i U_{\cdot i}^T B \right]_{i=1,2,\dots,N},$$

where $U_{\cdot i}$ is the ith column of U. By carrying out the summation, we have

$$[V^T X_l]_{i\cdot} = \alpha \frac{1 - (1 - \alpha\sigma_i^2)^{l+1}}{1 - (1 - \alpha\sigma_i^2)} \sigma_i U_{\cdot i}^T B, \quad i = 1, 2, \dots, N,$$

so that

$$[V^T X_l]_{i\cdot} = \begin{cases} 0 & \text{for } \sigma_i = 0; \\ \frac{1}{\sigma_i}[1 - (1 - \alpha\sigma_i^2)^{l+1}] U_{\cdot i}^T B & \text{for } \sigma_i > 0. \end{cases}$$

Since $1 - \alpha\sigma_i^2 \in (-1, 1)$ for all $i = 1, 2, \ldots, N$ with $\sigma_i > 0$, we have for such i that

$$\|[V^T X_l]_{i\cdot}\|_{\max} \leqslant \frac{2}{\sigma_i}\|U_{\cdot i}^T B\|_{\max} \leqslant \frac{2}{\sigma_i}\|U_{\cdot i}\|_1\|B\|_{\max} \leqslant \frac{2}{\sigma_{\min}}\sqrt{M}\|B\|_{\max} \tag{4.9}$$

where the last inequality follows from $\|U_{\cdot i}\|_2 = 1$, the standard inequality that relates $\|\cdot\|_2$ to $\|\cdot\|_1$, and the definition (4.5). By considering $V^T X_l$ one column at a time, we have

$$\|X_l\|_{\max} = \|VV^T X_l\|_{\max} \leqslant \|V\|_1\|V^T X_l\|_{\max} \leqslant \sqrt{N}\frac{2}{\sigma_{\min}}\sqrt{M}\|B\|_{\max},$$

and by applying (4.5), we obtain the result. □

An immediate corollary of this result is that if we replace B by $B/(\eta\|B\|_{\max})$ in (2.3), where

$$\eta := \frac{2}{\sigma_{\min}}\sqrt{MN}, \tag{4.10}$$

then the matrices X_l produced by the iterative process (2.10) have $\|X_l\|_{\max} \leqslant 1$ for all $l = 0, 1, 2, \ldots$. We note too that from the definition (2.11b) of W_{ff} and (4.10), we have by setting $B = I$ and $l = 0$ in Claim 4.1 that

$$\|W_{ff}\|/\eta \leqslant 1. \tag{4.11}$$

By applying this scaling, and writing the solution X of (2.3) as an infinite sum, we have

$$X = (\eta\|B\|_{\max})\sum_{k=0}^{\infty}(I - \alpha A^T A)^k \alpha A^T \frac{B}{\eta\|B\|_{\max}} = (\eta\|B\|_{\max})\sum_{k=0}^{\infty}(I - \alpha A^T A)^k \alpha A^T B_n, \tag{4.12}$$

where $B_n := B/(\eta \|B\|_{max})$. We use H_j to denote the jth scaled partial summation in (4.12), that is

$$H_{j+1} = \sum_{k=0}^{j} (W_{hop})^k W_{ff} \frac{B}{\eta \|B\|_{max}} \tag{4.13a}$$

$$= \sum_{k=0}^{j} (W_{hop})^k W_{ff} B_n \tag{4.13b}$$

$$= W_{hop} H_j + W_{ff} B_n. \tag{4.13c}$$

By setting $j = 0$ in (4.13c), we obtain

$$H_1 = W_{ff} B_n. \tag{4.14}$$

Note that H_l and X_l differ from each other only by the scaling factor $\eta\|B\|_{max}$, so we have from Claim 4.1 and eqn.(4.10) that

$$\|H_l\|_{max} = \frac{1}{\eta\|B\|_{max}} \|X_l\|_{max} \leqslant 1, \quad l = 1, 2, \ldots, \tag{4.15}$$

and thus

$$\|H_l\|_2 \leqslant \sqrt{NP}, \quad l = 1, 2, \ldots . \tag{4.16}$$

Algorithm 1 shows the steps to implement a Hopfield linear solver with scaled inputs. Goal of this algorithm is to find matrix X that can map matrix A to matrix B. Before we implement the iterative algorithm the input matrix B has to be scaled by parameter η (as derived in eqn. 4.10) and $\|B\|_{max}$ so that we can guarantee that intermediate computation

terms will always be in range $\in [0, 1]$.

Algorithm 1 Compute the transformation matrix that will map initial set of features (B) to the current set of observed input features (A). This recurrent algorithm was proposed in **[Hopfield_linear_solver]**

Input: Currently observed set of features (after proper scaling), $B_n = {}^{B}/_{\eta \|B\|_{max}}$
Output: Transformation matrix that will map matrix A to matrix B

procedure HopfieldSolver
  while $\delta \geqslant$ Mininum Error **do**
    $H_{j+1} = W_{ff} B_n + W_{hop} H_j$
    $\delta = \|H_{j+1} - H_j\|$
    $j = j + 1$
  end while
  $X = H_\infty \eta \|B\|_{max}$
end procedure

4.3 Summary

In this chapter we presented derived the scaling factor η (eqn. 4.10), for the input matrices, which will guarantee that the intermediate results would always be in range $\in [0, 1]$. The iterative nature of Hopfield neural networks involve multiple steps of matrix multiplications and additions. Because of these repeating arithmetic calculations the intermediate values may end up saturating. Therefore, it is important to maintain the correct range of computations otherwise the saturated result may get propagated to next set of iteration steps and would result in the algorithm to have an incorrect convergence value. Readers are advised to refer to chapter 9,section 9.1 to look at the analysis between the fraction of neurons that have saturated and the calculated scaling factor η for the input matrices.

5 IMPLEMENTATION

In chapter 3 we presented the digital logic design and TrueNorth neuron configurations to perform stochastic computing. Building upon the proposed arithmetic calculation units, in this chapter we will propose the architecture for implementing the proposed Hopfield neural network. The readers will understand how matrix multiplication is implemented with unipolar stochastic number representation (section 5.1), two ways to encode Hopfield neural network weights on TrueNorth hardware (section 5.2) and a complete setup to calculate generalized matrix using the recurrent neural network structure (section 5.3 and 5.4). The aim of this chapter is to present a complete picture to the readers about how the calculations are happening for streaming input random bitstreams.

5.1 Matrix multiplication with random bitstreams

In section 3.2 we had mentioned that the stochastic bitstreams will be represented using unipolar representation scheme, that is, the value will always be represented in range $\in [0, 1]$. If we are performing calculations with both positive and negative values, then the computations would have to be divided into two different planes, that is, separate set of computations for positive and negative values.

Fig. 5.1 shows five steps involved in performing matrix multiplication of matrices P and Q that have been encoded using unipolar representation scheme. The red color in the image is meant to represent flow of "positive" values whereas the green color is meant to represent flow of "negative" values. First step is to divide each input matrix P and Q into

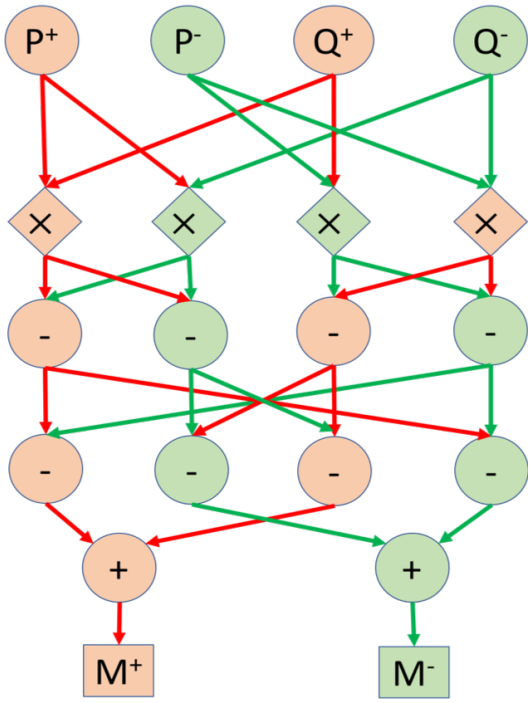


Figure 5.1: This figure illustrates how two matrices P and Q are multiplied using unipolar representation scheme for stochastic bitstreams.

positive (matrices P^+ and Q^+) and negative planes (matrices P^- and Q^-). Then each one of the four matrix is multiplied by the other three. Multiplication of two positive and negative terms will always have a positive result (hence, the red path), whereas, multiplication of a positive term with a negative term will always have a negative result (hence, the green path). In the next two steps we perform max-subtraction (detailed discussion in presented section 3.1.2). Finally we add the resulting matrices to get the final matrix multiplication answer.

The reason to perform max-subtraction in two steps is to ensure that if there is a non-zero value present in one of the signed planes, then the corresponding value in the other signed plane should be zero. For example, if m_{ij} is a non-zero value that is present in i^{th} row and j^{th} column of matrix M, and if m_{ij} is a positive value, then the (i, j) in matrix M^+

will be non-zero, whereas, (i, j) in matrix M^- will be zero. On the other hand, if m_{ij} is a negative value, then the (i, j) in matrix M^- will be non-zero, whereas, (i, j) in matrix M^+ will be zero.

5.2 Weight Assignment

We present two techniques to encode weights for the Hopfield neural network based linear solver. In the first subsection, we consider hardcoding the Hopfield neural network weights as TrueNorth neuron parameters. The extracted features of the image are used to compute weight matrices W_{hop} and W_{ff}, and these are converted further as TrueNorth neuron weight and threshold parameters. This scheme is suitable when the initial features do not change, as in 2-D image tracking. In the second subsection, we see how the computations are performed when weights are represented as spike trains (or represented as random bitstreams). This scheme is appropriate for scenarios in which the initial conditions may vary frequently, such as optical flow and inverse kinematics.

5.2.1 Hopfield neural network features encoded as TrueNorth weights and threshold

To perform matrix multiplication with weight matrices W_{ff} and W_{hop}, the floating point values of these two weight matrices are encoded as a ratio of TrueNorth weights to thresholds; see Algorithm 2. Here, a single synapse is used for each term in the dot product computation. In TrueNorth, each neuron can have up to four axon types as input, each of which can be assigned a unique synaptic weight in the range $[-255, 255]$. Figure 5.2(a)

Algorithm 2 Computes the weights and threshold values for performing dot product on TrueNorth

Input: Floating point values in the i^{th} row of weight matrices ($W_{i,\cdot}$)
Output: Assigned TrueNorth weight and threshold

1: **procedure** WeightThresholdAssignment
2: Threshold $= \text{Round}\left(\frac{255}{\max_i(|W_{i,\cdot}|)}\right)$
3: Weights $= \text{Round}\left(\frac{255}{\max_i(|W_{i,\cdot}|)} \times W_{i,\cdot}\right)$
4: **end procedure**

shows the synaptic connections in TrueNorth that implement dot product between the vector $[H_k(1,1); H_k(2,1); H_k(3,1)]$ and the columns of the 3×3 weight matrix W_{hop} (which can have either positive or negative values). Each of the three values in H_k have been assigned a different axon type, so that they are multiplied with a corresponding weight value to compute a dot product of the form $w_1 H_k(1,1) + w_2 H_k(2,1) + w_3 H_k(3,1)$. Each neuron i, has its reset mode set to linear reset ($\gamma_i = 1$) and rest of the parameters of the LLIF neuron have the default initial value.

Figure 5.2(a) presents the scenario where all weights in the Hopfield neural network (both W_{ff} and W_{hop}) can be encoded on a single TrueNorth neuron. Using all of the four axon types available in a single TrueNorth neuron, we can encode a Hopfield neural network that has four W_{ff} and W_{hop} neurons. For scenarios where the Hopfield neural network might have more than four neurons in either W_{ff} or W_{hop}, then the matrix multiplication would have to be divided as partial sums across multiple TrueNorth neurons. Figure 5.2(b) presents the setup for multiplying vector H_k by a single column of the matrix W_{hop}. Multiple neurons would be required to handle partial sums in matrix multiplication. Partial summation of matrix dot product are computed in neurons N_1 and N_2. Both of these neurons have linear reset mode, and their weight and threshold values are computed using

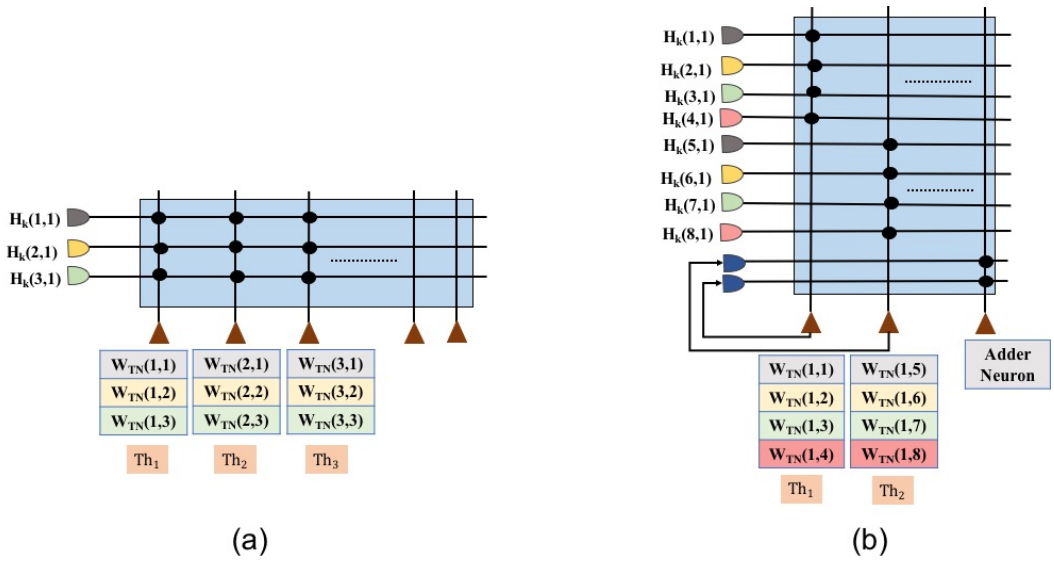


Figure 5.2: Synapse connection showing the dot product between first column of H_k and the weight matrix W_{hop}, and the corresponding threshold values for each neuron. (a) Shows the matrix dot product for the scenario in which W_{ff} and W_{hop} can be encoded using a single neuron. (b) Shows the matrix dot product for the scenario where W_{ff} and W_{hop} cannot be encoded using a single neuron. We would need multiple neurons to compute partial sums and later add them up together.

Algorithm 2. Once the partial sums have been computed, the results would go through a separate adder neuron where all of the intermediate sums would be computed. LLIF neuron parameters for an adder neuron is shown in fig. 3.10(a).

5.2.2 Hopfield neural network features using spiking inputs

For applications such as optical flow and inverse kinematics, where the initial input conditions may change dynamically, the hard-coding of TrueNorth weights discussed in previous subsection is not appropriate. We need an algorithm in which TrueNorth neurons can be used as arithmetic computation units and operate over spiking inputs. Chapter 3 showed that when the data is represented as stochastic rate-based coding, the theory of stochastic computing (as presented in the survey paper of [1]) shows that neurons can perform such

Algorithm 3 Computes weight matrices W_{ff} and W_{hop} using spiking inputs

Input: Spiking coding of the elements in the matrices $\frac{I}{2}$, $(\sqrt{\frac{\alpha}{2}})A^T$, $(\sqrt{\frac{\alpha}{2}})A$, $\frac{\alpha A^T}{\eta}$. Before giving these elements as input to Truenorth, separate the elements into positive and negative domains.

Output: Assigned TrueNorth weight and threshold

1: **procedure** ComputeWeightsUsingSpikes
2: $(\alpha A^T A)^+ = \max(\alpha A^T A, 0)$;
3: $(\alpha A^T A)^- = \max(-\alpha A^T A, 0)$;
4: $(\alpha A^T)^+ = \max(\alpha A^T, 0)$;
5: $(\alpha A^T)^- = \max(-\alpha A^T, 0)$;
6: $P_1 = \max\left(\frac{I}{2} - \frac{(\alpha A^T A)^+}{2}, 0\right)$;
7: $P_2 = \max\left(\frac{(\alpha A^T A)^+}{2} - \frac{I}{2}, 0\right)$;
8: $P_3 = \left(\frac{(\alpha A^T A)^-}{2}\right)$;
9: $W_{hop}^+ = 2(P_1 + P_3)$;
10: $W_{hop}^- = 2(P_2)$;
11: $W_{ff}^+ = (\alpha A^T)^+/\eta$;
12: $W_{ff}^- = (\alpha A^T)^-/\eta$;
13: **end procedure**

arithmetic operations as multiplication, addition, subtraction, and division. Algorithm 3 shows the computation scheme for representing the Hopfield neural network weight matrices W_{hop} and W_{ff} using spikes. Fig. 3.9(a) and 3.10(a) show the LLIF parameters of TrueNorth neurons that needs to be set to perform arithmetic operations such as multiplication and addition, Similarly, fig. 3.10(c) and (d) shows the LLIF parameters of TrueNorth neurons that needs to be set to perform subtraction.

Since the matrix computations for W_{hop} and W_{ff} will be done in the hardware itself, we need to reconstruct the iteration formula in such a way that every term that serves as an input to the hardware has magnitude less than 1, otherwise the computations might saturate and give us the wrong result. We rewrite (4.13a) as follows, to ensure that each

bracketed term has all its elements in the range $[-1, 1]$:

$$H_{j+1} = \sum_{k=0}^{j} \left(\frac{I}{2} - \left(\sqrt{\frac{\alpha}{2}} A^T \right) \left(\sqrt{\frac{\alpha}{2}} A \right) \right)^k 2^k \left(\frac{\alpha A^T}{\eta} \right) \left(\frac{B}{\|B\|_{max}} \right). \tag{5.1}$$

We state the formal claim as follows.

Claim 5.1. *All elements of the matrices* $\left(\sqrt{\frac{\alpha}{2}} A \right)$, $\left(\sqrt{\frac{\alpha}{2}} A^T \right)$, W_{hop}, *and* $\left(\frac{\alpha A^T}{\eta} \right)$ *lie in the interval* $[-1, 1]$. *That is, the max-norms* (4.1) *of these four matrices are all less than* 1.

Proof: Because of (4.2), it suffices to show that $\| \cdot \|_2 \leqslant 1$ for all four of the matrices in question.

For the first matrix, note from (4.6) that $\sqrt{\alpha/2} \leqslant 1/\sigma_{max} = 1/\|A\|_2$. Thus $\left\| \sqrt{\frac{\alpha}{2}} A \right\|_2 \leqslant 1$, as required. The proof for the second matrix is identical.

For the third matrix we note that W_{hop} is a square symmetric matrix with eigenvalues in the range $[-1, 1]$. Thus the eigenvalues of W_{hop}^2 will be in the range $[0, 1]$, so $\|W_{hop}\|_2 \leqslant 1$, as required.

For the fourth matrix, we have from (4.6), the definition of η in (4.10), and the fact that $\|A\|_2 = \sigma_{max}$ that

$$\left\| \frac{\alpha A^T}{\eta} \right\|_2 \leqslant \frac{2}{\sigma_{max}^2} \sigma_{max} \frac{\sigma_{min}}{2\sqrt{MN}} = \frac{\sigma_{min}}{\sigma_{max}\sqrt{MN}} \leqslant 1.$$

■

5.3 Datapath

5.3.1 TrueNorth implementation

The spiking neural substrates can operate only for values in the range $[0, 1]$. Thus, to perform computation on numbers that can be either positive or negative, the computations must be divided into two separate domains, one working with the positive parts of the matrices and one with the negative parts (as discussed in section 5.1). As discussed in chapter 3, we have selected unipolar data representation for stochastic computing implementation. We selected this representation scheme because TrueNorth does not have XNOR gates, as a result supporting bipolar multiplication would be complicated because we would have to setup neuron connections and parameters to replicate XNOR gate. As discussed in section 3.2, it is easier to implement an AND gate on TrueNorth, therefore, a SNN hardware setup to perform calculations with unipolar representation is easier when compared to a SNN hardware setup to perform calculations with bipolar representation. Algorithms 4 and 5 implement the formula (4.13b). Algorithm 4 performs the preprocessing step or the feedforward path of the neural network architecture, while algorithm 5 implements the recurrent part of the Hopfield architecture. The steps shown in Algorithms 4 and 5 ensure that the intermediate computation values never saturate. This is managed by performing subtraction of intermediate results followed by addition in the final step. Since the input values were normalized by the scaling factor, as shown in equation 4.10, the addition of partial sums would never saturate. We use the following definition of the positive and

negative parts of a matrix:

$$Y^+ := \max(Y, 0), \quad Y^- := \max(-Y, 0), \tag{5.2}$$

where the max-operation is applied component-wise. The proposed architecture ensures that nonzero elements in the positive-part matrices have zeros in the corresponding elements of the negative-part matrices, and vice versa. We do not have to scale the values in Algorithm 5 while computing matrices M, PS_1, and PS_2 because Claim 4.1 guarantees that no quantity will exceed 1, by choice of scale factor η. The max function used in Algorithms 3, 4, and 5 can be implemented with a LLIF (linear leaky integrated fire) neuron.

To implement these arithmetic operations we set the TrueNorth neuron parameters appropriately. A detailed description of individual neuron parameters and their behavior with respect to TrueNorth's spiking neurons can be found in [13]. Figure 3.10(a) shows the neuron parameters and connections for implementing an adder function that is required to compute variables such as M in Algorithm 5. Similarly, Figure 3.10(c) and (d) shows the neuron parameters and connections for max-subtractor neuron that is meant to compute variables such as PS_1, and PS_2 in Algorithm 5, or, variable B_s in Algorithm 4.

5.3.2 Computation with Spiking Weights

For applications in which matrix A might change dynamically, it is not possible to hard-code the weights on TrueNorth. Instead, we borrow concepts from stochastic computing ([30, 1]) to perform multiplication between input streams using a single neuron. In stochastic computing, if the inputs are represented as independent streams of bits, then the multi-

Algorithm 4 Computes the scaled value of input features B based on W_{ff} and the normalizing factor. These scaled values serve as the input for recurrent network

Input: Coordinates of the current input features B
Output: Scaled values of input features for the recurrent network. These values are divided among four domains

procedure PREPROCESSING
  if Weights are hard-coded on TrueNorth **then**
    $B_n = \text{Normalize}(B, \eta \|B\|_{max})$;
  else if Weights are given as spiking inputs **then**
    $B_n = \text{Normalize}(B, \|B\|_{max})$;
  end if
  $T^{\langle +,+\rangle} = \max(W_{ff}^{+} B_n^{+} - W_{ff}^{+} B_n^{-}, 0)$;
  $T^{\langle +,-\rangle} = \max(W_{ff}^{+} B_n^{-} - W_{ff}^{+} B_n^{+}, 0)$;
  $T^{\langle -,-\rangle} = \max(W_{ff}^{-} B_n^{-} - W_{ff}^{-} B_n^{+}, 0)$;
  $T^{\langle -,+\rangle} = \max(W_{ff}^{-} B_n^{+} - W_{ff}^{-} B_n^{-}, 0)$;
  $B_s^{\langle +,+\rangle} = \max(T^{\langle +,+\rangle} - T^{\langle -,+\rangle}, 0)$;
  $B_s^{\langle -,+\rangle} = \max(T^{\langle -,+\rangle} - T^{\langle +,+\rangle}, 0)$;
  $B_s^{\langle -,-\rangle} = \max(T^{\langle -,-\rangle} - T^{\langle +,-\rangle}, 0)$;
  $B_s^{\langle +,-\rangle} = \max(T^{\langle +,-\rangle} - T^{\langle -,-\rangle}, 0)$;
end procedure

plication between these two values can be implemented with just one AND gate. In our implementation, the values are represented as stochastically rate coded spikes, similar to bit streams mentioned earlier. The AND gate can be modeled with an LLIF neuron as shown in Figure 3.9(a).

5.3.3 Importance of decorrelators in recurrent path

Since the computation involves sending the values through a recurrent path, it is crucial to maintain independence of spike occurrence (occurence of bit value 1) between the inputs from feedforward path and inputs coming in from recurrent path. Therefore, the inputs that are fed back need to be passed through a decorrelator. Fig. 3.11 shows the TrueNorth setup that adds the decorrelation in the recurrent path. On the other hand,

Algorithm 5 Solve for system of linear equations defined as $AX = B$. Computations are divided into negative and positive parts

Input: Matrices A and B that have been divided into positive and negative domains
Output: Solution for the system of linear equation, matrix X

procedure HopfieldSolver_Split
  while $\delta \geqslant$ Minimum Error **do**
    $M^{\langle +,+\rangle} = (B_s^{\langle +,+\rangle}) + (W_{hop}^{+} H_k^{+})$;
    $M^{\langle +,-\rangle} = (B_s^{\langle +,-\rangle}) + (W_{hop}^{+} H_k^{-})$;
    $M^{\langle -,-\rangle} = (B_s^{\langle -,-\rangle}) + (W_{hop}^{-} H_k^{-})$;
    $M^{\langle -,+\rangle} = (B_s^{\langle -,+\rangle}) + (W_{hop}^{-} H_k^{+})$;
    $PS_1^{\langle +,+\rangle} = \max(M^{\langle +,+\rangle} - M^{\langle +,-\rangle}, 0)$;
    $PS_1^{\langle +,-\rangle} = \max(M^{\langle +,-\rangle} - M^{\langle +,+\rangle}, 0)$;
    $PS_1^{\langle -,-\rangle} = \max(M^{\langle -,-\rangle} - M^{\langle -,+\rangle}, 0)$;
    $PS_1^{\langle -,+\rangle} = \max(M^{\langle -,+\rangle} - M^{\langle -,-\rangle}, 0)$;
    $PS_2^{\langle +,+\rangle} = \max(PS_1^{\langle +,+\rangle} - PS_1^{\langle -,+\rangle}, 0)$;
    $PS_2^{\langle -,+\rangle} = \max(PS_1^{\langle -,+\rangle} - PS_1^{\langle +,+\rangle}, 0)$;
    $PS_2^{\langle -,-\rangle} = \max(PS_1^{\langle -,-\rangle} - PS_1^{\langle +,-\rangle}, 0)$;
    $PS_2^{\langle +,-\rangle} = \max(PS_1^{\langle +,-\rangle} - PS_1^{\langle -,-\rangle}, 0)$;
    $\tilde{H}_{k+1}^{+} = (PS_2^{\langle +,+\rangle} + PS_2^{\langle -,-\rangle})$;
    $\tilde{H}_{k+1}^{-} = (PS_2^{\langle +,-\rangle} + PS_2^{\langle -,+\rangle})$;
    if Weights are hard-coded on TrueNorth **then**
      $H_{k+1}^{+} = \tilde{H}_{k+1}^{+}$;
      $H_{k+1}^{-} = \tilde{H}_{k+1}^{-}$;
    else if Weights are given as spiking inputs **then**
      $H_{k+1}^{+} = \text{Decorrelate}(\tilde{H}_{k+1}^{+})$;
      $H_{k+1}^{-} = \text{Decorrelate}(\tilde{H}_{k+1}^{-})$;
    end if
    $\delta = \left\| (H_{k+1}^{+} - H_{k+1}^{-}) - (H_k^{+} - H_k^{-}) \right\|$;
    $k = k + 1$;
  end while
  $X_k = \text{Rescale}(H_k, \eta \left\| B \right\|_{max})$;
end procedure

fig 3.5 shows the digital logic setup which introduces decorrelation in the recurrent path. Sections 3.1.2 and 3.2, we explained how the decorrelator units operate. Figure. 5.3 shows two different setups of simple Hopfield neural network architecture. Fig. 5.3a the recurrent

paths in Hopfield neural network do not have decorrelators, whereas in fig. 5.3b there are decorrelators (marked with green boxes) present in the recurrent path.

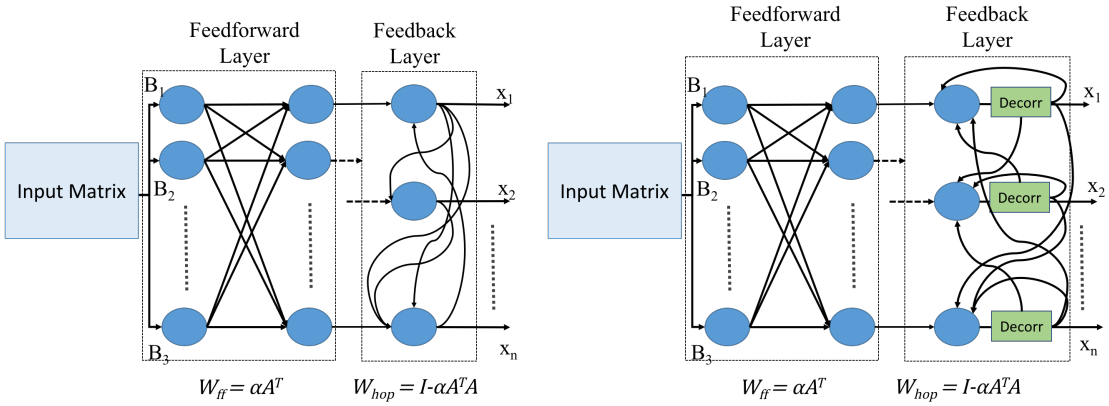


Figure 5.3: Two different setups for Hopfield neural network. In fig. 5.3a the recurrent paths in Hopfield neural network do not have decorrelators, whereas in fig. 5.3b there are decorrelators (marked with green boxes) present in the recurrent path.

The decorrelator (presented in [15]) preserves the spiking rate (frequency of bitstream) of the input signal, but makes the occurrence of spikes (bits in random bitstream) independent of the randomly generated feedforward values. To evaluate the presence of decorrelators for the proposed linear solver, we generated 25 different input matrices each with different matrix size. The smallest matrix dimension was 2 and the largest dimension was 10. Each element in the generated matrix was randomly selected in the range $\in [-1, 1]$. The loss was calculated based on the formula shown in equation 5.3, where X_{actual} refers to the output matrix computed using stochastic linear solver and $X_{expected}$ is the output matrix computed using the proposed iterative equation 2.6 that had 64-bit double precision compute units when simulated in MATLAB. The experiment was repeated 25 times and we report average loss for all 25 experiments in the results.

$$\text{Loss} = \|X_{actual} - X_{expected}\|_F \tag{5.3}$$

Fig. 5.4 shows the change in loss over time due to the absence or presence of decorrelators in recurrent path of linear solver. It can be observed in fig. 5.4, for the scenario when we did not have decorrelators present in the recurrent path (shown by red plot, corresponding setup shown in fig. 5.3a), the loss eventually started to increase over time. This is because as the number of ticks increased, there were correlations that started appearing between the input values and recurrent path spikes (or bitstream). As a result, the two multiplication bitstreams were no longer independent. As discussed in chapter 3, the AND gate based multiplication works in stochastic computing only if two input bitstreams are independent from each other. Having an independent decorrelator unit in the recurrent path ensures that there is additional randomness that gets introduced in the recurrent path values. As result the Hopfield neural network that had decorrelator decorrelators present in the recurrent path (shown by blue plot, corresponding setup shown in fig. 5.3b), its loss decreased over time.

5.4 Computing α on-chip using stochastic computing

As shown in eqn 2.9, the term α will guarantee that the proposed iterative equation 2.6 is guaranteed to converge. In this section we will discuss the steps to calculate the learning rate α using input random bitstreams.

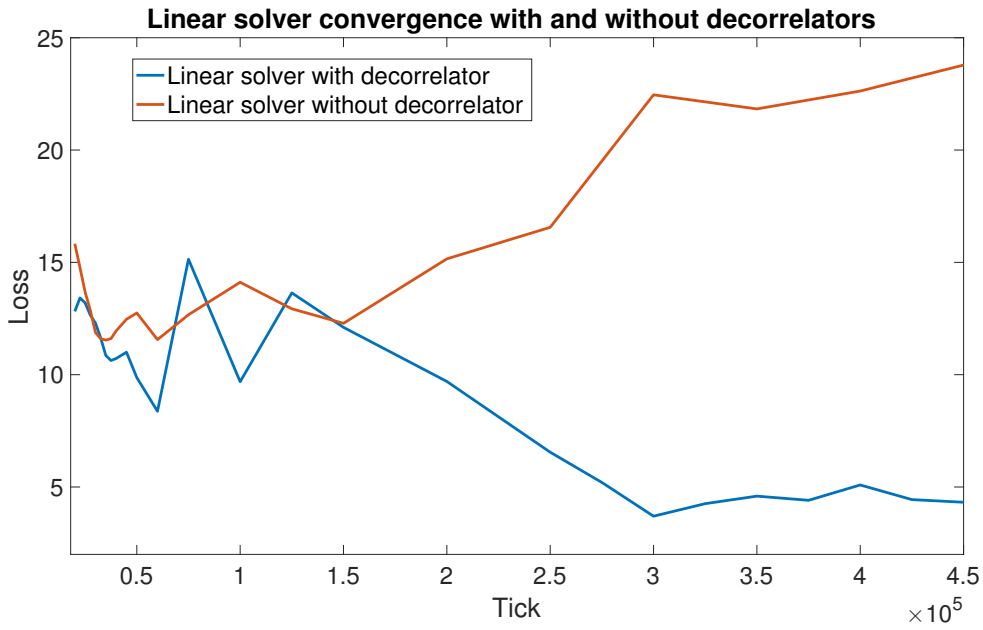


Figure 5.4: Variation is loss over time for linear solver with and without decorrelators in the recurrent path.

$$\tilde{\alpha} = \frac{c}{trace(A^T A)}, \tag{5.4}$$

where, c is a constant term which satisfies the condition,$c \in (0, 2)$

Based on the condition derived in eqn. 2.9, we can guarantee that $\tilde{\alpha}$ computed using the eqn. 5.4 will always converge. This statement is based on the argument that Frobenius norm of a matrix (denoted as $\|.\|_F$) is always greater than the 2-norm of the same matrix (denoted as $\|.\|_2$) [64]; that is, $\|A^T A\|_F \geqslant \|A^T A\|_2$. Computationally it is much easier to calculate $trace$ (or Frobenius norm) of a square matrix instead of 2-norm square matrix [**Hopfield_linear_solver**]. Calculating the 2-norm of $A^T A$ requires the designer to design the datapath for implementing power iteration [78] or Rayleigh quotient iteration [80]. Whereas, $trace$ of $A^T A$ can be easily calculated by squaring each term of the

matrix and adding them up. Fig. 5.5 shows the mathematical intuition behind element wise operations that happen when calculating $trace(A^TA)$.

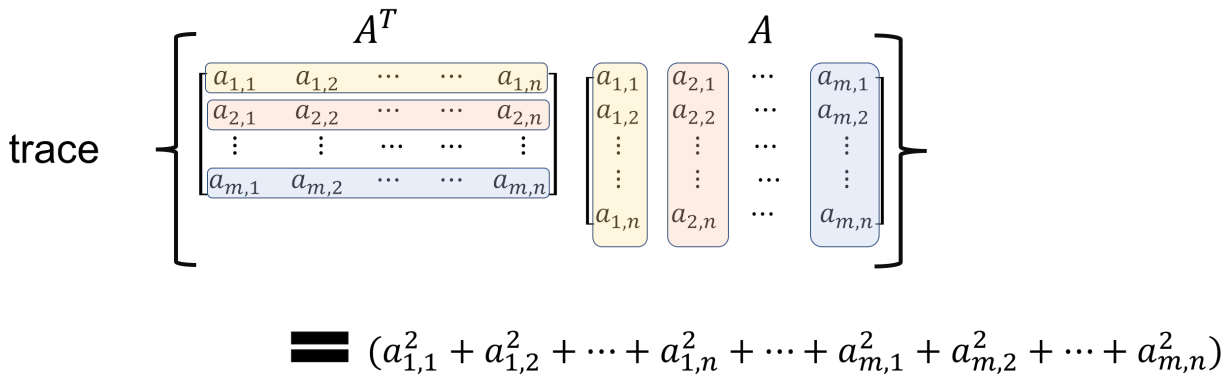


Figure 5.5: Element-wise matrix calculations to compute trace of A^TA

Building upon the concepts presented in eqn. 5.4 and fig. 5.5, the term $\tilde{\alpha}$ can be calculated using stochastic bitstreams with the setup shown in fig. 5.6.

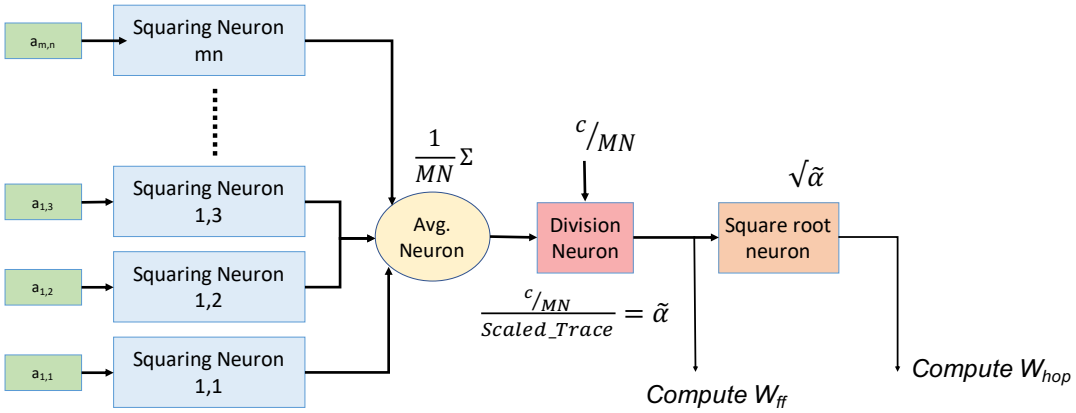


Figure 5.6: Setup for computing $\tilde{\alpha}$ on-chip on stochastic bitstreams. The term $c \in (0, 2)$ to guarantee that iterative eqn. 2.6 will converge.

Let us assume input matrix A in fig. 5.6, has a dimension of MxN. The following steps tell us how $\tilde{\alpha}$ is calculated using stochastic bitstreams.

1. Calculate $trace(A^TA)$. As discussed earlier this is achieved by squaring every element of matrix A. In fig. 5.6 a_{ij} represents an element present in i^{th} row and j^{th} column

of matrix A. After this we calculate an averaged summation of the squared values. Because $a_{ij} \in [0,1]$, then $\sum_{i=1}^{M}\sum_{j=1}^{N} a_{ij} \leqslant MN$. To make sure every intermediate value falls in the specified range of [0,1], we average the summation result by a factor of MN. At the end of first step, we have calculated $\frac{trace(A^TA)}{MN}$

2. Divide the constant input bitstream $\frac{c}{MN}$ (where c is a constant $\in (0,2)$) with $\frac{trace(A^TA)}{MN}$. After the second step we get the value of $\tilde{\alpha}$. The computed $\tilde{\alpha}$ can be multiplied with every element of input matrix A^T to get W_{ff}

3. Calculate $\sqrt{\tilde{\alpha}}$ which can be implemented using the stochastic square root compute unit (refer to fig. 3.3(f) and fig. 3.9(h) for the implementation details regarding stochastic computing and TrueNorth setup to calculate square-root operation.). The computed $\sqrt{\tilde{\alpha}}$ can be used for calculating W_{hop} for the recurrent path weights.

5.5 Summary

This section proposed the datapath design for implementing Hopfield neural network based linear solver using the stochastic arithmetic computing units that were presented in chapter 3. There have been considerable number of literature that discuss about implementing iterative functions using stochastic computing such as CORDIC algorithm [39], singular value decomposition [100], long short term memory [90]; but this is the first work where we discussed about implementing an iterative algorithm to compute generalized matrix inverse using random bitstreams. As discussed in chapter 2, the intention of this proposal was to bridge the gap between the mathematical framework that reason about how neurons

are able to compute generalized matrix inverse with Hopfield neural networks, and how such a recurrent neural network model can be implemented using spiking neurons. The proposed datapath can also be extended to stochastic computing hardware platforms such as FPGAs, or low-power ASICs.

6 POPULATION CODING

In section 3.1.3 we discussed about the advantages and disadvantages of performing stochastic computing. One of the challenges that we face with stochastic computing is the issue of latency [35]. In this chapter we will discuss about how latency of computations can be reduced by taking inspiration from biology called population coding scheme. Readers will first learn about the significance of population coding from the perspective of neuroscience and how these concepts get translated to stochastic computing. Later we will also discuss how to address the hardware requirements that come up due to presence of decorrelators in linear solver architectures, especially in the proposed population coding scheme where each linear solver is meant to operate on different bitstream. Finally, we will evaluate the population coding scheme against a minimum error selection technique, an approach which takes inspiration from stochastic gradient descent [49]. In this minimum error selection technique we have parallel linear solver units and each unit has different α or Hopfield neural network weights.

6.1 Neural coding

There are three ways in which information gets encoded in biological neurons [13], viz., (i) rate coding, (ii) population coding, and (iii) temporal coding. Fig. 6.1 shows how a value say 0.4 can be encoded using one of the encoding techniques. Following points describe each one of the encoding techniques:

- **Rate coding** This type of encoding scheme represents frequency of occurrence of

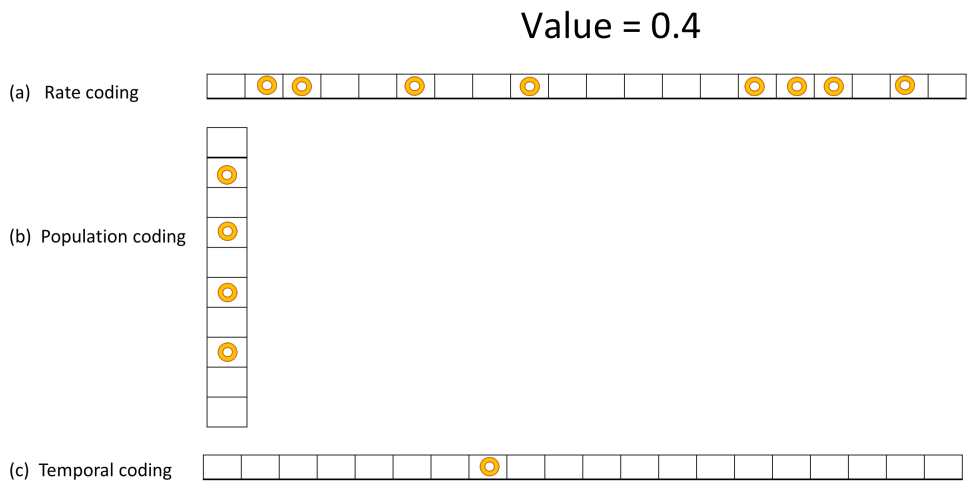


Figure 6.1: Three different neural coding techniques to encode information for computations in biological neurons. This figure shows a value like 0.4 can be encoded with three different neural coding techniques. (a) Rate coding. (b) Population coding. (c) Temporal coding

spikes (or bits) over a period of time. For instance, if the user wants to encode a value $\in [0, 1]$, then $\frac{\text{\# of spikes in t clock ticks}}{\text{tclockticks}} = value$. Fig. 6.1(a), a value of 0.4 has been encoded over a period of 20 time ticks. This 20 time tick window needs to have 8 spikes to represent the value 0.4.

- **Population coding** The values are encoded based on number of neurons that fire in a single time instance. Unlike rate coding scheme where the value is represented over a time period, in population coding scheme the values get represented in space. The encoding scheme can be formulated as $\frac{\text{\# of neurons firing at time t}}{\text{totalnumberofneurons}} = value$. As per the example shown in fig. 6.1(b) a value of 0.4 has been encoded in space by having any four neurons (out of ten available neurons) start firing at that time instant.

- **Temporal coding** In temporal coding, the time at which a spiking activity occurs carries the information. For instance, to represent a value, say 1, a spike should occur at the last time tick of a period t. Similarly, having a spike occur at the middle of the

time period t can represent a value say 0.5. As presented in fig. 6.1(c), a value say 0.4 can be represented over a period of 20 time ticks by having a spike occur at 8^{th} time slot.

6.1.1 Population coding

Population coding formulates how a group of neuron body will respond to external stimulus. As explained in [63], in tasks such as motion planning, the information is encoded using grid cells. These group of neurons that form a grid, fire with a certain distinct pattern to navigate a biological body through maze or certain path. Experimental studies have shown that this type of encoding scheme is common in sensory and motor areas of the brain [65]. Also one of the most interesting discoveries show that population coding is much faster than rate, and with population coding scheme neurons can respond much faster to external stimuli, almost instantaneously [40]. This finding motivated us to understand how can we take the advantage of population coding, a scheme which motivated nicely by neuroscience, and apply it to our mathematical framework of Hopfield neural network based linear solver. As a part of this dissertation, we have proposed a parallel computation scheme in which we have divided our computations in both time and space, and later empirically show that this parallel computing scheme (motivated from population coding) reduces the latency of calculations significantly.

6.2 A population coding based Hopfield linear solver

Figure 6.2 shows a high-level idea for a population coding based architecture that divides the computations into temporal and spatial domain. The input values and partial summation H_k of matrices are fed to n different feedforward computation units. To ensure that each feedforward unit receives the same value with different stochastic bitstream, input values would have to go through a decorrelator, and every linear solver has its own decorrelator implementation. Similarly to ensure that each recurrent path value is sufficiently independent from the input values, there are decorrelators addd in the recurrent path of the linear solvers. The output of all of the linear solvers is collected and averaged across space and time.

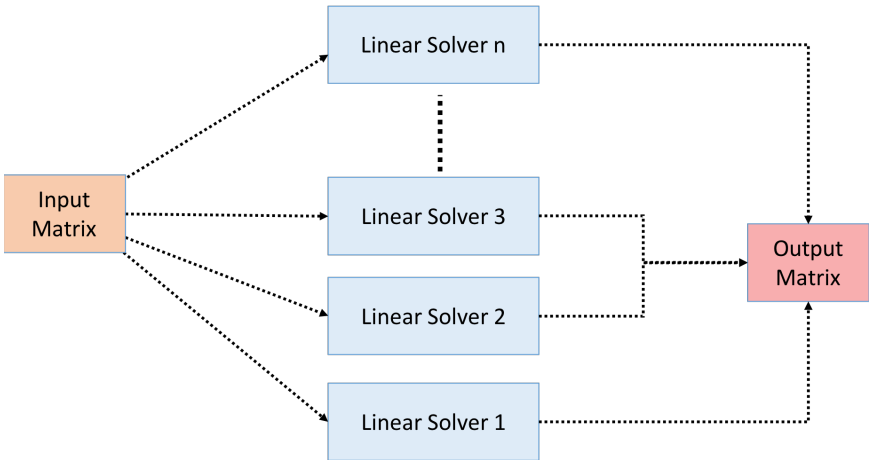


Figure 6.2: A high-level idea for population coding architecture for linear solver. The motivation here is to divide computations in temporal and spatial domain

The motivation behind this proposal is that since we are performing computations on random bitstreams, the average of all of these independent units will reduce the error that comes up due to variance [93]. Because the error gets reduced faster, the results will stabilize early as a result speeding up the computation.

6.3 Removing decorrelators

Sec. 5.3.3 explains the need for decorrelators. Especially, for a population coding based setup as discussed in previous section 6.2, we would need decorrelators for input matrices as well as recurrent path values to make sure each instance of linear solver is operating on different set of values at a given time tick. Unfortunately, decorrelators require random number generators (RNGs) which can be expensive in terms of hardware resource requirements and power. In a matrix-based algorithm, a single feedback loop could require several decorrelators for each element of a matrix. For this reason, we want to eliminate the need for decorrelators whenever possible. Indeed, our experiments show that this is possible if population coding is implemented in a specific way. Before we make the decision regarding which population coding based setup would be best for us, it is important for us to first understand different possible population coding architectures configurations that we can have with Hopfield neural network style linear solvers. As shown in table 6.1 there are eight possible configurations to design the population coding based architecture.

Individual linear solver with parallel operations refers to the setup where we have n different instances of linear solver unit, each one operating in isolation from the other units. Once the iterations are complete, the user would collect the results from every of of these linear solver units and average out the n-different results to get the output matrix. The setup for individual linear solvers have been shown in figures 6.3, 6.5, 6.7 and 6.9.

Averaged feedback linear solver refers to the setup where we have n different feedforward linear solver units operating in parallel, then we take an average of all of the n results,

and finally the averaged result is passed back in to the linear solver through recurrent path. The averaging unit is implemented using the fixed gain division circuit that was proposed in chapter 3 (refer to fig. 3.3(d) for stochastic computing based design and fig. 3.9(d) for TrueNorth based spiking neural network design). The setup for averaged feedback linear solvers have been shown in figures 6.4, 6.6, 6.8 and 6.10

Table 6.1: Population coding based architectures with different allotment of decorrelators

Linear solver setup	Decorrelators for input values	Decorrelators in recurrent path
Individual linear solver with parallel operations	Yes	Yes
Individual linear solver with parallel operations	Yes	No
Individual linear solver with parallel operations	No	Yes
Individual linear solver with parallel operations	No	No
Averaged feedback linear solver	Yes	Yes
Averaged feedback linear solver	Yes	No
Averaged feedback linear solver	No	Yes
Averaged feedback linear solver	No	No

In later subsections we will discuss the architecture for each one of the eight units as proposed in table 6.1.

6.3.1 Decorrelators for input values and feedback path

This section describes the setup where we have decorrelators present for both the input values as well as the recurrent path values. Figures. 6.3 and 6.4 shows two possible setup

for linear solver when we have decorrelators present for the input values as well as the recurrent path values. In fig. 6.3 we have individual linear solvers operating in parallel. Whereas in fig. 6.4 we average out the linear solver results before they are fed back into each one of the linear solver units.

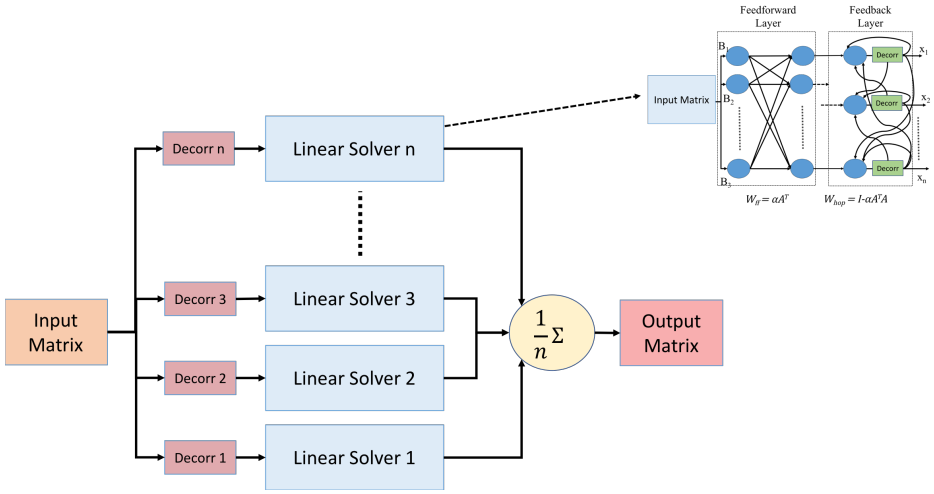


Figure 6.3: The proposed architecture for population coding based approach when we have decorrelators present for input values and feedback path values. This figure shows the setup for individual linear solvers operating in parallel.

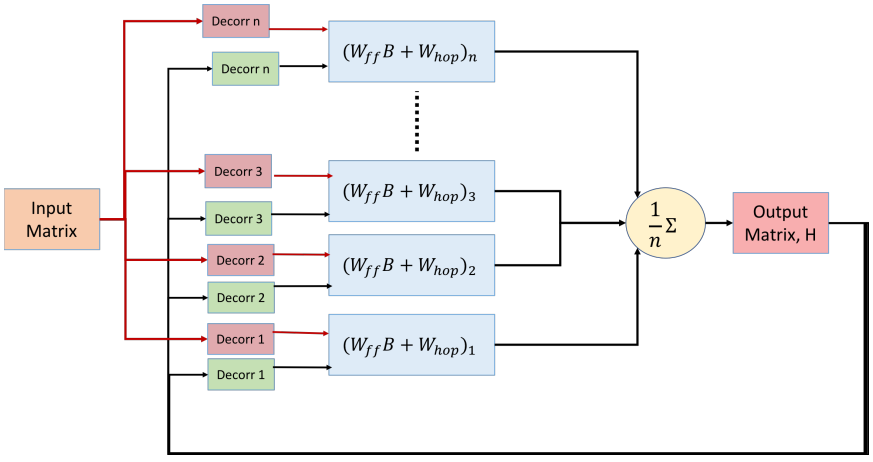


Figure 6.4: The proposed architecture for population coding based approach when we have decorrelators present for input values and feedback path values. This figure shows the setup where the linear solver results are first passed through a constant averaging unit before they are fed back into each one of the linear solver units.

6.3.2 Decorrelators only for input values

This section describes the setup where we have decorrelators only in the front path, that is, decorrelators present for input values. Figures. 6.5 and 6.6 shows two possible setup for linear solver when we have decorrelators present only for the input values and are absent from the recurrent path. In fig. 6.5 we have individual linear solvers operating in parallel. Whereas in fig. 6.6 we average out the linear solver results before they are fed back into each one of the linear solver units.

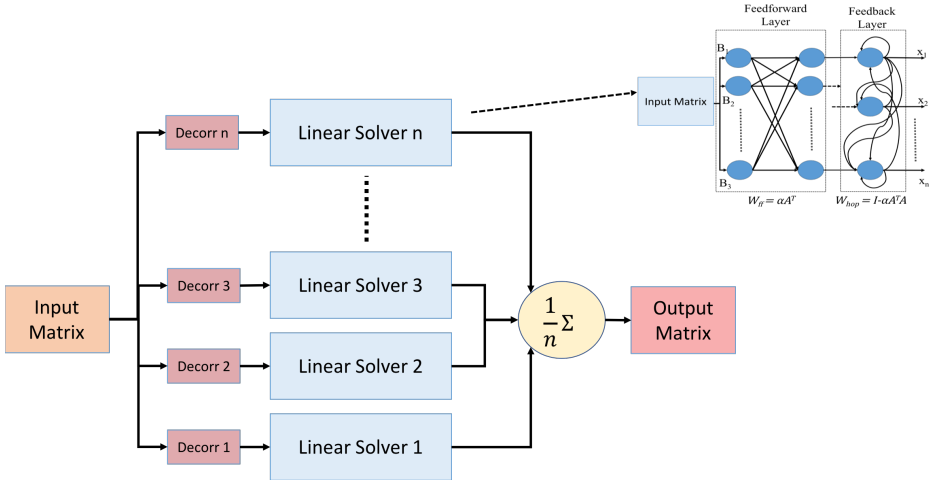


Figure 6.5: The proposed architecture for population coding based approach when we have decorrelators present only for the input values and are absent in the recurrent path. This figure shows the setup for individual linear solvers operating in parallel.

6.3.3 Decorrelators only in the recurrent path

This section describes the setup where we have decorrelators only in the recurrent path, that is, decorrelators present for the values are fed back into the linear solver. Figures. 6.7 and 6.8 shows two possible setup for linear solver when we have decorrelators present only for the recurrent path values and are absent for the input values. In fig. 6.7 we have

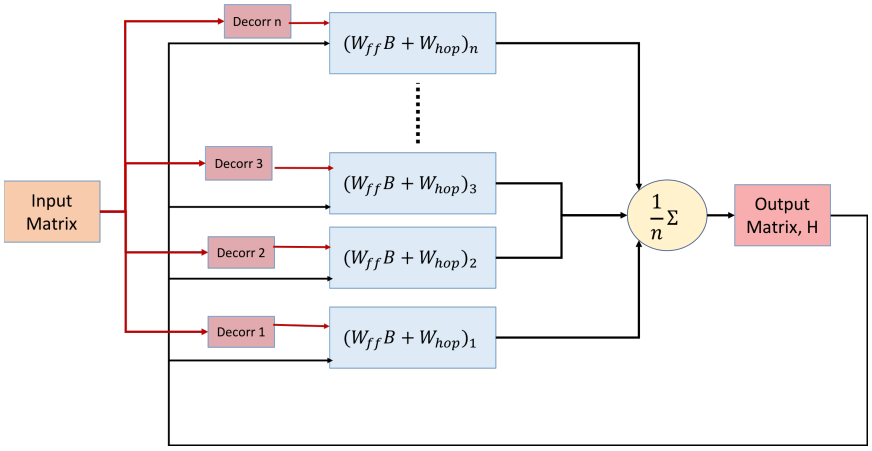


Figure 6.6: The proposed architecture for population coding based approach when we have decorrelators present only for the input values and are absent in the recurrent path. This figure shows the setup where the linear solver results are first passed through a constant averaging unit before they are fed back into each one of the linear solver units.

individual linear solvers operating in parallel. Whereas in fig. 6.8 we average out the linear solver results before they are fed back into each one of the linear solver units.

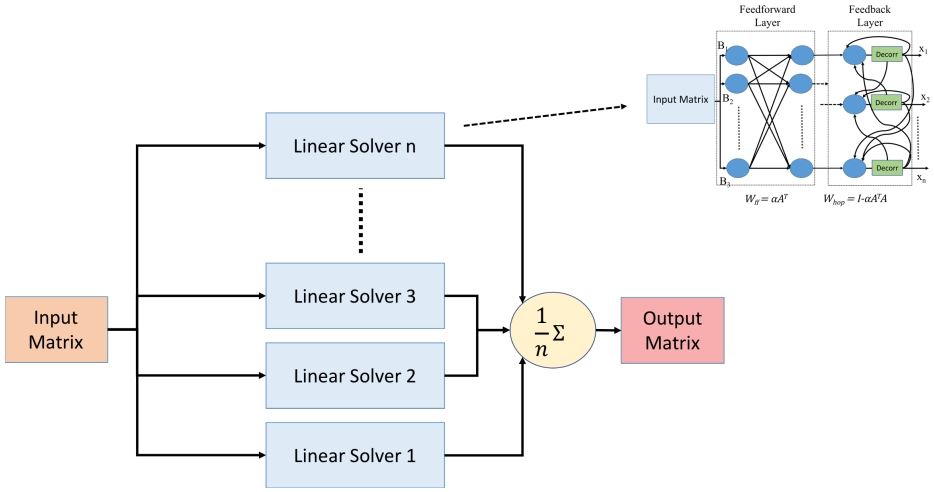


Figure 6.7: The proposed architecture for population coding based approach when we have decorrelators present for recurrent path values. This figure shows the setup for individual linear solvers operating in parallel.

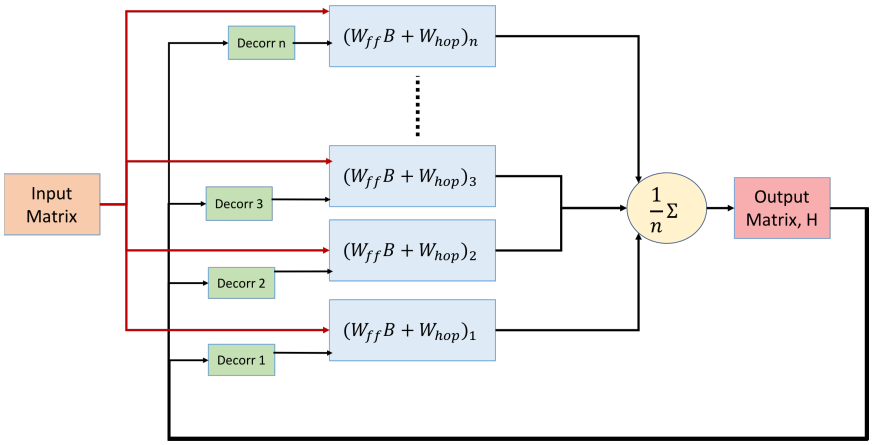


Figure 6.8: The proposed architecture for population coding based approach when we have decorrelators present for recurrent path values. This figure shows the setup where the linear solver results are first passed through a constant averaging unit before they are fed back into each one of the linear solver units.

6.3.4 No decorrelators present in the population coding architecture

This section describes the setup where we have do not have any decorrelators present. Figures. 6.9 and 6.10 shows two possible setup for linear solver when we do not have any decorrelators. In fig. 6.9 we have individual linear solvers operating in parallel. Whereas in fig. 6.10 we average out the linear solver results before they are fed back into each one of the linear solver units.

Even though the two setups explained in this section are part of the 8 configurations that were proposed in table 6.1, we won't be analyzing these two setups because they are not interesting from evaluation point-of-view. As per the proposed design for both of these setup, each one of the linear solver is performing the exact same set of operations, on exact same bitstream, at the same instance of time. The behavior of these two architectures will be similar to having a single instance of linear solver because the spatial representation of data is not present.

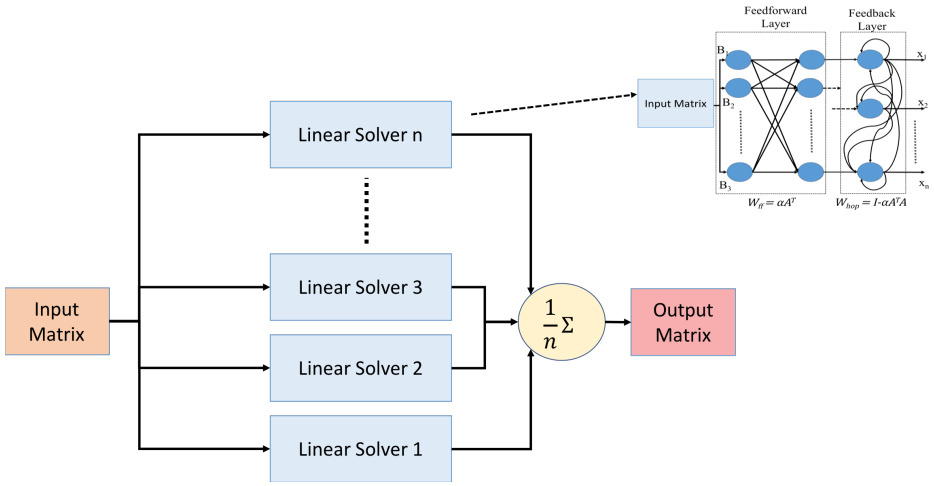


Figure 6.9: The proposed architecture for population coding based approach when we do not have decorrelators. This figure shows the setup for individual linear solvers operating in parallel.

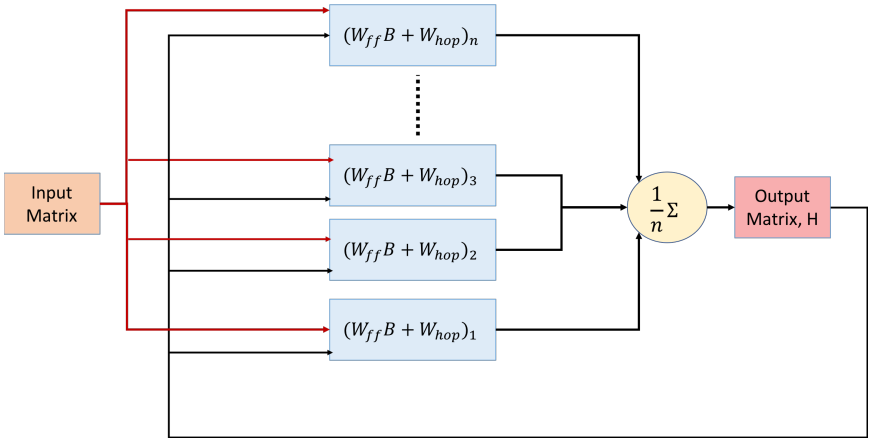


Figure 6.10: The proposed architecture for population coding based approach when we do not have decorrelators. This figure shows the setup where the linear solver results are first passed through a constant averaging unit before they are fed back into each one of the linear solver units.

6.3.5 Removing decorrelators analysis

We conducted accuracy and precision experiments for 6 different architectures (as proposed in table 6.1) to empirically evaluate which implementation style would be the better option when compared with the baseline models of figures 6.3 and 6.4. As discussed in previous section 6.3.4, we did not evaluate the architecture style where no decorrelators were present

in the proposed design. The evaluation was done by generating 10 different matrices all of which have dimensions of 3x3. The data for for each one of these matrices was generated randomly in the range $\in [-1, 1]$. The population count was 5 for all of the different implementation style.

The results of these experiments are shown in figures 6.11 and 6.12. As expected, in fig. 6.11 the lowest loss is achieved by the architectures where we have decorrelators present for input values and in recurrent path (figures 6.3 and 6.4). Because each one of the linear solver instance is performing calculations for different independently distributed bitstream at the same instance of time, the average of this architecture achieves the fastest reduction in loss. Unsurprisingly we can observe that for the setup that individual linear solver units without decorrelators in recurrent path (refer to fig. 6.5) shows an increase in loss over time because the inputs become more and more correlated. This behavior is similar to the architectures that we had looked when we discussed about importance of decorrelators in section 5.3.3. Interestingly, as per the plots shown of fig. 6.12, the average loss of the implementation in Fig. 6.6 is similar to the results that we get from the baseline implementation of figures 6.3 and 6.4. Intuitively, the averaging operation introduces enough randomness to the feedback to keep the inputs uncorrelated (since each output is uncorrelated with the other outputs it is being averaged against).

Thus, using the proposed implementation of population coding in Fig. 6.6 can provide up to 1.7x reduction in FPGA resource utilization requirement (# LUTs and # FFs). This advantage, along with the ability to combat scaling issues, illustrates that the implementation details of population coding matter. Additionally, the trade-off space is not simply latency versus area – it includes accuracy, scaling factors, and non-intuitive implementation

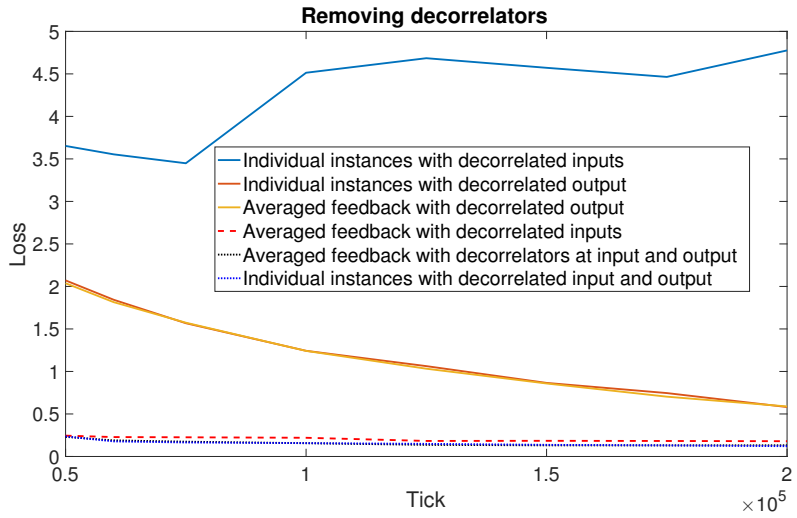


Figure 6.11: Average loss of population coded linear solver implementation. The population count is 5 for all of the different implementation style. The average loss was calculated over 200,000 iterations. This figure shows the comparison of average loss between six different implementation style. Individual feedback refers to the implementation shown in figures 6.3, 6.5, and 6.7. Averaged feedback refers to the implementation style of figures 6.4, 6.6, and 6.8.

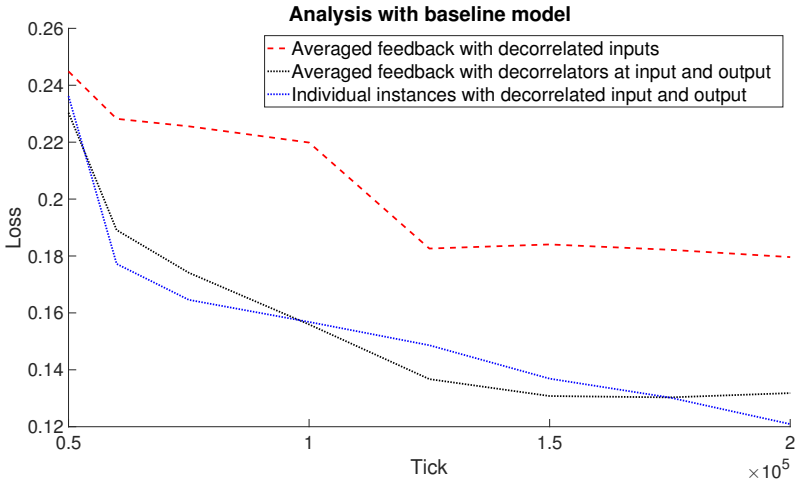


Figure 6.12: This figure shows the shows the zoomed-in plot of fig. 6.11 for averaged feedback linear solver architecture(fig 6.6) and baseline models that have decorrelators present for input values and recurrent path (fig. 6.3 and 6.4).

dynamics.

6.4 Selecting output from multiple linear solvers each with different α

In section 5.4 we had proposed a stochastic computing architecture for computing the learning rate parameter α that will guarantee the iterative equation 2.6 will always converge. But the proposed α is a conservative value that will guarantee convergence. Taking inspiration from theory of stochastic gradient descent [49] we will evaluate if there are multiple linear solvers that are operating in parallel, each with different value of α, how fast can such a setup converge.

Fig. 6.13 shows the setup for proposed error selection method. There are n different linear solver units, each operating on same input value bitstream, with decorrelators present in recurrent path. Each linear solver unit has a different α value, as a result, every Hopfield neural network in the proposed setup has different W_{ff} and W_{hop} weight values. Once the required number of iterations are complete, a separate hardware unit will iterate through all of the n linear solver units and in the end select the output value that has the minimum error, calculated as per the eqn. 6.1.

$$\text{Error} = \min_{k} \|A^T A X_k - A^T B\|_F \tag{6.1}$$

Where X_k in eqn. 6.1 is the output matrix for the k^{th} linear solver and $k \in [1, n]$.

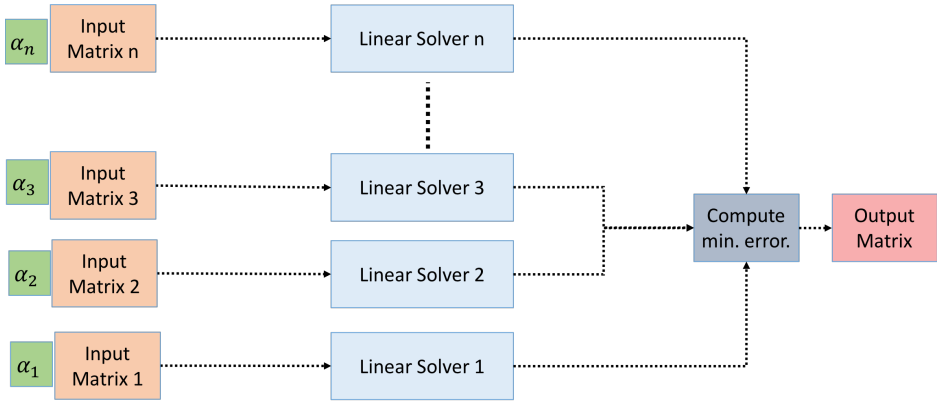


Figure 6.13: Shows multiple linear solver units operating in parallel where each one of the instances has a different α. Once the required number of iterations are complete, a separate hardware unit would iterate through each one of the n linear solvers, and select the output bits with the minimum error, that is, $\min_k \|A^T A X_k - A^T B\|_F$

6.4.1 Minimum error selection technique evaluation

To understand the behavior of minimum error selection technique, we generated 25 different input matrices each with different matrix size. The smallest matrix dimension was 2 and the largest dimension was 10. Each element in the generated matrix was randomly selected in the range $\in [-1, 1]$. The value of α_k for each one of the linear solver units was selected in range from $[\alpha, 2\alpha]$. The population count (number of independent linear solver units) was 25 for this experiment. The loss was calculated based on the formula shown in equation 6.2, where X_{actual} refers to the output matrix computed from stochastic linear solver with minimum error (as given by eqn. 6.1) and $X_{expected}$ is the output matrix computed using the proposed iterative equation 2.6 that had 64-bit double precision compute units when simulated in MATLAB. The experiment was repeated 25 times and we report average loss for all 25 experiments in the results.

$$\text{Loss} = \|X_{actual} - X_{expected}\|_F \tag{6.2}$$

Plots shown in figures 6.14 and 6.15 represent a comparison between the variation of loss over increasing number of time ticks for different population coding based setup and minimum error selection technique as presented in this section. Just as we had predicted, similar to stochastic gradient descent theory, having multiple linear solvers that are operating in parallel, each with different value of α, has a much faster reduction in loss compared to a single instance based approach. But a naive population coding technique with a population count of 5 and 10, attains a much smaller loss with fewer time ticks compared to minimum error selection technique. Unlike minimum error selection technique, a basic population coding architecture does not require any additional hardware that would iterate through every linear solver and select the output values that achieves the minimum loss. As a result, the primary focus of this dissertation will be on population coding architectures that were discussed in previous section 6.2.

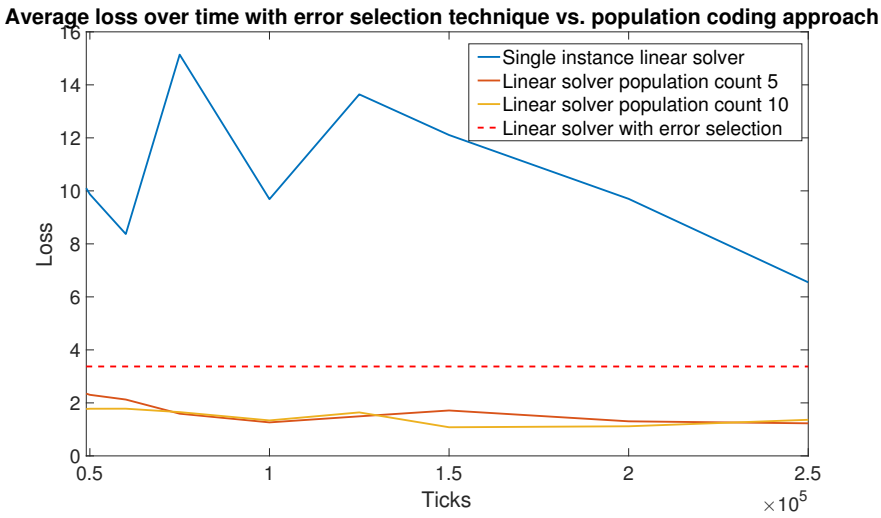


Figure 6.14: Shows the variation in loss over time for single instance of linear solver, the variation of loss due to minimum error selection technique and comparison with population coding technique (discussed in section 6.2) for population count of 5 and 10. Even though it seems that the loss with minimum error selection technique does not change, a zoomed-in plot as shown in fig. 6.15 shows that the loss value does reduce over time.

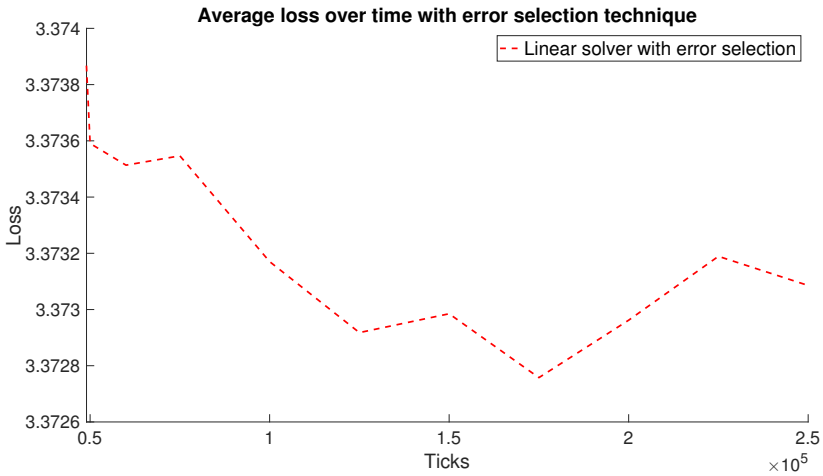


Figure 6.15: This figure is a zoomed-in plot of fig. 6.14, which shows the variation of loss over time ticks for minimum error selection technique.

6.5 Summary

As stated in section 3.1.3, one of the challenges with stochastic computing is long latency. We build upon the theory and functional units that was presented in chapter 3, and 5, and propose a population coding based architecture for stochastic computing. Similar to neuroscience, our approach here is to speed up the calculations by having multiple linear solver units performing calculations in parallel and combining the results that we get from each one of the independent linear solver unit. We also discussed having a population coding setup where the input values have to pass through decorrelators and the recurrent path values have to first pass through an averaging unit before they get fed back in, shows reduction in loss which is very close to baseline approaches. This evaluation helped us in achieving 1.7x reduction in hardware resources (# LUTs and # FFs in FPGA) when compared to baseline approaches where we had decorrelators present for input values and recurrent path values. Finally, we evaluated the population coding scheme against a

minimum error selection technique, an approach which takes inspiration from stochastic gradient descent [49]. We observed that having multiple linear solvers that are operating in parallel, each with different value of α, has a much faster reduction in loss compared to a single instance based approach. But a naive population coding technique with a population count of 5 and 10, attains a much smaller loss compared to minimum error selection technique.Readers are advised to refer to chapter 9; section 9.4 to look at the evaluation of population coding approach against more standard approaches. Later in chapter 9; section 9.5 we will be analyzing power and energy consumption of population coding based approach on an FPGA against more standard techniques.

7 ADAPTIVE SCALING

Chapter 4 introduced the importance of having a proper scaling factor to guarantee that intermediate computations never saturate. The term scaling factor, η, that was computed in equation 4.10 guarantees that under no circumstance, the spiking neural implementation will saturate. But this mathematical bound takes into account the worst case behavior of the algorithm. It is very likely that the firing rate of neurons are not even close to the estimated bound, as can be observed from figure 9.1. Therefore, division by such a high value scaling factor might slow down the computation speed since it will take longer time to represent the same value in terms of number of spikes. Apart from slower calculations, estimating the term η requires us to first compute the minimum singular value of matrix A, that is, σ_{min}. This approach has a crucial drawback, that is, now we would need a separate hardware that computes σ_{min} for streaming applications. Hence, in this chapter we propose an on-chip architectural solution for adaptively scaling the calculations in Hopfield based linear solver implementation. To address this issue we have proposed an adaptive scaling unit, which would detect if there is a neuron that is firing at the maximum firing frequency, then it would scale down the input matrix appropriately. The following subsections will present the architectural change in detail.

7.1 Adaptive scaling spiking neural network architecture

Figure 7.1 shows the proposed architecture for adaptive on-chip scaling. The figure shows the architectural implementation for hard-coded weights, where the values of input ma-

trix B are being scaled. In case of spike based weight representation scheme as per the mathematical proofs and equation 5.1, the values of matrix A would have to be scaled.

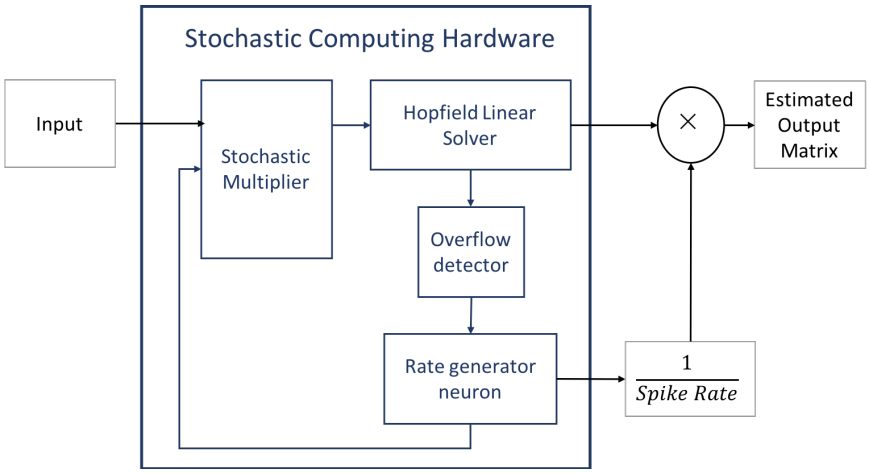


Figure 7.1: The TrueNorth architecture for adaptive scaling

Figures 7.2(a) and (b) show the TrueNorth neuron setup for overflow detector and rate generator neurons. The rate generator neuron initially fires at maximum frequency, thus representing the value 1. Initially, all of the values of input matrix B are passed through stochastic multiplier and serve as input to Hopfield linear solver. The overflow detector makes sure that none of the neurons inside the linear solver implementation fire at maximum frequency. Logically purpose of overflow detector is to count the number of ones that are appearing continuously. It is meant for the user to specify how many number of continuous ones represent that a neuron has saturated. The TrueNorth based neuron circuit that calculates this string of ones is shown in fig. 7.2(a). If the overflow detector detects any one of the neuron in linear solver is firing at maximum frequency, it will send an inhibition signal to the rate generator (shown as red circle in figure 7.2(b)) and reduce the membrane potential (or counter value in fig. 7.3(b)) of the said neuron so that its firing rate decreases, as a result, the scaled down values of input matrix B are passed on to the linear

solver. There would be an additional buffer at the output of linear solver which would store the results. So whenever a neuron saturation is detected, the buffer would flush the values that it was storing. As the spike rate represents value in the range $\in (0, 1]$, we would have to do the conversion to proper scaling factor off-chip, that is, not on TrueNorth or in stochastic computing hardware. Since the denominator will have a value less than or equal to 1, this division cannot be done on a spiking neural substrate.

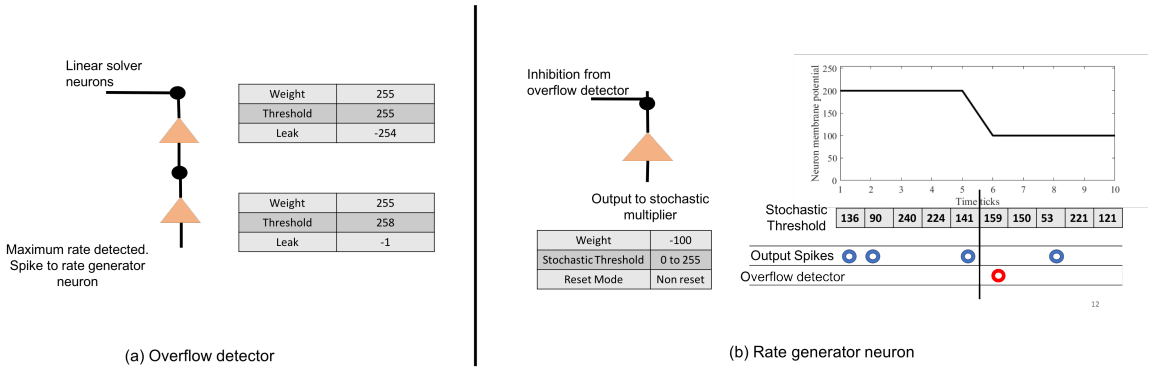


Figure 7.2: The neuron parameters and setup for (a) Overflow detector (b) Rate generator neuron

7.2 Adaptive scaling stochastic computing architecture

Figures 7.3(a) and (b) show the digital logic design setup for overflow detector and rate generator units in a stochastic computing environment. The goal of this architecture is the same as spiking neural network setup, that is, if there is any arithmetic unit that is firing at maximum frequency, then overflow detector 7.3(a), will detect this firing saturation. The Th1 and Th2 parameters shown in fig. 7.3(a) are the threshold values for overflow detector logic. To replicate the TrueNorth neuron behavior (as shown in fig. 7.2(a)), the parameters Th1 and Th2 can be set to 255 and 258. Similarly, rate generator logic of fig. 7.3(b) can be

made to behave similar to the TrueNorth neuron by having the decrement value to be -100 and let the random number generator (RNG) vary between 0 to 1024.

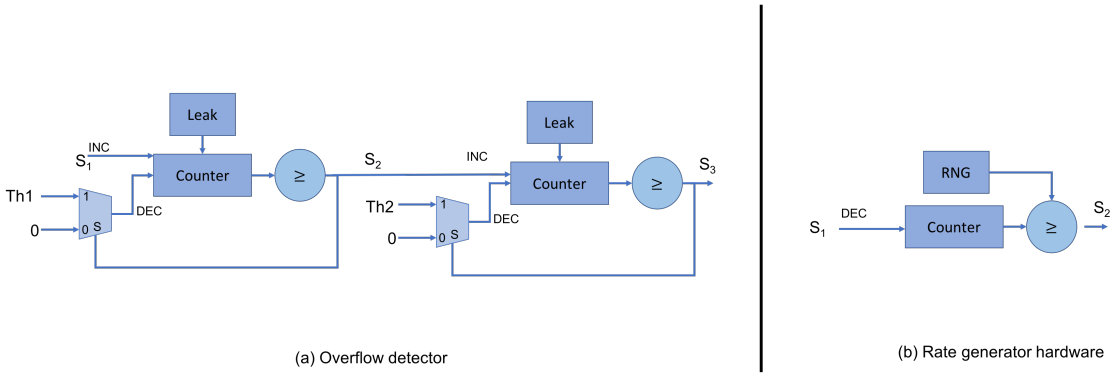


Figure 7.3: The stochastic computing setup for (a) Overflow detector (b) Rate generator logic, using digital circuit design.

7.3 Summary

This chapter described an adaptive scaling architecture for random input bitstreams that would keep the computations in the range $\in [0, 1]$. The digital design can be implemented on IBM TrueNorth and it can be also extended to stochastic computing substrate. This computation mechanism does not require the user to calculate the value of minimum singular value of matrix A, that is, σ_{min} and if there is any compute unit whose intermediate calculation value has saturated, the proposed mechanism will automatically scale the input matrix values appropriately. A thorough evaluation of adaptive scaling approach is presented in chapter 9, section 9.6. Readers are advised to refer to the evaluation to better understand the impact of adaptive scaling architecture when compared to scaling the input matrix values by the factor η in eqn. 4.10.

8 EXPERIMENTAL SETUP

This chapter will present a detailed experimental setup using which we performed our evaluations. The initial portion of this chapter will focus on generating the test examples which we used to evaluated the accuracy and performance of the algorithms. Later we will present the methodology we used to evaluate the proposed algorithm on different hardware substrates such as IBM TrueNorth, Xilinx FPGAs and Lattice FPGAs.

8.1 Bitstream accuracy and precision analysis

To understand the behavior of minimum error selection technique, we generated 25 different input matrices each with different matrix size. The smallest matrix dimension was 2 and the largest dimension was 10. The matrix does not have to be a square matrix. Each element in the generated matrix was randomly selected in the range $\in [-1, 1]$. The loss was calculated based on the formula shown in equation 8.2, where X_{actual} refers to the output matrix computed from stochastic linear solver and $X_{expected}$ is the output matrix computed using the proposed iterative equation 2.6 that had 64-bit double precision compute units when simulated in MATLAB. A lower absolute error would be preferred. The experiment was repeated 25 times and we report average loss for all 25 experiments in the results.

$$AX = B \tag{8.1}$$

$$\delta_{absolute-float} = \|(X_{actual} - X_{expected})\|_F \tag{8.2}$$

8.2 Application analysis

This section describes the experimental setup for applications that were tested using TrueNorth based linear Hopfield solver.

8.2.1 Target tracking

The first class of application that is considered is a typical target tracking scenario, shown in Fig. 8.1. These examples are widely popular in drone control, image localization and image tracking. Usually for such applications the image or target is predefined, that is, the shape of the object is known beforehand and the goal is to be able to continuously monitor the motion of the said object by placing a bounding box around it.

In the proposed example, a real-time video input is preprocessed to extract features (e.g. edges of particular orientations) to form a feature set. This feature set is then compared against a set of templates to identify objects of interest, with the goal of tracking the objects in the image frame as they move in three dimensions. As a proof of concept, a very simple image was drawn whose feature set consists of just three edges similar in appearance to the letter H. The height and width of this symbol is 1 cms, each. The position of the image can vary from -15 cms to 15 cms along vertical and horizontal directions and the scale can change by 0.5, 0.25, 1, 2, and 4 times from the previous size.

Equations 8.3 and 8.4 show the structure of matrices A and B that contains x and y coordinates of the features. To determine size and placement of the bounding box for the tracked image, the theory of affine transforms [38] is used which states that a current

image B can be matched to its template A with an affine transformation X using a matrix multiplication $AX = B$, as long as the image has only been transformed with respect to that template in scale, rotation, or 2D translation (a similar self-learning visual architecture was investigated in [92]). Algorithm 6 presents the algorithmic details of performing object tracking. By employing matrix division implemented in a recurrent Hopfield network, the affine transform X can be derived that maps the current image input B to the template A, and can determine the scale, horizontal and vertical transformations from the corresponding entries in matrix X.

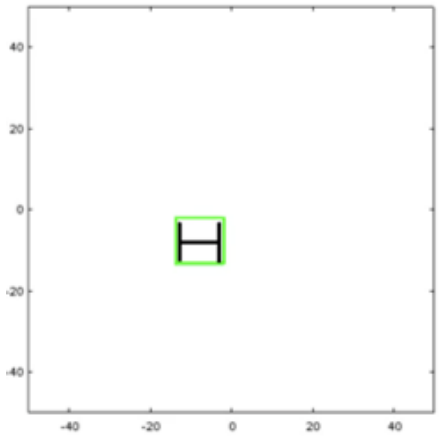


Figure 8.1: Screenshot illustrates target tracking application.

$$A = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} \tag{8.3}$$

$$B = \begin{bmatrix} \widehat{x_1} & \widehat{y_1} & 1 \\ \widehat{x_2} & \widehat{y_2} & 1 \\ \widehat{x_3} & \widehat{y_3} & 1 \end{bmatrix} \tag{8.4}$$

Algorithm 6 Object detection using Harris corner detector for feature extraction

1: Extract pairs of feature points [47] and keep the coordinates in matrix A
2: Extract pairs of feature points and keep the new set of coordinates in matrix B
3: Solve for X to to get transformation matrix for determining the location, scale and rotation of the target : $X = (A^\top A)^{-1}B$.

8.2.2 Inverse Kinematics

The second class of application that is considered is inverse kinematics, specifically the two joint arm problem as shown in fig. 8.2. In this algorithm the objective is to move the two joints of an arm until the end effector is close to the target position (shown as a bold 'X' symbol in the image).

Eqn. 8.5 shows the equation for two-joint arm based inverse kinematics equation for which the Hopfield network based linear solver was used. Equations 8.6 and 8.7 show A and B matrices, respectively. For the purpose of these experiments the arm lengths, viz., L_1 and L_2 have been selected as unit lengths or 1 cms. The target can appear at any spot within a radius of 2 cms from the origin. When the target changed its position, the arm would start moving from the position it stopped at in the previous iteration, that is, the position of end-effector would start from where it previously left-off.

This demonstration shows how a linear solver can be used in inverse kinematics based applications. Steps involved in inverse kinematics application has been shown in algo. 7. The coordinates of arm 2 (represented by green markers in fig. 8.2) can be extracted using a feature extraction technique. As the length of arms have been set to unit length, the values in matrix A will all have a magnitude of less than 1. When the arm moves from one position to another, the matrix Θ_{k+1} gets updated in every step. Based on the new position

of arms, the value of matrices A and B are computed again so that they can be later used for the next step.

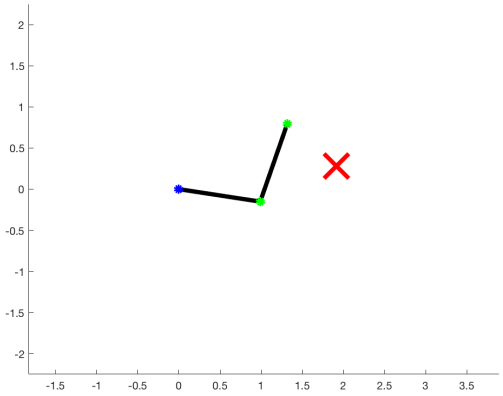


Figure 8.2: This screenshot that illustrates inverse kinematics experiment.

$$\Theta_{k+1} = \Theta_k - (LearningRate)A^{\dagger}B \tag{8.5}$$

$$A = \begin{bmatrix} -L_1 sin(\theta_{k,1}) & -L_2 sin(\theta_{k,2}) \\ L_1 cos(\theta_{k,1}) & L_2 cos(\theta_{k,2}) \end{bmatrix} \tag{8.6}$$

$$B = \begin{bmatrix} x_{target} - x_{current} \\ y_{target} - y_{current} \end{bmatrix} \tag{8.7}$$

8.2.3 Optical flow

Finally, the third class of application which we considered is optical flow, shown in fig. 8.3. Optical flow is another popular algorithm that is used in applications such as drone control and object tracking. Unlike the previously described target tracking algorithm, there is

Algorithm 7 Inverse Kinematics

Require: Learning rate, γ, that decides motion of robotic arm.

1: **for** $k = 1, 2, \ldots$ **do**
2: Current value of angles of the robotic arm w.r.t x-axis. Store these angles in matrix Θ_k
3: Compute current x and y coordinates of robotic arm positions. Have these values in matrix A.
4: Compute difference between desired coordinates for the end of robotic arm to reach and current coordinates for the end of robotic arm. keep the difference in matrix B.
5: Get the matrix inverse of A.
6: Solve the equation $\Theta_{k+1} = \Theta_k - \gamma A^{-1} B$
7: **end for**

no need for feature extraction, rather the computations are carried out by calculating the Jacobian between pixel intensities across different frames, as a result, significantly reducing the computation time. The main underlying principle of this algorithm is that for a short duration of time there is no change in pixel intensity. Optical flow outputs direction and speed by which an observed object is moving.

To compute the velocities of optical flow, the Lucas-Kanade algorithm was used. Algo. 8 summarizes the steps for performing Lucas-Kanade algorithm [58]. Equations 8.8 and 8.9 show matrices A and B, respectively. $I_x(qi)$, $I_y(qi)$ and $I_t(qi)$ represent derivatives across x direction, y direction and time, respectively, around pixel qi. For this experiment, we assume a window of size 5-by-5 pixels.

The presented optical flow demonstration is similar to the one reported in [24] where the horizontal bar is continuously moving upwards and the vertical bars are moving to the left of the screen. In the developed prototype it is demonstrated that the direction of the bar's movement can be reported without any error and also the approximate speed by which the two bar's move can be reported. The thickness of the two lines was selected to be 3 cms and the speed of the bar's movements was set to 12 cms per second. The size of

the frame that was selected to draw these lines is 50 cm by 50 cm. The velocity of the two line's movement is calculated by solving for X in the equation $AX = B$, where matrix A contains partial derivatives of initial image frame with respect to directions x and y, and matrix B contains partial derivatives of pixel positions between initial image frame and image frame at time t. After implementing matrix division, output matrix X will report the speed and direction of the image pixels, by computing the pseudoinverse of matrix A.

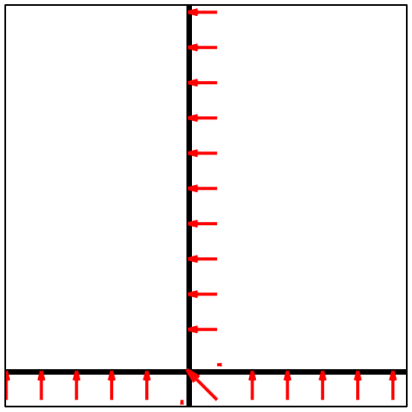


Figure 8.3: Optical flow application screenshot

$$A = \begin{bmatrix} I_x(q1) & I_y(q1) \\ I_x(q2) & I_y(q2) \\ \vdots & \vdots \\ I_x(qn) & I_y(qn) \end{bmatrix} \tag{8.8}$$

$$B = \begin{bmatrix} -I_t(q1) \\ -I_t(q2) \\ \vdots \\ -I_t(qn) \end{bmatrix} \tag{8.9}$$

Algorithm 8 Lucas Kanade Optical Flow

1: Compute x and y spatial derivatives of frame at time t. These values are in present in matrix A.
2: Compute temporal derivatives between frame at time t and frame at time $(t+T)$, where $T > 0$. These values are in present in matrix B.
3: Solve for X to get velocity of motion of the object appearing in field of vision : $X = (A^\top A)^{-1}B$.

8.2.4 Error analysis

To analyze these proposed set of applications, we report the relative error (eqn. 8.10) and absolute error (eqn. 8.11) between the output that is obtained from the TrueNorth-based Hopfield linear solver and the output that is obtained using MATLAB's double precision pseudoinverse library function. A lower absolute error or relative error would be preferred based on how much error can the application tolerate. T

$$\delta_{relative} = \| \frac{(\text{Output}_{experiment} - \text{Output}_{actual})}{\text{Output}_{actual}} \|_F \tag{8.10}$$

$$\delta_{absolute} = \| (\text{Output}_{experiment} - \text{Output}_{actual}) \|_F \tag{8.11}$$

8.2.5 Robotic Bee

We evaluate the proposed architecture for deployment on robotics applications such as micro-aerial vehicles (MAVs) [61]. Particularly, in this dissertation we will be focusing on deploying algorithms on the Robotic Bee MAV. Robotic bee (RoboBee) as shown in Fig. 8.4 is an insect-scale micro-aerial vehicle (50-500 mg) that can be used for many applications, such as surveillance, climate monitoring, hazardous environment exploration and assisted

agriculture. The size of this robot imposes stringent mass (<500 mg) and power (<350 mW) constraints [22] on all components of the system. But 90% is used by the actuator to keep the robot airborne [26]. Therefore, all of the matrix computations have to be performed in a power budget of 35 mW. The onboard computing must be optimized to perform a variety of computations within acceptable bounds on performance and under the available power budget. Prior works [18] and [17] have considered a spiking neural network based controller for stabilizing the flapping insect-scale robot. Similarly, other works such as [22] have proposed an ASIC that can perform optical flow based control, while keeping the computations in the acceptable power budget for RoboBee. Unfortunately, these designs can only perform the minimum computation required to keep the bee in stable flight. They fail to address how the RoboBee will be able to perform the computations required for its many use-cases. Having identified the underlying algorithms to target the RoboBee's applications, the goal of our work is to devise algorithmic and computing schemes for operations such as calculating matrix pseudoinverse and singular value decomposition, and deploy these computations on low-power hardware substrates like FPGAs. Unlike the work done in [22], our approach is not restricted to a custom ASIC that can perform only one kind of operation. Instead, we can map the proposed computations to any hardware that supports stochastic computing.

Matrix dimensions

In this section we summarize the matrix dimensions on which we performed the experiments for different algorithms. The least squares minimization were deployed on FPGA for the matrix dimensions described in this section.

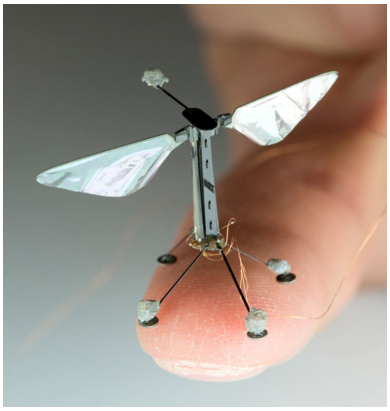


Figure 8.4: Image of RoboBee, a micro-aerial vehicle. This image has been taken from [60].

Optical Flow

For this application, the convolution window size is 5×5 pixels, the matrix A will have a dimension of 25×2 and matrix B will have a dimension of 25×1. The output vector X (which reports the motion vectors) will be of dimension 2×1.

Inverse Kinematics

Since the proposed algorithm is for two-arm inverse kinematics problem, the rotation matrix A will have a dimension of size 2×2, and coordinate vector B will be of dimension 2×1. The output vector, X, will have a dimension of size 2×1.

Object tracking

We consider a simple setup where matrix A had a dimension of 3×3 containing only three extracted features. Similarly, matrix B had a dimension of 3×3. The resulting 2-D affine transformation matrix X will have a dimension of 3×3.

8.2.6 Hardware substrate evaluation

Keeping in mind the power and latency constraints that are discussed in Sec. 8.2.5, the first objective was to evaluate the feasibility of deploying standard algorithms and architectures for matrix computations. We performed the first set of evaluations on an ARM A15 processor. The QR inverse algorithm (for least squares minimization) were implemented using eigen library [33] and the simulations were done on Gem5 [10] using system call emulation mode. The power numbers were collected using McPat simulator [**Li2009**]. Based on the power numbers that were reported from McPAT, ARM A15 would consume static power of 11.73mW and a peak dynamic power 779.85 mW. This is more than the allowable power budget for RoboBee.

A second set of evaluations were performed for ultra low-power Lattice FPGAs, because they are able to meet the power constraints of RoboBee. For deploying algorithms on Lattice FPGAs, we first develop these algorithms for Xilinx platforms in Vivado and based on the post-implementation utilization report generated, we extrapolate the resource consumption to Lattice FPGAs. Since logic cell design for Xilinx FPGAs [101] and Lattice FPGAs [89] are the same, we assume that the LUT and flip-flop counts that are reported for the Xilinx-based implementation will be similar on a Lattice FPGA. If an algorithm consumes resources unavailable on a Lattice FGPA, then we report the data with respect to the Xilinx ZYNQ-7000 FPGA. Our complete FPGA-based evaluations include floating point and fixed point HLS implementations, as well as SC implementations of varying population size.

All our implementations are required to meet the real-time deadline of the RoboBee. Some implementations, such as the floating point HLS implementation, may be much

faster than the deadline.

8.3 Experimental setup for hardware analysis

In this dissertation we have compared the power and energy consumption stochastic computing based linear solver implementation on three different hardware platforms.

***TrueNorth*:** The spiking neural network deployment was done on IBM's TrueNorth neurosynaptic system, specifically we did the evaluation on ns1e board (as shown in fig. 3.6)

***FPGA*:** The second set of hardware substrates on which we performed our evaluation are Xilinx and Lattice FPGAs. We deployed the proposed stochastic computing architecture on Xilinx Virtex-7 and Xilinx ZedBoard hardware. Based on the resource utilization report that we collected from Xilinx Vivado software, we estimated the amount of resources that might be consumed on a Lattice FPGA. The evaluations were done on different FPGA models to see which architecture would give us the least power consumption for the deployed algorithm without sacrificing on performance.

For evaluations against more standard approaches, we implemented two baseline models on FPGA. In the first baseline model we selected the QR-inverse function which is a part of Xilinx HLS linear algebra library. Input matrices of different applications were provided as arguments to the QR-inverse function library. In the second baseline we implemented the iterative Hopfield neural network algorithm using Xilinx HLS syntax. The iterative algorithm was implemented using floating point and fixed point arithmetic units. The proposed fixed point arithmetic implementation was for 24-bit compute units out of which 20 bits were allocated to fractional bits. We made sure that while reporting

the resource utilization for fixed point implementation, no DSP units were being used and all of the arithmetic operations were being performed using only LUT and FF units. The target hardware operated at 50 MHz operating frequency.

***ARM A15 processor*:** Third hardware substrate against which we performed the comparison is ARM A-15 processor. The ARM A-15 processor was simulated using gem5 [11] in the system call emulation mode and the power consumption details were collected using *McPat* [54]. We used the ARM A15 configurations for Gem5 and McPat simulations that have been presented in [23].

***ASIC*:** The power numbers for ASIC implementation of linear solver have been reported using DesignVision software. The power numbers were reported for TSMC 40nm technology.

8.4 Summary

This chapter presented the various experimental setup using which we evaluated the proposed algorithm. These experimental setup are later used in chapter 9 to empirically understand various implementation trade-offs and benefits of stochastic computing. Section 8.1 serves as the basic setup for accuracy and precision analysis of the linear solver by generating matrices of different sizes and different values making up its elements. Later, in section 8.2 we presented three different applications that were used for understanding the behavior of stochastic computing based linear solver in the context of real-time applications and extend these evaluations for RoboBee hardware that was presented in section 8.2.5. Finally, Section 8.3 presented three different hardware platforms that were used for eval-

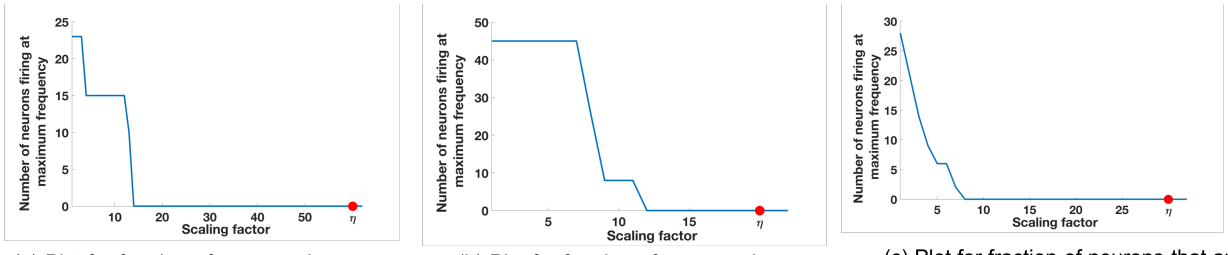
uating the proposed algorithm and analyze the energy, power and area benefits that we get from the proposed stochastic computing architecture. Section 8.3 also serves as an extension to study the power and energy constrains for RoboBee application

9 RESULTS

This section presents the validation results and observations for the mathematical models for scaling factor and the precision analysis. For all experiments, we set $\alpha = 1.9/\text{trace}(A^T A)$; equation (2.9) guarantees convergence of the iterative process for this value of the parameter.

9.1 Implementation Analysis

When the firing rates of TrueNorth neurons saturate, the actual outputs of the Hopfield linear solver algorithm may no longer match the expected output; in fact, the difference may be quite large. However, for a large enough input scaling factor, the firing rates of neurons will be low enough so that they will never saturate.



(a) Plot for fraction of neurons that are saturating vs scaling factor for case 1

(b) Plot for fraction of neurons that are saturating vs scaling factor for case 2

(c) Plot for fraction of neurons that are saturating vs scaling factor for case 3

Figure 9.1: Comparison of scaling factor for different matrix structures

We refer to cases 1, 2, and 3 in Table 9.1 for range analysis. Figures 9.1(a), 9.1(b), and 9.1(c) show the plots for number of neurons that are saturating at maximum frequency vs. the scaling factor that was assigned to normalize the values of input. The neuron firing rates were collected using the corelet filter API which is a part of IBM TrueNorth's corelet programming environment [4]. The firing rate of neurons was gathered for matrices A and B with different set of values, as shown in Table 9.1. The factor η (the scale factor bound

Table 9.1: Sample matrices for worked out examples

Case 1	Case 2	Case 3	Case 4	Case 5
$A = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}$	$A = \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix}$	$A = \begin{bmatrix} 0.1 & -0.1 & 0.2 \\ -0.2 & 0.1 & 0.1 \\ 0.1 & 0.4 & -0.1 \end{bmatrix}$	$A = \begin{bmatrix} 0.08 & 8 \\ -1 & 0.01 \end{bmatrix}$	$A = \begin{bmatrix} 0.8 & 1.25 \\ 1 & 0.00008 \end{bmatrix}$
$B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	$B = \begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & 1 \\ 1 & 1 & -1 \end{bmatrix}$	$B = \begin{bmatrix} -4 \\ 0.2 \end{bmatrix}$	$B = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
$\eta = 60$	$\eta = 20$	$\eta = 30$	$\delta_{\text{hardcoded}} = 0.025\%$ $\delta_{\text{spiking}} = 3.39\%$	$\delta_{\text{hardcoded}} = \text{N/A}$ $\delta_{\text{spiking}} = 0.8\%$

calculated in Section 4) proves that the computed values never saturate, irrespective of whether the computations are happening in the positive or negative domain. The bounds shown are high because the maximum value that every element in the matrix can have after the geometric series summation would be a multiple of σ_{min}^{-1}. If the matrix A contains elements with very small magnitude then the term σ_{min} will be small as well; as a result we get a larger scale factor. The scenarios where η is close to the desired bound is when all of the elements in a matrix are the same and each element has values of high magnitude, similar to Case 2 in Table 9.1 (Figure 9.1(b)).

Cases 4 and 5 of Table 9.1 show the comparison of absolute errors when the same matrices are given as inputs, where W_{ff} and W_{hop} are either hard coded on TrueNorth or are supplied as spike train inputs. As per case 4, absolute error for hardcoded weights ($\delta_{\text{hardcoded}}$) is less than spiking weights (δ_{spiking}) for same number of spike ticks. This is because hardcoding the weights gives us more control over precision when compared with spiking weights. In Case 5, $\delta_{\text{hardcoded}}$ cannot be computed because TrueNorth neuron's threshold parameter has a limited number of bits, so W_{ff} cannot be mapped onto the board

using the technique of Algorithm 2. This problem does not occur with the spike train representation, as higher precision can be represented with longer duration.

9.2 Application analysis

This section presents the results and discussions for the applications that were tested on TrueNorth using Hopfield linear solver. Table 9.2 shows the amount of hardware that was utilized to map the relevant Hopfield linear solver for the corresponding applications. Note that, optical flow requires considerable number of more neurons than the other two applications. This is because for optical flow, rather than having hard-coded weights on TrueNorth, we chose to implement matrix multiplication by using neurons as multipliers. Since one neuron is being used to multiply two spike based inputs, the amount of required hardware grows quickly. We report the relative error (eqn. 8.10) and absolute error (eqn. 8.11) between the output that is obtained from the TrueNorth-based Hopfield linear solver and the output that is obtained using MATLAB's double precision pseudoinverse library function. A lower absolute error would be preferred as it would indicate how close is the predicted value from the ideal value in terms of precision. The result tables also summarize the number of ticks it took to compute the generalized inverse on TrueNorth.

9.2.1 Target Tracking

In this section the precision errors for target tracking application is reported. The setup for the experiment has been described in section 8.2.1. Matrix A was fixed during the entire simulation, but matrix B changed its values randomly. The simulations were done for 100

Table 9.2: TrueNorth hardware utilization

Application	Number of neurons used	Number of cores	Amount of on-chip cores utilized (in %)
Target tracking	288	2	0.05
Inverse kinematics (Hard-coded weights)	288	2	0.05
Inverse kinematics (Spiking weights)	160	11	0.2
Optical flow	712	11	0.2

different random feature values of matrix B. Table 9.3 presents the error that is seen for image scaling parameter, specifically the height and width of the bounding box, table 9.4 reports the error in calculated x-coordinates of bounding box and lastly, table 9.5 reports the error in computed y-coordinates of bounding box. It can be inferred from the reported results that if the computations are carried out for longer duration of time ticks, the error reduces.

Table 9.3: Error in reporting bounding box scale (width and height)

Number of clock ticks	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error(in cms)	Standard deviation absolute error(in cms)
3000	10.67	18.94	0.1647	0.1943
5000	4.14	9.89	0.0825	0.13
10000	2.96	4.78	0.0736	0.098

Note there is one anomaly in the results and that is for computing x-coordinate of bounding box after the simulation runs for 10,000 clock ticks. The relative error for this case increases instead of decreasing. This happens because for the cases where ideal x-

Table 9.4: Error in reporting bounding box horizontal position (x-coordinate)

Number of clock ticks	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error(in cms)	Standard deviation absolute error(in cms)
3000	5.92	21.26	0.1039	0.102
5000	1.13	2.09	0.049	0.0559
10000	6.88	34.16	0.039	0.0379

Table 9.5: Error in reporting bounding box vertical position (y-coordinate)

Number of clock ticks	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error(in cms)	Standard deviation absolute error(in cms)
3000	1.72	4.83	0.0922	0.0960
5000	1.69	3.23	0.052	0.0703
10000	0.74	2.29	0.0443	0.0392

coordinate is supposed to be 0, the Hopfield solver computes a value of 0.02 or 0.03, as a result, showing a relative of 200% or 300% for just these cases. Therefore, the reported relative error increases, but the absolute error is still small.

Fig. 9.2 shows the bar plot of the error in estimating the affine transformation matrix in target tacking application. These error plots represent the affine transformations that were computed using TrueNorth hardware over a period of 5000 ticks and were later compared with MATLAB's double precision pseudoinverse function for the same set of input matrices. Y-axis shows the error in estimating the affine transformation and X-axis the precision by which different image parameters (scale, x-shift and y-shift) changed in the developed target tracking application.

From fig. 9.2(a) we can observe that the neuromorphic implementation of the proposed

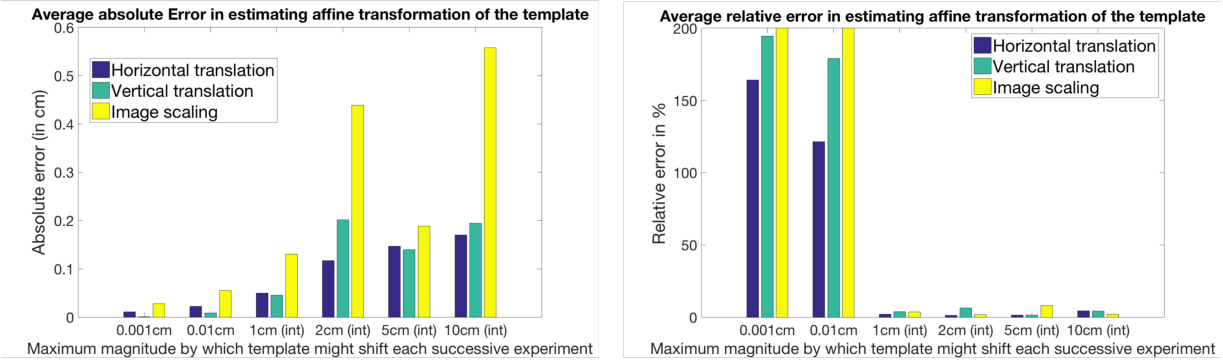


Figure 9.2: Error plots of the estimated affine transformation matrix in target tracking application. The TrueNorth based affine transformations were computed over a period of 5000 ticks and were later compared with MATLAB's double precision pseudoinverse function for the same set of input matrices.(a) Average absolute error for the estimated affine transformation. (b) Average relative error for the estimated affine transformation

Hopfield linear solver has very small absolute error (< 0.1 cm) when the affine transformation changes are very small in the proposed target tracking application; whereas, the absolute error is quite high (> 0.5 cm) when the affine transformation changes are considerably more. But the relative error shows an opposite trend when compared with absolute error trend. Similar to the discussion presented earlier, the relative error is very high when the affine transformation changes are very small because even though the estimated result is very close to the double precision result, the relative difference is very high. The relative error can be reduced significantly if we run the experiments for a longer duration (as inferred from table 9.13).

9.2.2 Inverse kinematics

To evaluate the results of experimental setup for inverse kinematics (as described in section 8.2.2), we chose to setup two different implementation schemes. In the first implementation technique the feedforward and recurrent weights Hopfield linear solver are

hard-coded on TrueNorth, whereas in the second implementation technique the weights were represented as spikes.

Hopfield network weights hard-coded on TrueNorth

Similar to target tracking application, this experimental setup was tested for 100 different random positions of the arm. The simulations were done for 60,000 clock ticks for each step. Unlike target tracking, here the computations have to be done on high-precision values, as a result, the algorithm takes longer to converge and requires a larger time window to represent very small values as spikes. Also, the matrices A and B change for every set of simulations that were conducted. Table 9.6 summarizes the results that were obtained following the experiments with inverse kinematics setup.

Apart from the results shown in table 9.6, we note that the Euclidean distance between the end effector position of the arm and the intended target position has exactly the same value when compared between MATLAB's pseudoinverse function and the Hopfield linear solver. That is, despite using an approximate computing technique with several sources of intermediate error, the effector always reaches the correct position irrespective of whether the computations were done using Hopfield solver or they were done with double precision MATLAB pseudoinverse function. These values were checked for 60 different target positions.

Error analysis of inverse kinematics with hard-coded weights on TrueNorth

Fig. 9.3 shows the relative and absolute error plots in estimating the position of the robotic arm along horizontal or vertical directions. Y-axis shows the % error in estimating the

Table 9.6: Error in reporting end effector positions

Attribute	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error (in cms)	Standard deviation absolute error (in cms)
Horizontal position of end effector	8.38	34.32	0.0886	0.076
Vertical position of end effector	49.39	83.1	0.357	0.3

position of robotic arm along x or y direction and X-axis represents the precision up to which the robotics arm motion were changing. The blue bars represent relative error %, whereas the yellow bars represent absolute error %. Note that we are computing error at each position where we update the vector Θ of eqn. 8.5. So until the arm reaches its desired position, after every update the error keeps on getting accumulated. As a result, if there are multiple positions where we make an error in estimating the vertical position of the robotic arm, all of these error get added up.

The error plots for estimating horizontal position of the end effector (as shown in fig. 9.3a) has a trend that is similar to target tracking error plots of fig. 9.2. In experiment setups where the x shift is small in magnitude (around 0.001 cm or 0.01 cm), the relative error of the estimated result is high when compared to absolute error in the same experiment. Whereas for x shift values that are comparatively larger in magnitude (5cm or 10cm), relative error is small but the absolute error is high.

On the contrary, the error plots for estimating vertical position of the end effector (as shown in fig. 9.3b) has a different trend. Similar to all of the other previous experiments we can see as the magnitude of shift increases (from 0.001 cm to 10 cm), the % absolute error

increases (yellow bar in fig. 9.3b). Absolute error % for estimating the vertical position is similar to absolute error % for estimating the horizontal position. There is very small difference in terms of absolute error % values. But, the % relative error is consistently high for all of the experiments. This is because of the error in estimating the vertical position for the scenarios where robotic arm is closer to the x-axis (y-axis component is < 0.1cm). Even though the estimated position is very close to the actual result, because the y-axis component is very small, it takes multiple steps to reach the desired position, as a result, because of these multiple steps of error, relative error % gets added up and becomes high.

As stated earlier, despite using an approximate computing technique with several sources of intermediate error, the effector always reaches the correct position irrespective of whether the computations were done using Hopfield solver or they were done with double precision MATLAB pseudoinverse function.

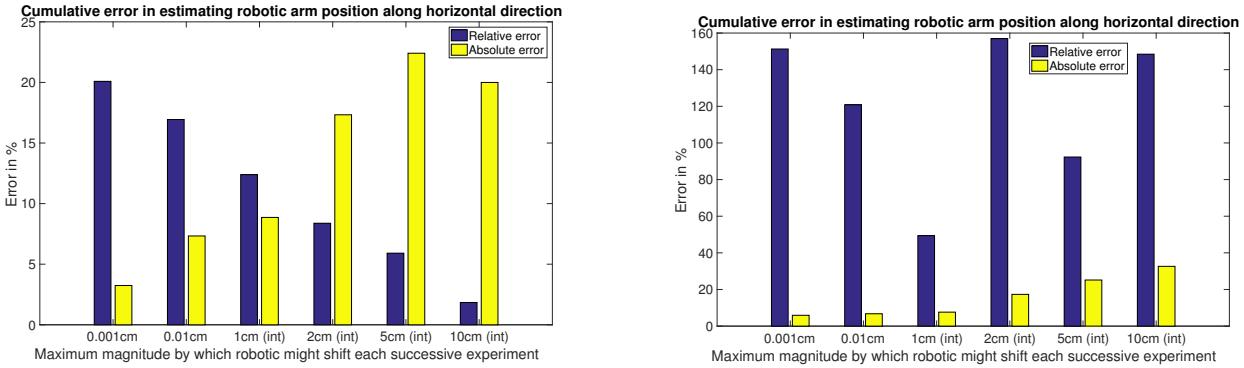


Figure 9.3: Y-axis shows the % error in estimating the movement of robotic arm along x or y direction and X-axis represents the precision up to which the robotics arm motion were changing.Fig. 9.3a % error in estimating the horizontal position of robotic arm. Fig. 9.3b % error in estimating the vertical position of the robotic arm

Hopfield network weights encoded as spikes

This experimental setup is similar to optical flow experiment and it was tested for 100 different random positions of the arm. The simulations were done for over 3 million clock ticks for each step. As the Hopfield solver weights are represented as spikes, it requires longer clock ticks to reach an answer that is close enough to the expected result.

In addition to the results shown in table 9.7, we note that the Euclidean distance between the end effector position of the arm and the intended target position has an average error of 0.0069 cms when compared between MATLAB's pseudoinverse function and the Hopfield linear solver. These values were checked for 20 different target positions. Even though the results show high relative error for vertical position of robotic arm, the end effector reaches very close to the intended position (an average difference of 0.0069 units) irrespective of whether the computations are done using Hopfield linear solver or MATLAB's pseudoinverse function.

Table 9.7: Error in reporting end effector positions

Attribute	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error (in cms)	Standard deviation absolute error (in cms)
Horizontal position of end effector	10.81	35.31	0.1263	0.3799
Vertical position of end effector	137.67	82.42	0.0978	0.1885

9.2.3 Optical flow

Experiments for optical flow application differ from the previous two applications. As the weight representation scheme chosen for these experiments is stochastically-code spike based, the simulations need to run for a considerably longer duration to converge to a solution. The purpose for such a computation scheme is that the proposed Hopfield linear model has the potential to be used as an online pseudoinverse calculator where the TrueNorth neurons could be used as arithmetic operation units. Table 9.8 summarizes the precision error that were obtained when the optical flow matrices were simulated for 3 million clock ticks for 100 different randomly chosen values from frame number 1 and frame number 12. While selecting the matrices A and B we ensured that the ideal output after matrix inverse should not have both the velocities as 0. Otherwise, the results would report incorrect observations for regions where there is no motion present. Similar to inverse kinematics testing, the A and B matrices for this application were tested in every set of simulations that were carried out. The results never gave the wrong prediction in terms of direction of motion or the velocity. In fact, the signs converged to correct values very early during the simulations. Table 9.8 summarizes the results that were obtained following the experiments with optical flow setup (as described in section 8.2.3).

9.2.4 Application analysis summary

This section presented precision and latency achieved for three different applications that were analyzed using TrueNorth based linear solver implementation. This section presented the hard-coded implementation style proposed in section 5.2.1 for object recognition

Table 9.8: Error in reporting magnitude of velocities for optical flow

Attribute	Mean relative error (in %)	Standard deviation of relative error (in %)	Mean absolute error(in cms/sec)	Standard deviation absolute error(in cms/sec)
Magnitude of horizontal velocity	18.39	36.56	0.1649	0.5485
Magnitude of vertical velocity	7.65	18.27	0.2057	1.0

application and showed that users can achieve mean absolute error that is as low as 0.0886 cms. Similarly, we also presented a stochastic computing style implementation of section 5.2.2 for optical flow and showed that we can achieve mean absolute error as low as 0.2 cm/sec in estimating the motion of object. The common issue that we observe across all three applications is that to estimate low-precision results we may need a longer computation time. This observation is evaluated empirically later in section 9.7 for TrueNorth based optical flow implementation.

9.3 Architecture-Application Analysis

9.3.1 Motivation: Comparison of TrueNorth with Standard Matrix Inversion Approach

Analysis presented in this section is meant to understand the strengths and weaknesses of implementing a stochastic computing based linear solver on TrueNorth when compared to more standard approaches. The evaluations presented in this section serves as the

motivation for architectural changes that were proposed in chapter 6 and chapter 7.

9.3.2 Proposed linear solver vs QR-inverse implementation

Prior work [91] showed how these linear solvers can be used to compute transformation matrices for applications such as inverse kinematics, object tracking and optical flow. To analyze the proposed Hopfield linear solver in a practical implementation scenario, we tested the proposed work for Lucas-Kanade based optical flow application that has a setup similar to the one shown in fig. 8.3 and described in the section 8.2.3. The image in fig. 8.3 is a grayscale image in which high intensity pixels are represented with a value of 1 and low-intensity pixels are represented with 0. The two black bars in the figure have a pixel width of 5 pixels. The resolution of the image was set at 240-by-320 pixels which is same as QVGA format videos. The horizontal and vertical bars were initially positioned at the center along height and width of the image, respectively. The two lines intersected at the center of the image. The sequences of images are streaming in to the hardware at 30 frames per second. For the first set of frames, the horizontal bar is moving upwards, and the vertical bar is moving towards left. In the implementation, the frame size of QVGA video was first reduced by a factor of 4 to 120-by-160 pixels, then a 5-by-5 pixels convolutional operation was applied to it.

The implementation of a Hopfield linear solver in such a setup is challenging since the Hopfield neural network (W_{ff} and W_{hop}) weights change continuously . Also, in this setup there is no training or testing data involved. The goal here is to compute the results online by just looking at the streaming input values without any prior knowledge of the

experiment or scenario. We observe additional benefits by deploying multiple linear solvers in parallel since we have to calculate pseudoinverse for multiple different locations on the image at the same time. These experiments give us better insights with respect to selecting TrueNorth as a potential substrate for deployment of such algorithms, and provides a vehicle for energy analysis when compared with more traditional approaches. In this experiment we measure the motion vector error against the baseline, but have also utilized an approximately correct metric: as long as the solver correctly detects flow in one of eight possible ordinal and cardinal directions, we count it as correct. The velocity of the movement of two bars is calculated by solving for X in the equation $AX = B$. Matrix A contains partial derivatives of initial image frame with respect to directions x and y around pixel qi. This is represented by terms $I_x(qi)$ and $I_y(qi)$, in equation 8.8. Matrix B contains partial derivatives of pixel positions between initial image frame and image frame at time t around pixel qi. This is represented by terms $I_t(qi)$, in equation 8.9. After implementing matrix division, output matrix X will report the speed and direction of the image pixels, by computing the pseudoinverse of matrix A.

In the proposed setup, we can have multiple input matrices A and B (see (2.5)), that are independent of each other, since the convolution operation can operate on separate and independent patches of image at the same time. The results of these independent convolutions can be streamed as different input matrices A and B. As a result, we can have multiple independent linear solvers running in parallel to compute different pseudo-inverses for these different input matrices. For a frame of size 120-by-160 pixels, linear solver implementation processed 9800 pixels of a single frame to predict the motion vectors.

Using the optical flow implementation described above, we compare the power and

energy consumption of TrueNorth based linear solver implementation with more traditional approaches like QR inverse algorithm on Virtex-7 FPGA (xc7vx980t) and on an ARM cortex A15 mobile processor.

On TrueNorth we can implement 392 instances of the Hopfield linear solver that operate in parallel independent from one another. These 392 instances required 4092 cores of the available 4096 cores and can process roughly 9800 pixels for predicting the motion vectors. Therefore, we would need to compute optical flow motion vectors in the specified scenario in batches of two streaming input pixels for a single 120-by-160 pixels frame.

To maintain the throughput of 30 FPS for 9800 pixels we needed an 8-core ARM chip operating at 2.5 GHz. For the same FPS and pixel count we had to instantiate 32 instances of QR inverse algorithm on Virtex-7. A detailed discussion about each of the implementation technique is presented as follows:

TrueNorth: We have implemented 392 instances of the Hopfield linear solver which operate in parallel, independent from one another. These 392 instances required 4092 cores of the available 4096 cores. The power consumption values were reported from IBM's test and development board. For a supply voltage of 0.8 V and 1KHz operating frequency, the scaled leakage power of our implementation is 46.31 mW and the scaled active power 18.67 mW. Since the goal is to implement optical flow at 30 FPS, we increase the operating frequency of TrueNorth NS1e hardware to 9KHz and report a linearly scaled active power of 168.03 mW for these experiments.

Virtex-7 FPGA: The QR inverse algorithm was implemented using the matrix algebra libraries present in Xilinx Vivado HLS [103] and the frequency of the platform was set at 20 MHz. Power analysis of the following implementations were done using Xilinx

Power Estimator tool [104]. For 32 parallel instances of QR inverse solver the total power consumption is 1.881W with a static power consumption of 383 mW.

8-core ARM A15 processor: The QR inverse algorithm was implemented using the C++-based eigen library [33]. The simulator setup of ARM processor has been described in section 8.3. For processing the QR inverse algorithm, the combined 8-cores of ARM Cortex A-15 chip consumed 6 W of power which includes 93.5 mW of static power.

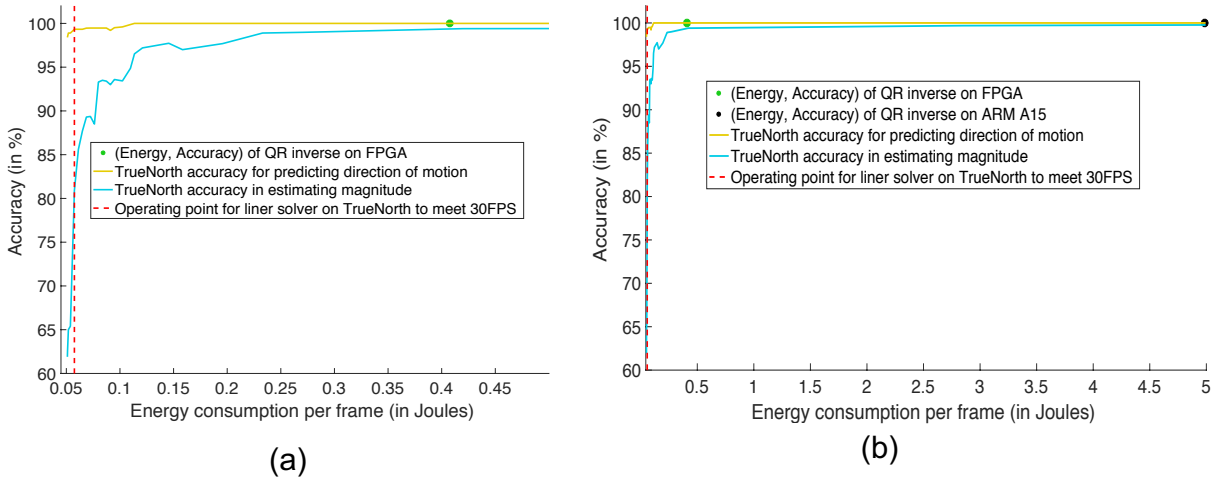


Figure 9.4: This figure shows a comparison between three different implementation techniques for matrix inversion. Y-axis of the plot shows the percentage accuracy in predicting the motion of bars for optical flow. And, X-axis of the plot shows the energy consumed per frame (in Joules) for optical flow. (a) Comparison of power consumed between FPGA and TrueNorth hardware. (b) Comparison of power consumed between ARM, FPGA and TrueNorth hardware.

Figure 9.4 shows the comparison of energy consumption and time elapsed for computation on three different hardware platforms. Since TrueNorth can perform computations on 9800 pixels at a time, it would have to time-multiplex 120-by-160 pixels into two batches to perform the linear solver operation. At 9KHz frequency each time multiplexed batch would need a maximum of 150 time ticks for computing the inverse on a portion of the

image. After 150 ticks, the accuracy of predicting the direction in optical flow is 99.33% and the speed of motion can be estimated with an accuracy of 80.9%. As per the plots in Figure 9.4 (a) and (b), the TrueNorth-based linear solver is more energy efficient than the ARM or FPGA implementations. For both the FPGA- and ARM-based QR inverse solvers, the accuracy is 100% as they are using floating point units for computation. The TrueNorth-based linear solver consumes 0.0575 J of energy per frame, the FPGA consumes 0.4074 J of energy per frame and, the ARM processor consumes 4.986 J of energy per frame. On the other hand, the accuracy of TrueNorth depends on how many ticks it requires. As a result, if TrueNorth is operated for more ticks, the solution achieves higher accuracy but consumes more energy.

9.3.3 Proposed linear solver implemented on TrueNorth vs Xilinx ZedBoard

Figure 9.5 shows the comparison of energy consumption and time elapsed for performing stochastic computing based matrix inversion when implemented on TrueNorth and Xilinx ZedBoard. As stated earlier, we observe additional benefits by deploying multiple linear solvers in parallel since we have to calculate pseudoinverse for multiple different locations on the image at the same time. TrueNorth based Hopfield neural network implementation is the same as we had described previous in subsection 9.3.2. In contrast, 262 stochastic computing based Hopfield linear solvers can be deployed in parallel on a Xilinx ZedBoard. The static power of a Xilinx ZedBoard is 120mW and the peak dynamic power is 115mW while operating at 20 MHz frequency.

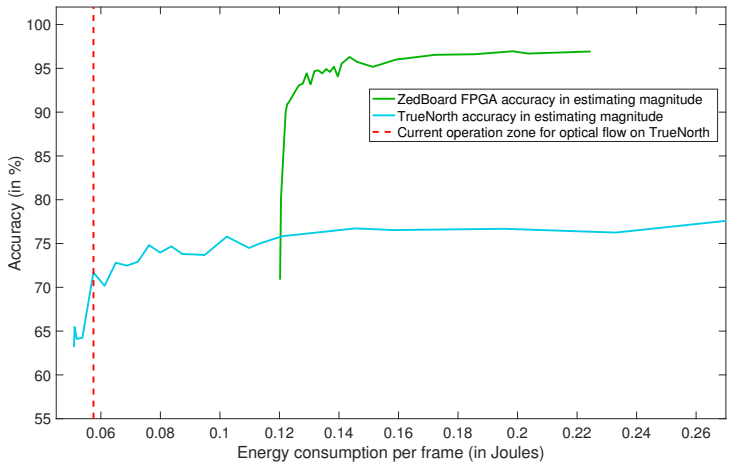


Figure 9.5: This figure shows an energy comparison of optical flow matrix inversion application that was implemented on TrueNorth and Xilinx ZedBoard.

Because the FPGA board is operating at a much higher frequency (when compared to operating frequency of TrueNorth being 9 KHz), it is able to achieve better accuracy much faster. But due to the difference in static power present between the two hardware substrates (120mW of FPGA and 40 mW in TrueNorth), the stochastic computing based FPGA implementation is not able to achieve energy consumption that is lower than 0.120 Joules. These results motivated us to deploy these stochastic computing architectures on hardware substrates such as Lattice FPGAs, that can operate at a higher clock frequency and consume much lower energy.

9.3.4 TrueNorth Performance Summary

It can be observed from results plotted in fig. 9.4, if TrueNorth based linear solver is ready consume as much energy as the 8-core ARM A15 based QR-inverse implementation, the accuracy of this stochastic computing based linear solver gets very close to 100%. But

there is still a considerable gap in terms (energy, accuracy) between a QR-inverse algorithm implemented on Virtex-7 FPGA and TrueNorth based linear solver. We did try to overcome this difference in performance by implementing the proposed linear solver on an FPGA that can operate at a higher clock frequency of 20 MHz (refer to fig. 9.5). Eventually, the goal of our algorithm would be to achieve better accuracy or lower loss in fewer number of time ticks, without having to use a hardware that operates at higher clock frequency. As we will see later, this can be achieved with population coding scheme in linear solver implementation (refer to evaluations presented in section 9.4).

9.4 Population coding results

As discussed in section 3.1.3, one of the challenges that we face with stochastic computing is the issue of latency [35]. Taking inspiration from neuroscience we proposed a population coding architecture in chapter 6 that would implement multiple stochastic linear solvers in parallel and cut down on computation latency to achieve the desired final accuracy. This section provides experimental results that support this analysis as shown in Fig. 9.6. For this result, we evaluated varying population sizes of an Hopfield neural network based linear solver with random input data as described in section 8.1. Each data point represents the average loss across all simulations at that time step. Notice that the population coded implementation for 2 and 5 populations approaches the same loss as the conventional implementation in approximately 1/2 and 1/5 the time. But, as we approach 100 populations, there is no reduction in latency compared to 20 or 50 populations. This is because the algorithms are iterative, so they require a minimum number of ticks to converge. Addition-

ally, if we are willing to consume more energy, we can increase the overall accuracy of an application by instantiating more populations without increasing the overall runtime (see Fig. 9.5).

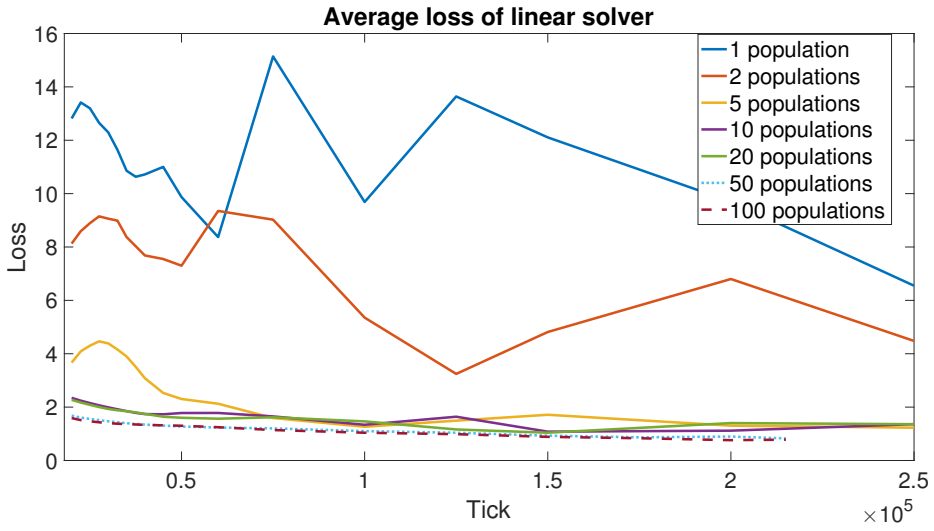


Figure 9.6: Average loss for linear solver for varying populations.

9.4.1 Population Coding Speedup

In this section we will analyze by how much amount do the population coded linear solvers achieve speedup when compared to a single instance (population count 1) based naive approach. In fig. 9.7 we carried out the linear solver simulations for longer number of time ticks for population counts of 1 and 2. To quantify the amount of speedup that is achieved with population coding scheme, we selected the same loss value for different linear solver implementations (as shown by solid red line in fig. 9.7) and measured number of ticks it took to reach that loss.

Fig. 9.8 shows the amount of speedup was achieved with population coding scheme when compared with a single instance implementation of a linear solver. As explained

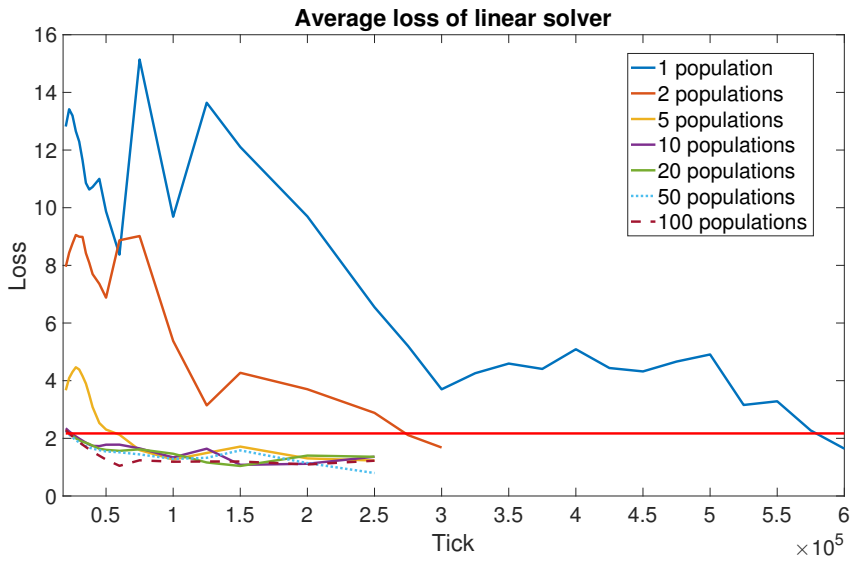


Figure 9.7: Average loss for linear solver for varying populations. Simulations for population counts of 1 and 2 were carried out for more number of time ticks to quantify the amount of speedup which we achieve with population coding scheme.

earlier, the same loss value was selected for all of the different linear solver implementations (as shown by solid red line in fig. 9.7) and measured number of ticks it took to reach that loss. Unsurprisingly, as we kept on increasing the population count we see an exponential increase in speedup compared to single instance based implementation. It can be observed from the plots shown in fig. 9.8, for a population count of 20, we achieve a speedup of 25.56x. As we keep on increasing the population to 50 and 100, we do not enjoy from a similar trend of speedup increase. This is because the algorithms are iterative, so they require a minimum number of ticks to converge.

9.4.2 Population Coding Analysis

This presented an empirical evaluation of benefits of population coding scheme for linear solver. It can be observed from plots in figures 9.6 and 9.7 having multiple linear solvers

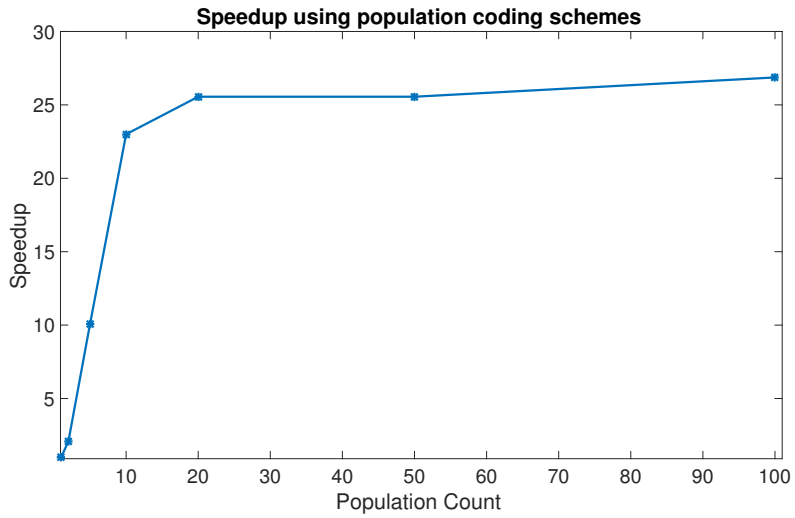


Figure 9.8: Speedup achieved with different population counts when compared with a single instance implementation of a linear solver.

operating in parallel and later combining their results allows us to achieve a smaller loss in shorter number of time ticks. As per the results shown in fig. 9.8, we can achieve a speedup of 25.56x with a population count of 20 when compared to a single instance implementation of linear solver. But, as we keep on increasing the population count we do not enjoy from a similar trend of speedup increase. This is because the algorithms are iterative, so they require a minimum number of ticks to converge.

The energy and power tradeoffs of stochastic computing based implementations on an FPGA are presented in section 9.5. This section complements the results shown in fig. 9.6 and presents the trade-offs and benefits that users can achieve with stochastic computing when compared to baseline implementations using HLS language.

9.5 Hardware Analysis

9.5.1 Area Results

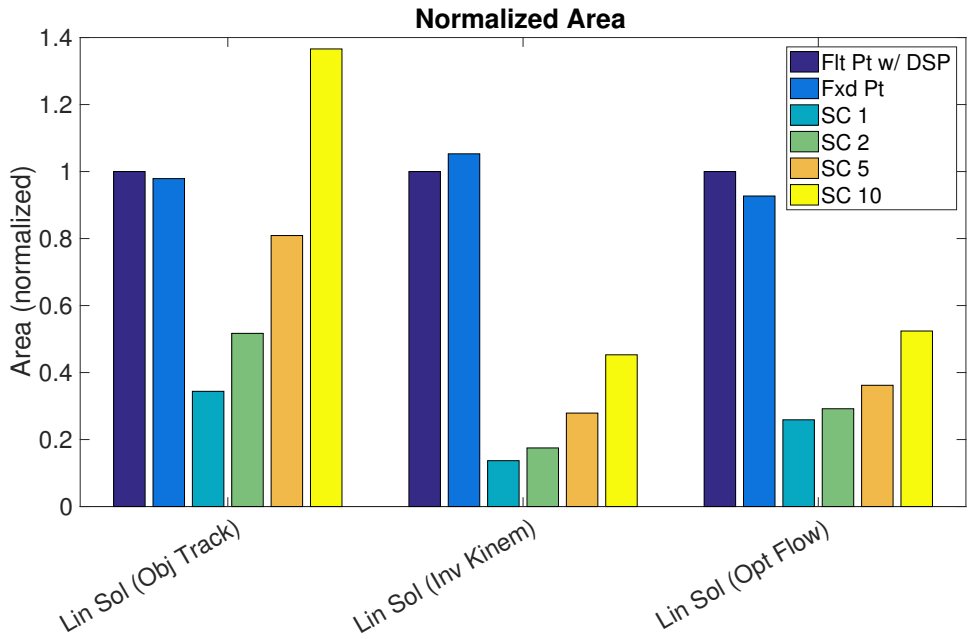


Figure 9.9: Normalized area consumption relative to floating implementation. Area is computed as # of LUTs + # of FFs. Normalized area does not include DSP units for floating point implementation. "SC n" indicates a stochastic computing implementation with n populations.

As shown in Fig. 9.9, SC implementations of each algorithm consume less area compared to floating point or fixed point implementations. For most of the linear solver SC implementations, the overall area is still lower than the floating point or fixed point implementations despite instantiating multiple populations. This is due to the large dimensionality of the inputs to each population. Each input is a matrix, and each element in the matrix requires its own decorrelator per population. Decorrelator units consume a large number of flip-flops which contributes significantly to the overall area of the SC implementations. Section 8.3

has explained the fixed-point implementation setup for hardware analysis. Due to the difference in size of matrices for different applications, we use different lattice FPGA family models for object tracking application and the other two applications. For object tracking application we use a lattice FPGA from LP8K model, whereas for optical flow and inverse kinematics applications we use LM4K lattice FPGA model.

It can be observed in Fig. 9.9, population count of 5 and 10 of SC object tracking implementation requires more LUT and FF count when compared to the same parallel implementations of inverse kinematics and optical flow applications. This is because the transformation matrix X_k has a dimension of 3x3 for object tracking, whereas for inverse kinematics and object tracking, the transformation matrix has a dimension of 2x1. That is, there are 4.5 times more values present in X_k of object tracking when compared to the same matrix of inverse kinematics and object tracking. When we start increasing the number of populations for SC implementations, the resource requirements grow proportionally to the dimension of X_k. Therefore, we can see that for population counts of 5 and 10, where dimension of X_k becomes significant in resource consumption, the number of LUT and FF counts for object tracking increases when compared to LUT and FF counts for inverse kinematics and optical flow.

9.5.2 Power and Energy Results

As discussed in Sec. 8.2.5, all designs are required to meet a 35 mW power budget. This is plotted as a dashed red line in Fig. 9.10. For all algorithms, the floating point and fixed point FPGA implementations fail to meet the power budget. This is largely due

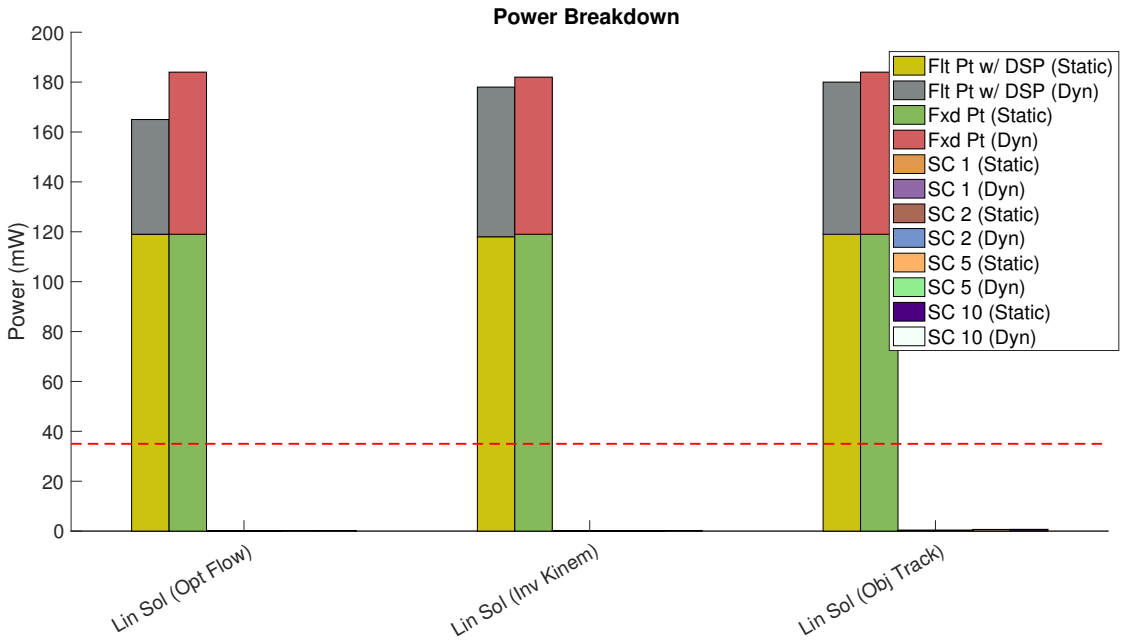


Figure 9.10: Power consumption for different implementations of the linear solver on FPGAs. This figure shows the comparison of power consumption between floating point, fixed point and SC based linear solvers for three different applications. "SC n" indicates a stochastic computing implementation with n populations. The red dashed line indicates the 35 mW power budget for the RoboBee. Refer to fig. 9.12 for energy plots for different FPGA based linear solver implementation.

to their heavy resource requirements which prohibit them from being mapped to ultra low-power FPGAs. Fig. 9.10 also illustrates that dynamic power is a more significant percentage of the total power in the floating point and fixed point implementations than the SC implementations (see Fig. 9.11). So, even if we were to partition the floating/fixed point designs across multiple low-power FPGAs to reduce the static power consumption, the HLS designs would still suffer a greater overall energy consumption due to their high dynamic power. For reference, our optical flow implementation would consume 375.3 uW of dynamic power and 1.14 uW of static power at a 40 nm technology node (which is considerably less than the required 35 mW power budget [22] [26]).

In all of the SC based implementations, a single instance of SC linear solver can fit on

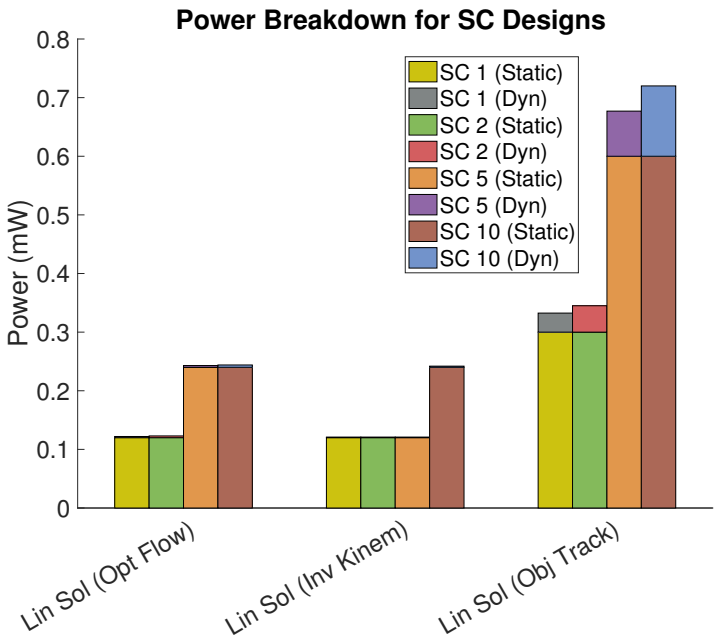


Figure 9.11: Power consumption for different implementations of the linear solver on FPGAs. This figure shows the comparison of power consumption between SC based linear solvers for three different applications. "SC n" indicates a stochastic computing implementation with n populations. Refer to fig. 9.12 for energy plots for different FPGA based linear solver implementation.

an ultra low power lattice FPGA. But as we keep on increasing the population, some of the proposed population coding setup may not fit on a single FPGA chip. Because of the regular compute that is present in population coding scheme, we can use multiple FPGA chips to map higher population count. As a result, there is an increase in static power for SC implementations that have higher population count. This behavior is evident in inverse kinematics application, when SC-10 requires more resources compared to SC-5. Therefore, we use two lattice FPGAs to implement SC-10 inverse kinematics application which explains an increase in static power (refer to fig. 9.11). Similarly, in optical flow application, when SC-5 required more resources compared to SC-2. Therefore, we use two lattice FPGAs to implement SC-5 and SC-10 population coded optical flow application which explains an increase in static power when moving from SC-2 to SC-5.

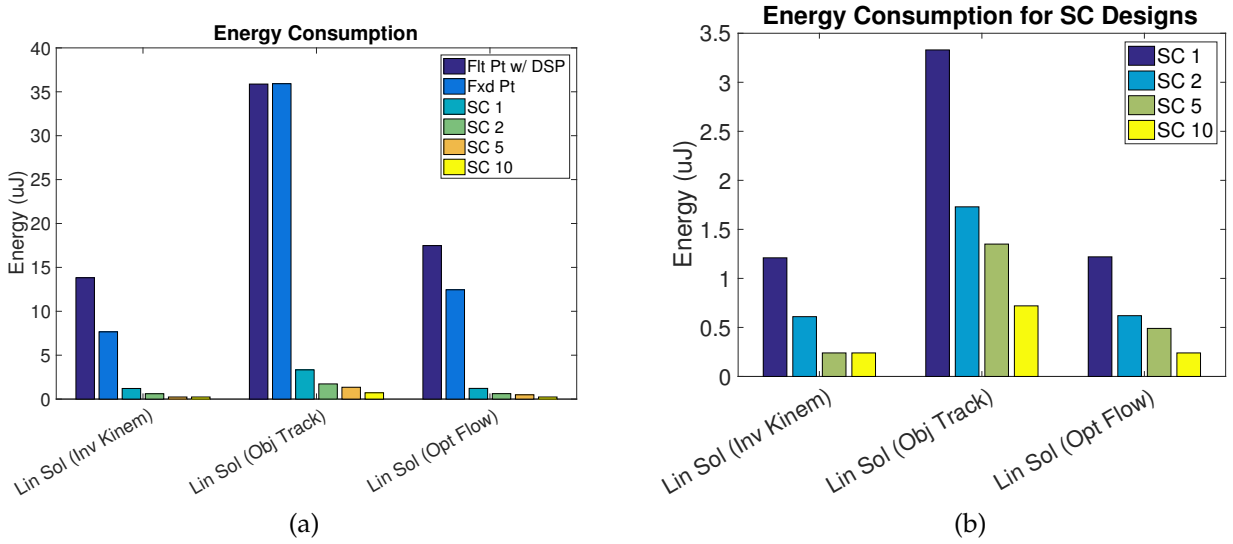


Figure 9.12: Energy consumption for different implementations of the linear solver on FPGAs. Fig. 9.12a shows the comparison of energy consumption between floating point, fixed point and SC based linear solvers for three different applications. Fig. 9.12b shows the comparison of energy consumption between SC based linear solvers for three different applications. "SC n" indicates a stochastic computing implementation with n populations. For comparison of power consumption between SC and baseline implementation techniques refer to figures 9.10 and 9.11

Despite running for a longer amount of time, all the SC implementations consume significantly less energy. Again, this is due to ultra low power consumption which is only possible because SC implementations consume very few resources. As discussed in section 9.4 one of the crucial advantage that we achieve from population coding approach is that we can cut down on the number of ticks to achieve a certain loss, by increasing the population size up to a certain point. Since population coding requires us to perform computations for shorter time duration, energy reduces with increase in SC population count (refer to fig. 9.12b). But reduction in energy is non-linear. For example, for inverse kinematics application, when going from SC-5 to SC-10, the energy consumption is the same because SC-10 requires two FPGAs for implementation whereas SC-5 requires only one FPGA for implementation. Therefore, static power of SC-10 is higher, hence, we do not

see any decrease in energy when moving from SC-5 to SC-10 for inverse kinematics.

9.5.3 Hardware Analysis of Population Coding Architecture

When an SC design consumes significantly low resources, multiple populations can fit on a single ultra low-power FPGA chip. This is the case for the linear solver implementations in Fig. 9.11. Since each population consumes < 1 mW of dynamic power, the total power remains effectively the same as populations increase, because static power dominates the total consumption. As illustrated in Fig. 9.12b, the linear solver enjoys an approximately linear reduction in energy. With stochastic computing based designs designers can achieve 7x area reduction and 275x reduction in power while achieving the same accuracy as fixed point implementation.

9.6 Adaptive Scaling Analysis

This section provides the analysis of stochastic computing architecture for adaptive scaling technique that was proposed in chapter 7,section 7.2. Table 9.9 lists the default values that were set for different parameters in the adaptive scaling architecture (as presented in fig. 7.1). Evaluation of the proposed architecture was done using the experimental setup and error analysis formula that was discussed in section 8.1.

Figure 9.13 shows how adaptive saturation can help us in getting better results faster compared to having a linear solver implementation where the input matrices scaled with a scaling factor parameter. Figure 9.13 shows the comparison between two different implementations of single instance of linear solver. The plot with blue colored line has its

Table 9.9: Parameter values of adaptive scaling architecture

Compute unit	Parameter value
Threshold 1 (Th1)	255
Threshold 2 (Th2)	258
RNG range	0-1024
Rate generator counter decrement value	4

input values were scaled by parameter η (as derived in eqn. 4.10), whereas, the plot with red colored line has implemented adaptive scaling architecture in the linear solver design. The experimental data was generated using the conditions explained in section 8.1 for 10 different randomly generated matrices.

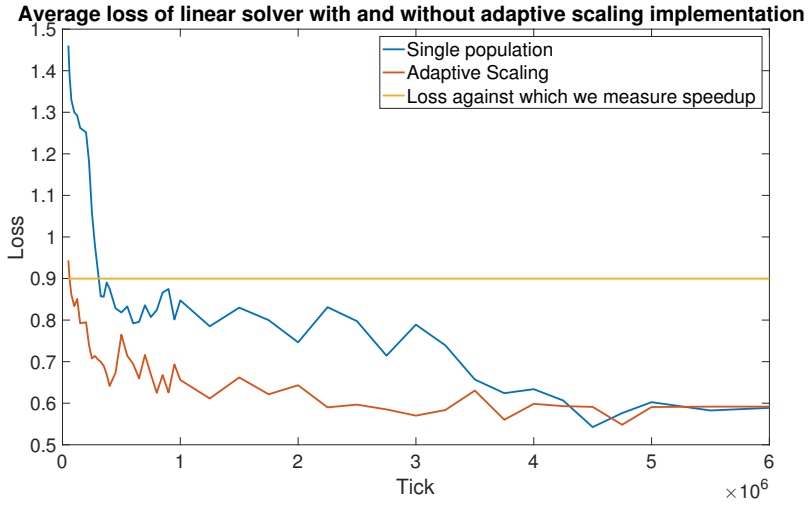


Figure 9.13: Average loss for linear solver with adaptive scaling implementation. This figure shows the comparison between average loss that was achieved in a linear solver where the input values were scaled by parameter η and a linear solver implementation with adaptive scaling architecture.

We can observe from fig. 9.13 that having an adaptive scaling implementation improves the speed of convergence. The yellow line in fig. 9.13 is meant to evaluate the number of time ticks it took to reach the same loss value with two different scaling factor approaches. As observed the adaptive scaling technique can achieve up to 5.4x speedup when compared

to having a conservative scaling factor value η.

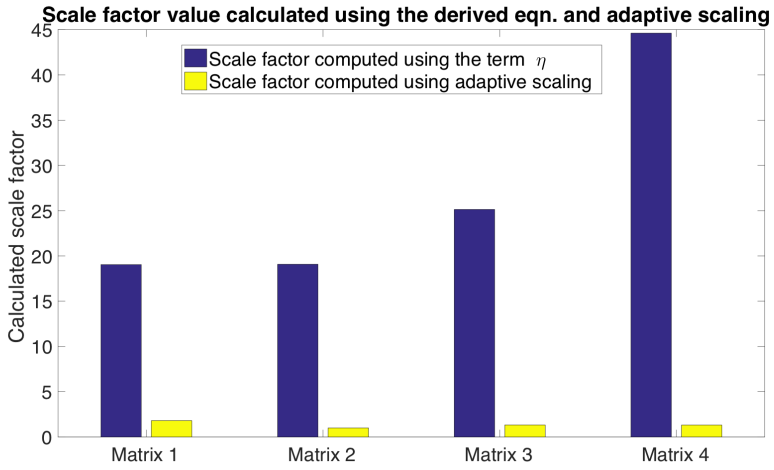


Figure 9.14: Calculated scaling factor for four different input matrices. Blue bars represent the scaling factor that was calculated using the conservative approach discussed in eqn. 4.10, whereas, yellow bar show the scaling factor values that were calculated using the adaptive scaling technique.

Fig. 9.14 presents an intuitive reason behind why adaptive scaling technique achieves smaller loss faster when compared to using a conservative scaling factor approach. The scale factors in fig. 9.14 were calculated after 10^6 time ticks for four different input matrices. An ideal scaling factor should not make the input values unnecessarily small. If the inputs become very small, it would require longer time ticks to represent small values accurately, as a result, it would take longer number of time ticks for the linear solver to achieve the desired accuracy. The adaptive scaling architecture addresses the issue of estimating scale factor values that are closer to 1. A scale factor that is close to 1, indicates that most of the input value bits are allowed to go into the linear solver. It can be inferred from the plots in fig. 9.14 that the adaptive scaling technique can estimate a scaling factor that is up to 33.41x smaller than the scaling factor estimated from eqn. 4.10. As a result, the input values get scaled by a reasonable amount and linear solver is able to achieve desired convergence

with fewer time ticks.

In the later part of this section we will present how the loss of linear solver gets affected when we vary different parameters of the architecture (as listed in table 9.9). It is crucial for users to select the correct set of parameter values for adaptive scaling because these values govern how much faster can the linear solver. Otherwise, if the user ends up selecting arbitrary parameter either the rate of convergence will become too small or the errors might accumulating, as a result, we will get incorrect result from the linear solver implementation.

Similar to our prior work on bit precision analysis, the matrix values were generated using the conditions as explained in section 8.1, and we repeated this experiments for over 25 random matrix values.

9.6.1 Experiments with decrement unit in rate generator

This subsection discusses how the average loss gets affected when we vary the value by which counter of rate generator unit gets decremented after overflow is detected. Table 9.10 lists out the different parameter values that were selected to decrement the counter value, and fig. 9.15 shows loss of the proposed linear solver over increasing number of ticks.

It can be observed from the plots that if the decrement value is low, such as, 2 or 4 (experiments 1 and 2 in table 9.10), the amount by which loss decreases over time is much slower because the adaptive scaling architecture has to keep adjusting itself multiple times to address saturation happening in compute units. On the other hand, if we set decrement parameter value to be too high such as, 256 or 512 (experiments 8 and 9 in table 9.10), the input values may end up becoming zero or too low, as a result, the loss increases over

Table 9.10: Different decrement values of rate generator counter

Experiment number	Parameter value
1	2
2	4
3	12
4	24
5	64
6	128
7	192
8	256
9	512

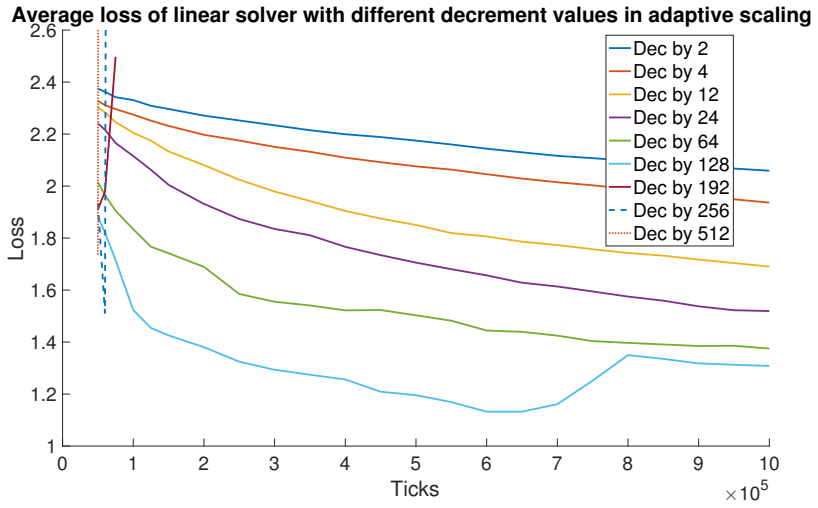


Figure 9.15: Average loss for linear solver when we vary the counter decrement value in adaptive scaling architecture.

time because the architecture would require longer time to converge and a very low rate generator frequency will result in a much larger scaling factor (refer to fig. 7.1).

Figures 9.16 and 9.17 explain the unusual behavior that appears in fig. 9.15 due to selecting high decrement values. The value of loss in fig. 9.16 and $(Scale_Factor)^{-1}$ in fig. 9.17 were calculated after 10^6 number of time ticks. Because of aggressively decrementing the counter value in rate generator unit, there are scenaiors when $(Scale_Factor)^{-1}$ may

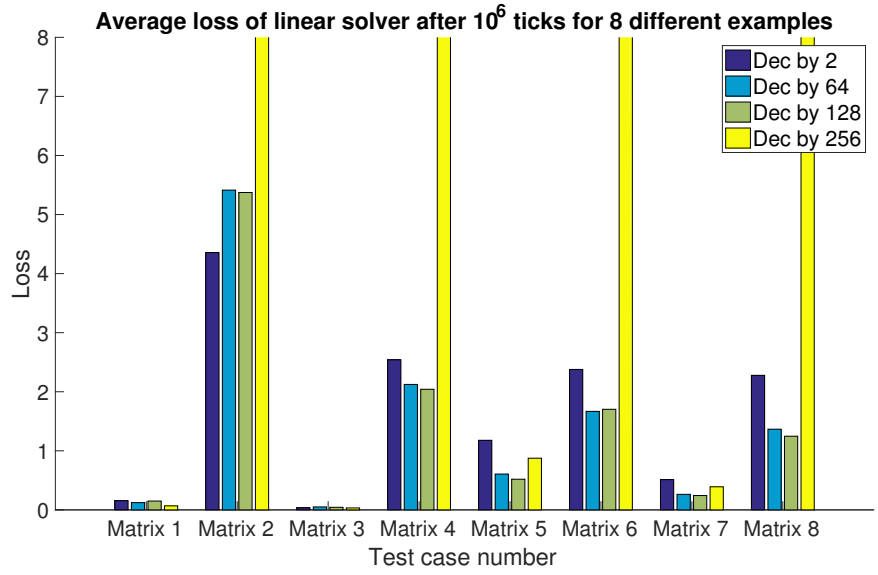


Figure 9.16: Average loss for 8 different input matrices with adaptive scaling architecture.

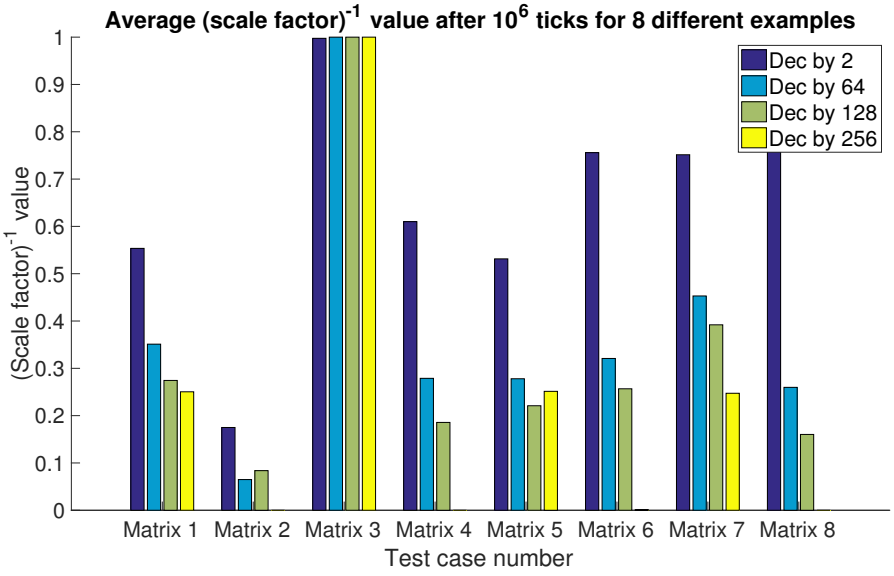


Figure 9.17: Computed $(\mathtt{Scale_Factor})^{-1}$ for 8 different input matrices with adaptive scaling architecture.

become 0 (Matrices 2, 4 and 6). As a result only stream of 0 bits come in as input values, hence, the loss for corresponding input matrices become high because linear solver is operating only on input matrices that have value 0. Therefore, when implementing any kind of adaptive scaling technique to iterative algorithms it is important to avoid selecting

extremely high values otherwise the scale factor may become zero and our linear solver may end up operating only on streams of 0 bits.

9.6.2 Experiments with different RNG range

Here we will discuss how the average loss gets affected when we vary the range of values that random number generator (RNG) can take. As explained in section 7.2, RNG and counter together are an integral part of rate generator unit (or neuron) of the adaptive scaling architecture because based on the value of counter and RNG range, the scaling signal is generated for the input matrices. Table 9.11 lists out the different range of values that RNG can have, and fig. 9.18 shows loss of the proposed linear solver over increasing number of ticks with respect to varying the range of values that RNG can have.

Table 9.11: Maximum range of values that random number generator (RNG) has

Experiment number	RNG range
1	0-1024
2	0-255
3	0-128

Based on the observations presented in section 9.6.1 and set of values that were defined in table 9.9, having a high range of values for RNG would mean that rate generator unit can have a fine-grained control over the amount by which input values can be scaled. But on the flip side, it would take longer to set the counter at an appropriate value so that the inputs get scaled to the proper value with fewer number of ticks. This analysis is corroborated in fig. 9.18, where an RNG that can have any value in the range from 0 to 1024 takes much longer to decrease its loss. On the contrary, an RNG with much smaller range like from

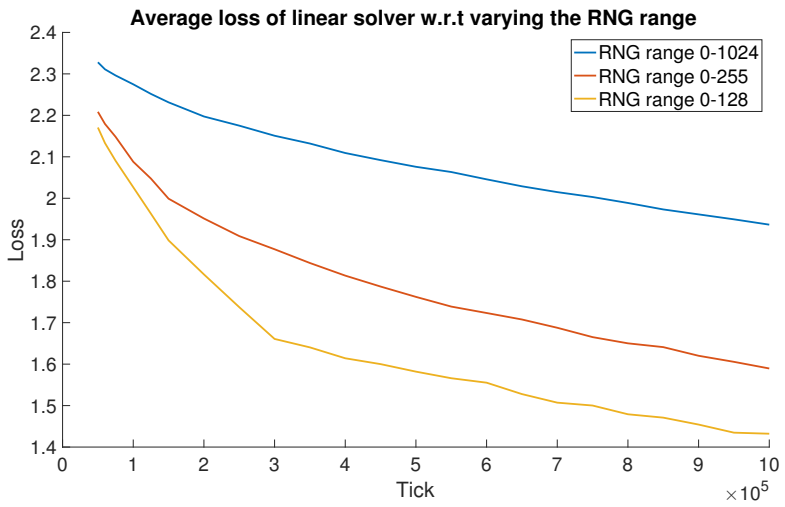


Figure 9.18: Average loss for linear solver when we vary the counter decrement value in adaptive scaling architecture.

0-16 decreases the rate generator frequency very fast, as a result, the loss (not shown in fig. 9.18) becomes very high in the initial number of ticks.

9.6.3 Experiments with overflow detector's threshold values

In this last analysis of adaptive scaling, we will look at how different threshold values for overflow detector affects convergence of the linear solver. As discussed in chapter 7, function of the overflow detector is to detect if there is any compute unit that has saturated and relay the result to rate generator unit. The stochastic computing logic design was presented in fig. 7.3(a). Table 9.12 shows the different threshold values that were selected for the overflow detector and fig. 9.18 shows how average loss changes over time with different threshold parameters.

It can be inferred from earlier discussions and fig. 9.19, a set of high threshold values indicate that it would take longer for overflow detector to detect if there are any compute units that are saturating. As a result, the adaptive scaling unit would have to adjust itself

Table 9.12: Different threshold values of overflow detector

Experiment number	Threshold 1 value	Threshold 2 value
1	255	258
2	128	130
3	64	66
4	32	34

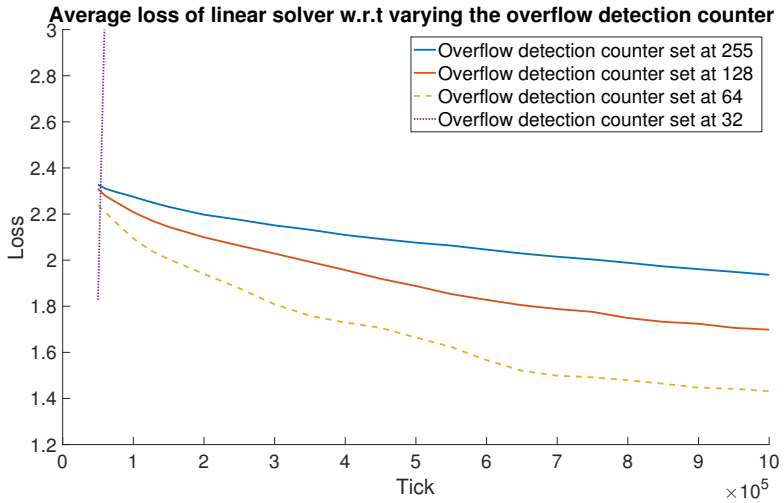


Figure 9.19: Average loss for linear solver with different threshold values of overflow detector in adaptive scaling architecture.

multiple times until the input values are scaled down by the appropriate factor. On the contrary, a set of low threshold values indicate that the overflow detector would let multiple volley of spikes to pass through if there are any compute units that have saturated. As a result, the counter in rate generator unit would decrement multiple number of times which would eventually lead to having higher average loss of the proposed linear solver.

9.6.4 Adaptive scaling analysis summary

This section presented the advantages of having an adaptive scaling technique for a single instance of linear solver instead of using a mathematical parameter η that was derived

in eqn. 4.10. Users can achieve up to 5.4x speedup with an adaptive scaling architecture when compared to an implementation where the input values had to be scaled. With an adaptive scaling architecture the estimated scale factor can be up to 33.41x smaller than the conservative value of parameter η. We also looked at the effect of change in loss after we select different design parameters. When designing these adaptive scaling architecture it is important for designers to select proper parameters listed in table 9.9 to ensure that the proposed Hopfield architecture converges to the desired solution within desired number of iterations. Designers should avoid assigning extremely high values or aggressive values to parameters in adaptive scaling architecture, otherwise, the calculated scaling factor may become very small or close to zero, which might result in rate generator unit inhibiting all of the input value bits.

9.7 TrueNorth convergence and precision analysis

A TrueNorth-based Hopfield linear solver was applied in the context of real-time robotics applications in [91]. This article looked at three different applications — target tracking (fig. 8.1), optical flow (fig. 8.3), and inverse kinematics (fig. 8.2) — and reported relative error and absolute error of these experiments. Each of these experiments required a different input matrix dimension, and results were reported for over 500 different input matrix values. However, [91] did not include a mathematically complete architecture, nor did it study the effects of computational limitations, both of which have been examined in this dissertation.

Table 9.13 shows the results for 15 different types of matrices, with each matrix repeated

20 times. These experiments were conducted using spike based weight representation scheme on the TrueNorth system. In total, 712 TrueNorth neurons were required to implement the proposed algorithm (just 0.067% of available hardware neurons) and we needed just 11 cores of the 4096 available cores. In the notation of Section 2, the dimensions are $M = 25$, $N = 2$, and $P = 1$. Thus, A is 25×2, B 25×1 and H_j in (4.13a) is 2×1.

In Table 9.13, Columns 5 and 6 show the percentage mean squared error (MSE) and percentage standard deviation squared error(SDSE) of the Hopfield linear solver output relative to double-precision MATLAB quantity ($\|\Delta H\|$). The results of Hopfield linear solver (matrix H) were compared against the results that were obtained using MATLAB's double precision pseudoinverse function. Per the principles of stochastic computing [1], progressive precision holds true for the proposed Hopfield solver. That is, as the number of clock ticks increases, the stochastic error asymptotically approaches zero. We can say from the results of Table 9.13 that the output matrix has a value which is quite close to its double-precision pseudoinverse counterpart in many cases.

Inputs that require low precision for computations (Experiments 1, 2, and 3 in Table 9.13) converge faster and show lower MSE in comparison to inputs that require higher precision (Experiments 9, 14, and 15 in Table 9.13). Note that the scenarios in which the Hopfield linear solver algorithm shows high MSE occur because the algorithm requires considerably more iterations to converge and precision greater than or equal to 10^{-6} to reach a solution. Since the proposed work is using stochastic computing, it would require at least 1 million ticks in the best-case scenario to represent a precision of 10^{-6} for a single value, as well as requiring more iterations to converge. While implementing the Hopfield solver on spiking neural substrate such as TrueNorth, the developer would have to consider this

speed-accuracy tradeoff. For low-precision values, the Hopfield solver would converge faster, but many more ticks may be required for high precision values.

9.8 Summary

This chapter presents a thorough evaluation of different style of proposed linear solver implementation. First, Section 9.1 presents the range analysis of Hopfield linear solver. We can guarantee that the proposed scaling factor will keep the firing rates of neurons low enough that they never saturate. Similarly, in section 9.2, we validate the proposed Hopfield neural network against robotics applications such as object tracking, optical flow and inverse kinematics. The output results were compared against the results achieved from MATLAB's double precision pseudoinverse function for the same set of input matrices.

Second, section 9.3 compares the TrueNorth-based Hopfield linear solver against standard QR inverse algorithms that were implemented on the ARM processor and in FPGA. Experiments with the optical-flow application showed the energy benefits of deploying a reduced-precision and energy-efficient generalized matrix inverse engine on the IBM TrueNorth platform. Since TrueNorth architecture was designed to be low power, deployment of multiple linear solvers running in parallel could give a $10\times$ to $100\times$ improvement in terms of energy consumed per frame over FPGA and ARM core baselines.

Third, we present the benefits of population coding approach against a single instance implementation in section 9.4, followed by section 9.5 which shows the benefits of implementing stochastic computing on an ultra low-power FPGA compared to more standard implementation approaches. With population coding approach we can achieve up to

25.56x speedup when compared to a single instance implementation of linear solver. The hardware analysis shows that with stochastic computing, designers can achieve 7x area reduction and 275x reduction in power while achieving the same accuracy as fixed point implementation.

Fourth, section 9.6 covers the benefits of having an adaptive scaling architecture over scaling the input values by the factor η (refer to eqn. 4.10). When compared to a single instance implementation of the linear solver with input values scaled by the factor η, the proposed adaptive scaling technique a linear solver can achieve up to 5.4x speedup and up to 33.41x smaller estimation of scaling factor.

Finally, Section 9.7 compares the results of proposed linear solver against MATLAB's double precision pseudo-inverse function. Results presented in Table 9.13 suggests that a stochastic-computing implementation can produce an output matrix that are quite close to their double-precision pseudoinverse counterparts in many cases. However, the developer would have to keep in mind speed-accuracy tradeoff. For low precision values, the Hopfield solver would converge faster, while many more ticks would be required for high precision values.

Table 9.13: Results for Hopfield linear solver with spike based weight representation. Column 2 describes how the values of matrices A and B were generated, while Column 3 explains why these matrices were chosen. Column 4 presents the number of clock ticks (or the spike duration) for each experiment. Columns 5 and 6 show the percentage mean (MSE) and percentage standard deviation (SDSE) of the squared error of the Hopfield linear solver output relative to double-precision MATLAB quantity ($\|\Delta H\|$).

Experiment number	Properties of matrices A and B	Additional comments	Time ticks (in millions)	MSE for $\|\Delta H\|$ (%)	SDSE for $\|\Delta H\|$ (%)
1	Each element chosen uniformly in $[-1, 1]$	Most basic test for Hopfield linear solver	1.05	0.0004	0.0013
2	Each element is an integer chosen uniformly in $[-100, 100]$	Observe the behavior of linear solver as the input range is increased	3.5	0.0025	0.008
3	Each element chosen uniformly in $[-100, 100]$	Each element can have fractional values	3.5	0.0014	0.0028
4	Each element chosen uniformly in $[1, 100]$	All elements of A have the same sign	4	0.0353	0.19
5	Each element chosen uniformly in $[0.001, 1]$	Analyzing the convergence when the values are small	4	0.0038	0.008
6	Each element chosen uniformly in $[0.0001, 1]$	Analyzing the convergence when the possible values are smaller than previous experiments	4.25	0.0068	0.0234
7	Each element chosen uniformly in $[-1000, 1000]$	Higher precision is required for calculation	4	0.0186	0.0413
8	Each element chosen uniformly in $[-10,000, 10,000]$	Testing for the cases when even more precision is required for calculation	4	0.32	0.83
9	Each element chosen uniformly in $[1, 10,000]$	Matrix A has elements with same sign; requires higher precision for convergence	4	1.16	2.97
10	Each element chosen uniformly in $[-1000, 1000]$ except that 50% of elements in A and B are 0	Effect of sparsity on final result and convergence	4	0.024	0.0488
11	Each element chosen uniformly in $[1, 10,000]$ except for 50% zeros	Effect of sparsity on final result and convergence	4	0.24	0.94
12	Each element chosen uniformly in $[0.0001, 1]$ except for 45% zeros in A and B	Effect of sparsity on final result and convergence when elements of A are small	4.25	0.0038	0.0114
13	Each element chosen uniformly in $[0, 50]$. For matrix A, ratio of smallest to largest singular values is about .25	Both the eigenvalues of W_{hop} will have magnitude close to 1, but will have opposite signs	4.25	0.37	1.01
14	Each element chosen uniformly in $[-5 \times 10^5, 5 \times 10^5]$	Testing for the cases when up-to 10^{-6} precision would be required for calculation	4.25	5.11	9.54
15	Each element chosen uniformly in $[1, 5 \times 10^5]$	Precision of better than 10^{-6} would be required for calculation and all matrix input values have the same sign	4.25	96.48	316.54

10 CONCLUSION AND REFLECTIONS

To the best of our knowledge, this dissertation is the first attempt to formalize a computational framework for determining scaling factors and population coding when deploying a recurrent numerical solver on limited-precision neural hardware. The proposed research developed a mathematical and algorithmic framework for calculating generalized matrix inverses on this hardware platform. Apart from using the proposed algorithm for real-time robotics applications, it could also be used for on-chip training of multi-layered perceptrons and extreme-learning machines [98] for a variety of classification and regression based tasks. We validate the mathematical model using a Hopfield network-based linear solver that has been implemented on the IBM TrueNorth spiking neural substrate. Our empirical results show that the analytic bounds are never violated for the scenarios evaluated.

10.1 Extending Hopfield neural network based linear solver to other hardware substrates

Sections 5.2.2 and 5.3 present algorithms that can compute matrix inverses using concepts from stochastic computing [30]. The proposed algorithms can be extended to other spiking and non-spiking hardware substrates that have the ability to perform stochastic computing and provides the capability to have recurrent neural network connections.

Prior work such as [68], [74], [29], [19], [97], [99] and [13] show that digital spiking neural substrates can perform stochastic computing. Because of the energy-efficiency of these architectures, they are promising for robotic control operations [18] and online learning [19].

The similarity in data representation allows these SNN architecture to perform arithmetic similar to stochastic computing [97] [50].

We can also perform stochastic computing on non-spiking hardware substrates such as FPGAs [52], FinFETs [107], and magnetic-tunnel-junction [59]. These technologies provide us with a promising opportunity to implement linear solvers based on Hopfield neural networks while being energy-efficient and operate at a higher frequency. Developers would have to keep in mind that the proposed linear solver is performing lossless addition (Figure 3.3(h)). When a neuron receives spikes from multiple inputs at the same time, its membrane potential increases by the same amount as the number of input spikes it has received at that time tick. The membrane potential decreases by one after the neuron fires. Scaling factor η that was derived in (4.10) and Claim 4.1 guarantees that even with a lossless addition present in the equations, the intermediate computation will never saturate.

Deterministic bitstreams: Other work [44] has proposed deterministic bitstreams to reduce latency. Unfortunately, this work only describes how to implement single arithmetic operations and does not discuss how to extend the work to cascaded, dependent operations, such as those required for the algorithms explored in this paper. To the best of our understanding, this would require buffering between each operation to regenerate the deterministic encoding. For this reason, we feel this scheme is less flexible than population coding, which seamlessly enables streaming dataflow computation in both feedforward and feedback configurations.

10.2 Reflections

It is very encouraging to see that research community has spent a considerable amount of effort in understanding the computations that are performed in a human brain and using these lessons to propose ultra-low power spiking neural network architectures that can be used for commercial applications. But even after these significant achievements there are certain problems both in neuroscience and stochastic computing that need to be addressed before we see these hardware getting deployed for real-time applications.

10.2.1 Information theory gap in neuroscience and stochastic computing

In chapter 6 we presented different coding schemes that is used by spiking neural networks to represent information. But the primary focus of neuromorphic community has been on developing algorithms that represent the numbers or values with rate coding scheme and replicate the behavior of traditional deep learning models [25] [20] or regular compute platforms. Recently, there have been work from architecture community that have proposed temporal coding scheme to divide the computations into space and time simultaneously [96].

At present we need better information theory models that bridge the gap between values that are presented by different coding schemes and how multiple neurons operate on these different representations and later interact with each other to output a sensible information. These mathematical and theoretical models will help the community to come

up with hardware platforms that can be extremely useful in real-world practical scenarios. These kind of research has benefitted deep learning community considerably [86] [94] and we hope such reasoning will take the neuromorphic community further.

10.2.2 Unsupervised learning for regression based problems in SNNs

The last decade has seen numerous research being proposed that is related to STDP learning mechanism. But so far most of these research have only focused on using STDP as unsupervised learning algorithm for image classification task [48] [21]. At present there is dearth of literature that explains how is it a human brain is able to perform regression based tasks effortlessly and at the same time learn the necessary model using unsupervised learning mechanism. Since the computations are being performed on streams of spikes (or bits), it is important for us to understand how is it that even after such a noisy data representation scheme a human brain is able to perform complex regression based tasks with very low latency. This dissertation has attempted to answer this question with the proposed mathematical model of Hopfield neural network but research community would need more theoretical models that can explain how new recurrent neural networks model emerge and learn to identify regression based problems.

10.2.3 Domain Specific Language for rapid stochastic computing prototyping

As we have seen in this dissertation and in prior work [100] stochastic computing has shown considerable promise for real-time applications, But these computing techniques are lim-

ited by lack of programming tools that we can use for rapid prototyping and deployment to reconfigurable fabrics. Various research groups have started looking at programming languages and compilation tools for rapid prototyping of SNNs on neuromorphic hardware [55] [4], but there still aren't any programming tools that we can use to rapidly prototype stochastic computing hardware and evaluate how much benefits we might get when we use such computing schemes when performing calculations on streaming input data.

BIBLIOGRAPHY

[1] A. Alaghi and J. Hayes. "Survey of Stochastic Computing". In: *ACM Trans. Embed. Comput. Syst.* 12.2s (May 2013), 92:1–92:19. ISSN: 1539-9087.

[2] A. Alaghi, C. Li, and J. P. Hayes. "Stochastic Circuits for Real-time Image-processing Applications". In: *Proceedings of the 50th Annual Design Automation Conference*. DAC '13. Austin, Texas: ACM, 2013, 136:1–136:6. ISBN: 978-1-4503-2071-9. DOI: 10.1145/2463209.2488901. URL: http://doi.acm.org/10.1145/2463209.2488901.

[3] M. Albert, A. Schnabel, and D. Field. "Innate visual learning through spontaneous activity patterns". In: *PLoS Computational Biology* 4.8 (2008).

[4] A. Amir et al. "Cognitive computing programming paradigm: A Corelet Language for composing networks of neurosynaptic cores". In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. Aug. 2013, pp. 1–10.

[5] F. Anselmi et al. "Unsupervised Learning of Invariant Representations in Hierarchical Architectures". In: *CoRR* abs/1311.4158 (2013). URL: http://arxiv.org/abs/1311.4158.

[6] D. W. Arathorn. *Map-Seeking Circuits in Visual Cognition: A Computational Mechanism for Biological and Machine Vision*. Stanford, CA, USA: Stanford University Press, 2002. ISBN: 0804742774.

[7] D. Arathorn. "Recognition under transformation using superposition ordering property". In: *Electronics Letters* 37.3 (Feb. 2001), pp. 164–166. ISSN: 0013-5194. DOI: 10.1049/el:20010123.

[8] A. Ben-Israel and A. Charnes. "Contributions to the theory of generalized inverses". In: *J. Soc. Indust. Appl. Math.* 11.3 (1963), pp. 55–60.

[9] B. V. Benjamin, P. Gao, and et. al. "Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations". In: *Proceedings of the IEEE* 102.5 (May 2014), pp. 699–716.

[10] N. Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718.

[11] N. Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964.

[12] *Biological Neuron*. URL: https://lloydscientists.weebly.com/chapter-43-the-nervous-system.html (visited on 07/16/2018).

[13] A. S. Cassidy et al. "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores". In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. Aug. 2013, pp. 1–10.

[14] J. Chang et al. "Development of precise maps in visual cortex requires patterned spontaneous activity in retina". In: *Neuron* 48.5 (2005), pp. 797–809.

[15] T. H. Chen and J. P. Hayes. "Analyzing and controlling accuracy in stochastic circuits". In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. Oct. 2014, pp. 367–373.

[16] K. Cheung, S. R. Schultz, and W. Luk. "NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors". In: *Frontiers in Neuroscience* 9 (2016), p. 516. ISSN: 1662-453X. URL: http://journal.frontiersin.org/article/10.3389/fnins.2015.00516.

[17] T. S. Clawson et al. "An adaptive spiking neural controller for flapping insect-scale robots". In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. Nov. 2017, pp. 1–7. DOI: 10.1109/SSCI.2017.8285173.

[18] T. S. Clawson et al. "Spiking Neural Network (SNN) Control of a Flapping Insect-scale Robot". In: *Conference on Decision and Control, IEEE*. 55. 2016, pp. 3381–3388. ISBN: 9781509018369.

[19] M. Davies et al. "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning". In: *IEEE Micro* 38.1 (Jan. 2018), pp. 82–99. ISSN: 0272-1732. DOI: 10.1109/MM.2018.112130359.

[20] P. U. Diehl et al. "TrueHappiness: Neuromorphic Emotion Recognition on TrueNorth". In: *CoRR* abs/1601.04183 (2016). arXiv: 1601.04183. URL: http://arxiv.org/abs/1601.04183.

[21] P. Diehl and M. Cook. "Unsupervised learning of digit recognition using spike-timing-dependent plasticity". In: *Frontiers in Computational Neuroscience* 9 (2015), p. 99. ISSN: 1662-5188. DOI: 10.3389/fncom.2015.00099. URL: https://www.frontiersin.org/article/10.3389/fncom.2015.00099.

[22] P. E. Duhamel et al. "Hardware in the loop for optical flow sensing in a robotic bee". In: *IEEE International Conference on Intelligent Robots and Systems* (2011), pp. 1099–1106. ISSN: 2153-0858. DOI: 10.1109/IROS.2011.6048759.

[23] F. A. Endo, D. Couroussé, and H.-P. Charles. "Micro-architectural Simulation of Embedded Core Heterogeneity with Gem5 and McPAT". In: *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. RAPIDO '15. Amsterdam, Holland: ACM, 2015, 7:1–7:6. ISBN: 978-1-60558-699-1.

[24] S. K. Esser et al. "Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores". In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. Aug. 2013, pp. 1–10.

[25] S. K. Esser et al. "Convolutional networks for fast, energy-efficient neuromorphic computing". In: *Proceedings of the National Academy of Sciences* 113.41 (2016), pp. 11441–11446. ISSN: 0027-8424. DOI: 10.1073/pnas.1604850113. eprint: http://www.pnas.org/content/113/41/11441.full.pdf. URL: http://www.pnas.org/content/113/41/11441.

[26] D. Floreano and R. J. Wood. "Science, technology and the future of small autonomous drones". In: *Nature* 521.7553 (2015), pp. 460–466. ISSN: 14764687. DOI: 10.1038/nature14542.

[27] D. Floreano et al. “Flying Insects and Robots”. In: *Springer-Verlag Berlin Heidelberg* 1 (2009), p. 316. DOI: 10.1007/978-3-540-89393-6.

[28] K. Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological Cybernetics*. Vol. 36. 1980, pp. 193–202.

[29] S. B. Furber et al. “The SpiNNaker Project”. In: *Proceedings of the IEEE* 102.5 (May 2014), pp. 652–665. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2304638.

[30] B. R. Gaines. “Stochastic Computing”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 149–156.

[31] V. C. Gaudet and A. C. Rapley. “Iterative decoding using stochastic computation”. In: *Electronics Letters* 39.3 (Feb. 2003), pp. 299–301. ISSN: 0013-5194. DOI: 10.1049/el:20030217.

[32] L. Goux et al. “Understanding of the intrinsic characteristics and memory trade-offs of sub-micron filamentary RRAM operation”. In: *Symposium on VLSI Circuit Digest of Technical Paper*. Vol. 88. 2011. 2013, pp. 2012–2013. ISBN: 9784863483477.

[33] G. Guennebaud, B. Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[34] A. G. Hashmi and M. H. Lipasti. “DISCOVERING CORTICAL ALGORITHMS”. In: *Proceedings of the International Conference on Neural Computation (ICNC)*. Oct. 2010.

[35] J. P. Hayes. “Introduction to stochastic computing and its challenges”. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2015, pp. 1–3. DOI: 10.1145/2744769.2747932.

[36] J. J. Hopfield. “Neural networks and physical systems with emergent collective computational abilities”. In: *Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558. ISSN: 0027-8424. DOI: 10.1073/pnas.79.8.2554. eprint: http://www.pnas.org/content/79/8/2554.full.pdf. URL: http://www.pnas.org/content/79/8/2554.

[37] M. Hopkins and S. Furber. “Accuracy and Efficiency in Fixed-Point Neural ODE Solvers”. In: *Neural Comput.* 27.10 (Aug. 2015), pp. 2148–2182.

[38] *https://en.wikipedia.org/wiki/Affine_transformation*. 2016. URL: https://en.wikipedia.org/wiki/Affine%5C_transformation.

[39] L. Huai et al. “Stochastic computing implementation of trigonometric and hyperbolic functions”. In: *2017 IEEE 12th International Conference on ASIC (ASICON)*. Oct. 2017, pp. 553–556. DOI: 10.1109/ASICON.2017.8252535.

[40] D. H. Hubel and T. N. Wiesel. “Receptive fields of single neurons in the cat’s striate cortex”. In: *The Journal of physiology* 148.3 (Oct. 1959), pp. 574–591. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1363130/.

[41] *IBM Neurosynaptic System Neuron Function Library Reference Manual*. Tech. rep. IBM Corporation, 2016.

[42] E. M. Izhikevich. "Which model to use for cortical spiking neurons?" In: *IEEE Transactions on Neural Networks* 15.5 (Sept. 2004), pp. 1063–1070. ISSN: 1045-9227. DOI: 10.1109/TNN.2004.832719.

[43] M. Jaderberg et al. "Spatial Transformer Networks". In: *CoRR* abs/1506.02025 (2015). arXiv: 1506.02025. URL: http://arxiv.org/abs/1506.02025.

[44] D. Jenson and M. Riedel. "A deterministic approach to stochastic computation". In: *Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD '16* (2016), pp. 1–8. ISSN: 10923152. DOI: 10.1145/2966986.2966988. URL: http://dl.acm.org/citation.cfm?doid=2966986.2966988.

[45] Z. Ji and J. Weng. "WWN-2: A biologically inspired neural network for concurrent visual attention and recognition". In: *Neural Networks (IJCNN), The 2010 International Joint Conference on*. July 2010, pp. 1–8. DOI: 10.1109/IJCNN.2010.5596778.

[46] X. Jin, S. B. Furber, and J. V. Woods. "Efficient modelling of spiking neural networks on a scalable chip multiprocessor". In: *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. June 2008, pp. 2812–2819.

[47] D. Kerr et al. "Spiking Hierarchical Neural Network for Corner Detection". In: *Proceedings of the International Conference on Neural Computation Theory and Applications* (2011), pp. 230–235. DOI: 10.5220/0003682402300235. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0003682402300235.

[48] S. R. Kheradpisheh et al. "STDP-based spiking deep convolutional neural networks for object recognition". In: *Neural Networks* 99 (2018), pp. 56–67. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2017.12.005. URL: http://www.sciencedirect.com/science/article/pii/S0893608017302903.

[49] J. Kiefer and J. Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function". In: *Ann. Math. Statist.* 23.3 (Sept. 1952), pp. 462–466. DOI: 10.1214/aoms/1177729392. URL: https://doi.org/10.1214/aoms/1177729392.

[50] H. Kim, J. Yu, and K. Choi. "Hybrid spiking-stochastic Deep Neural Network". In: *2017 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. Apr. 2017, pp. 1–4. DOI: 10.1109/VLSI-DAT.2017.7939642.

[51] J. Z. Leibo et al. *Learning Generic Invariances in Object Recognition: Translation and Scale*. Tech. rep. MIT-CSAIL-TR-2010-061. Massachusetts Institute of Technology, Dec. 2010.

[52] B. Li, M. H. Najafi, and D. J. Lilja. "Using Stochastic Computing to Reduce the Hardware Requirements for a Restricted Boltzmann Machine Classifier". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: ACM, 2016, pp. 36–41. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847340. URL: http://doi.acm.org/10.1145/2847263.2847340.

[53] P. Li et al. "Computation on Stochastic Bit Streams Digital Image Processing Case Studies". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.3 (Mar. 2014), pp. 449–462. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2247429.

[54] S. Li et al. “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures”. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2009, pp. 469–480.

[55] C.-K. Lin et al. “Mapping Spiking Neural Networks Onto a Manycore Neuromorphic Architecture”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 78–89. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192371. URL: http://doi.acm.org/10.1145/3192366.3192371.

[56] M. Lipasti and C. Schulz. *End-to-End Stochastic Computing*. Austin, TX, USA, Feb. 2017. URL: http://pharm.ece.wisc.edu/papers/wp3_2017.pdf.

[57] O. Lomp, C. Faubel, and G. Schoner. “A Neural-Dynamic Architecture for Concurrent Estimation of Object Pose and Identity”. In: *Frontiers in Neurorobotics* 11 (2017), p. 23. ISSN: 1662-5218. DOI: 10.3389/fnbot.2017.00023. URL: https://www.frontiersin.org/article/10.3389/fnbot.2017.00023.

[58] B. D. Lucas and T. Kanade. “An Iterative Image Registration Technique with an Application to Stereo Vision”. In: *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI’81. Vancouver, BC, Canada: Morgan Kaufmann Publishers Inc., 1981, pp. 674–679. URL: http://dl.acm.org/citation.cfm?id=1623264.1623280.

[59] Y. Lv and J. Wang. “A Single Magnetic-Tunnel-Junction Stochastic Computing Unit”. In: *2017 International Electron Devices Meeting*. IEDM ’17. San Francisco, California, USA, 2017.

[60] K. Y. Ma. *RoboBee*. 2015. URL: http://www.aboutkevinma.com/index.html#publications (visited on 04/01/2018).

[61] K. Y. Ma et al. “Controlled Flight of a Biologically Inspired, Insect-Scale Robot”. In: *Science* May (2013), pp. 603–607. ISSN: 1095-9203. DOI: 10.1126/science.1231806.

[62] W. Maass. “Networks of spiking neurons: The third generation of neural network models”. In: *Neural Networks* 10.9 (1997), pp. 1659–1671. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(97)00011-7. URL: http://www.sciencedirect.com/science/article/pii/S0893608097000117.

[63] A. Mathis, A. V. M. Herz, and M. Stemmler. “Optimal Population Codes for Space: Grid Cells Outperform Place Cells”. In: *Neural Computation* 24.9 (2012). PMID: 22594833, pp. 2280–2317. DOI: 10.1162/NECO_a_00319. eprint: https://doi.org/10.1162/NECO_a_00319. URL: https://doi.org/10.1162/NECO_a_00319.

[64] Matrix norm. *Matrix norm — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-August-2018]. 2018. URL: https://en.wikipedia.org/wiki/Matrix_norm.

[65] J. H. Maunsell and D. C. V. Essen. “Functional properties of neurons in middle temporal visual area of the macaque monkey. I. Selectivity for stimulus direction, speed, and orientation”. In: *The Journal of physiology* 49 (May 1983), pp. 1127–1147. URL: https://www.ncbi.nlm.nih.gov/pubmed/6864242.

[66] R. Memisevic and G. Exarchakis. "Learning invariant features by harnessing the aperture problem". In: *Proceedings of the 30th International Conference on Machine Learning*. Atlanta, Georgia, USA, 2013, June 2013.

[67] P. A. Merolla et al. "A million spiking-neuron integrated circuit with a scalable communication network and interface". In: *Science* 345.6197 (2014), pp. 668–673.

[68] P. A. Merolla et al. "A million spiking-neuron integrated circuit with a scalable communication network and interface". In: *Science* 345.6197 (2014), pp. 668–673. ISSN: 0036-8075. DOI: 10.1126/science.1254642. eprint: http://science.sciencemag.org/content/345/6197/668.full.pdf. URL: http://science.sciencemag.org/content/345/6197/668.

[69] Y. Mroueh, S. Voinea, and T. Poggio. "Learning with Group Invariant Features: A Kernel Perspective". In: *Advances in Neural Information Processing Symposium*. 2015. URL: http://arxiv.org/abs/1311.4158.

[70] M. Murphy. *IBM has built a digital rat brain that could power tomorrow?s smartphones*. Aug. 8, 2015. URL: https://qz.com/481164/ibm-has-built-a-digital-rat-brain-that-could-power-tomorrows-smartphones/.

[71] M. H. Najafi et al. "Polysynchronous Clocking: Exploiting the Skew Tolerance of Stochastic Circuits". In: *IEEE Transactions on Computers* 66.10 (Oct. 2017), pp. 1734–1746. ISSN: 0018-9340. DOI: 10.1109/TC.2017.2697881.

[72] S. Narayanan, A. Shafiee, and R. Balasubramonian. "INXS: Bridging the throughput and energy gap for spiking neural networks". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. May 2017, pp. 2451–2459.

[73] A. Nere. "Computing with Hierarchical Attractors of Spiking Neurons". PhD thesis. University of Wisconsin - Madison, 2013.

[74] A. Nere et al. "Bridging the semantic gap: Emulating biological neuronal behaviors with simple digital neurons". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2013, pp. 472–483. DOI: 10.1109/HPCA.2013.6522342.

[75] J. von Neumann. "Probabilistic logics and synthesis of reliable organisms from unreliable components". In: *Automata Studies*. Ed. by C. Shannon and J. McCarthy. Princeton University Press, 1956, pp. 43–98.

[76] J. K. Paik and A. K. Katsaggelos. "Image restoration using a modified Hopfield network". In: *IEEE Transactions on Image Processing* 1.1 (Jan. 1992), pp. 49–63. ISSN: 1057-7149. DOI: 10.1109/83.128030.

[77] T. Poggio et al. *Learning Generic Invariances in Object Recognition: Translation and Scale*. Tech. rep. MIT-CSAIL-TR-2012-035. Massachusetts Institute of Technology, Dec. 2012.

[78] Power iteration. *Power iteration — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-August-2018]. 2018. URL: https://en.wikipedia.org/wiki/Power_iteration.

[79] W. Qian et al. “An Architecture for Fault-Tolerant Computation with Stochastic Logic”. In: *IEEE Transactions on Computers* 60.1 (Jan. 2011), pp. 93–105. ISSN: 0018-9340. DOI: 10.1109/TC.2010.202.

[80] Rayleigh quotient iteration. *Rayleigh quotient iteration — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-August-2018]. 2018. URL: https://en.wikipedia.org/wiki/Rayleigh_quotient_iteration.

[81] A. Ren et al. “SC-DCNN: Highly-Scalable Deep Convolutional Neural Network Using Stochastic Computing”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. Xi’an, China: ACM, 2017, pp. 405–418. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037746. URL: http://doi.acm.org/10.1145/3037697.3037746.

[82] M. Riesenhuber and T. Poggio. “Hierarchical models of object recognition in cortex”. In: *Nature Neuroscience* 2 (1999), pp. 1019–1025.

[83] E. Rolls. “Invariant Visual Object and Face Recognition: Neural and Computational Bases, and a Model, VisNet”. In: *Frontiers in Computational Neuroscience* 35 (June 2012). DOI: 10.3389/fncom.2012.00035.

[84] S. Sabour, N. Frosst, and G. E. Hinton. “Dynamic Routing Between Capsules”. In: *CoRR* abs/1710.09829 (2017). arXiv: 1710.09829. URL: http://arxiv.org/abs/1710.09829.

[85] N. Saraf et al. “IIR filters using stochastic arithmetic”. In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.086.

[86] A. M. Saxe et al. “On the Information Bottleneck Theory of Deep Learning”. In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=ry_WPG-A-.

[87] J. Schemmel, J. Fieres, and K. Meier. “Wafer-scale integration of analog neural networks”. In: *Proceedings of the International Joint Conference on Neural Networks* (2008), pp. 431–438.

[88] C. D. Schuman et al. “A Survey of Neuromorphic Computing and Neural Networks in Hardware”. In: *CoRR* abs/1705.06963 (2017). URL: http://arxiv.org/abs/1705.06963.

[89] L. semiconductors. *DS1048 - iCE40 Ultra Family Data Sheet*. June 2016. URL: http://www.latticesemi.com/~/media/LatticeSemi/Documents/DataSheets/iCE/iCE40UltraFamilyDataSheet.pdf..

[90] A. Shrestha et al. “A spike-based long short-term memory on a neurosynaptic processor”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 631–637. DOI: 10.1109/ICCAD.2017.8203836.

[91] R. Shukla, E. Jorgensen, and M. Lipasti. “Evaluating Hopfield-network-based linear solvers for hardware constrained neural substrates”. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. May 2017, pp. 1–8.

[92] R. Shukla and M. Lipasti. "A self-learning map-seeking circuit for visual object recognition". In: *2015 International Joint Conference on Neural Networks (IJCNN)*. July 2015, pp. 1–8.

[93] R. Shukla et al. "Computing Generalized Matrix Inverse on Spiking Neural Substrate". In: *Frontiers in Neuroscience* 12 (2018), p. 115. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00115. URL: https://www.frontiersin.org/article/10.3389/fnins.2018.00115.

[94] R. Shwartz-Ziv and N. Tishby. "Opening the Black Box of Deep Neural Networks via Information". In: *CoRR* abs/1703.00810 (2017). arXiv: 1703.00810. URL: http://arxiv.org/abs/1703.00810.

[95] H. Sim and J. Lee. "A New Stochastic Computing Multiplier with Application to Deep Convolutional Neural Networks". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. Austin, TX, USA: ACM, 2017, 29:1–29:6. ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062290. URL: http://doi.acm.org/10.1145/3061639.3062290.

[96] J. Smith. "Space-Time Algebra: A Model for Neocortical Computation". In: *2018 International Symposium on Computer Architecture (ISCA)*. Los Angeles, CA, USA, June 2018.

[97] S. C. Smithson et al. "Stochastic Computing Can Improve Upon Digital Spiking Neural Networks". In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. Oct. 2016, pp. 309–314. DOI: 10.1109/SiPS.2016.61.

[98] J. Tang, C. Deng, and G. B. Huang. "Extreme Learning Machine for Multilayer Perceptron". In: *IEEE Transactions on Neural Networks and Learning Systems* 27.4 (Apr. 2016), pp. 809–821. ISSN: 2162-237X. DOI: 10.1109/TNNLS.2015.2424995.

[99] C. S. Thakur et al. "Bayesian Estimation and Inference Using Stochastic Electronics". In: *Frontiers in Neuroscience* 10 (2016), p. 104. ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00104. URL: https://www.frontiersin.org/article/10.3389/fnins.2016.00104.

[100] P. S. Ting and J. P. Hayes. "Stochastic Logic Realization of Matrix Operations". In: *2014 17th Euromicro Conference on Digital System Design*. Aug. 2014, pp. 356–364. DOI: 10.1109/DSD.2014.75.

[101] S. University. *FPGA Logic Cells Comparison*. Apr. 2014. URL: http://ee.sharif.edu/~asic/Docs/fpga-logic-cells_V4_V5.pdf.

[102] K. Wu et al. "Computing matrix inversion with optical networks". In: *Opt. Express* 22.1 (Jan. 2014), pp. 295–304. DOI: 10.1364/OE.22.000295. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-22-1-295.

[103] Xilinx. *Vivado Design Suite User Guide-High Level Synthesis*. May 2014. URL: https://www.xilinx.com/support.html.

[104] Xilinx. *Xilinx Power Estimator*. 2017. URL: https://www.xilinx.com/products/technology/power/xpe.html.

[105] B. Zhang. "Computer vision vs. human vision". In: *Cognitive Informatics (ICCI), 2010 9th IEEE International Conference on*. July 2010, pp. 3–3. DOI: 10.1109/COGINF.2010.5599750.

[106] Y. Zhang, D. Guo, and Z. Li. "Common Nature of Learning Between Back-Propagation and Hopfield-Type Neural Networks for Generalized Matrix Inversion With Simplified Models". In: *IEEE Transactions on Neural Networks and Learning Systems* 24.4 (Apr. 2013), pp. 579–592. ISSN: 2162-237X. DOI: 10.1109/TNNLS.2013.2238555.

[107] Y. Zhang et al. "Design Guidelines of Stochastic Computing Based on FinFET: A Technology-Circuit Perspective". In: *2017 International Electron Devices Meeting*. IEDM '17. San Francisco, California, USA, Dec. 2017.