Cognitive and Brain-inspired Processing Using Parallel Algorithms and Heterogeneous Chip Multiprocessor Architectures

by

Daniel R. Mendat

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

October, 2017

Abstract

This thesis explores how some neuromorphic engineering approaches can be used to speed up computations and reduce power consumption using neuromorphic hardware systems. These hardware designs are not well-suited to conventional algorithms, so new approaches must be used to take advantage of the parallel nature of these architectures. Background regarding probabilistic graphical models is presented along with brain-inspired ways to perform inference in Bayesian networks. A spiking neuron implementation is developed on two general-purpose parallel neuromorphic hardware devices, the SpiNNaker and the Parallella. Scalability results are shown along with speed improvements as compared to using mainstream processors on a desktop computer.

General vector-matrix multiplication computations at various levels of precision are also explored using IBM's TrueNorth Neurosynaptic System. The TrueNorth contains highly-configurable hardware neurons and axons connected via crossbar arrays and consumes very little power but is less flexible than a more general-purpose neuromorphic system such as the SpiNNaker. Nevertheless, techniques described here

ABSTRACT

enable useful computations to be performed utilizing such crossbar arrays with spiking neurons including computing word similarities using trained word vector embeddings. Another technique describes how to perform computations using only one column of the crossbar array at a time despite the fact that incoming spikes normally affect all columns of the array.

A way to perform cognitive audio-visual beamforming is presented. Using two systems, each containing a spherical microphone array, sounds are localized using spherical harmonic beamforming. Combining the microphone arrays with 360 degree cameras provides an opportunity to overlay the sound localization with the visual data and create a combined audio-visual salience map. Cognitive computations can be performed on the audio signals to localize specific sounds while ignoring others based on their spectral characteristics.

Finally, an ARM Cortex M0 processor design is shown that will be used to bootstrap and coordinate other processing units on a chip developed in the lab for the DARPA Unconventional Processing of Signals for Intelligent Data Exploitation (UPSIDE) program. This design includes a bootloader which provides full programmability each time the chip is booted, and the processor interfaces with other hardware modules to access the Networks-on-Chip and main memory.

Primary Reader: Andreas G. Andreou

Secondary Reader: Sang ("Peter") Chin

ABSTRACT

Committee Members: Najim Dehak and Philippe Pouliquen

Acknowledgments

I am grateful for many people who have influenced me while I worked on this dissertation. I want to thank my advisor, Andreas Andreou, for all his support and for being a great source of inspiration. He has taught me to think about interesting approaches to problems that span multiple fields. His energy is contagious and his devotion to research and students has helped many a career, mine included.

I also want to thank my co-advisor, Peter Chin. Collaborating with him has been a joy, and his tireless efforts to mentor me have always been valuable. I am grateful for Ralph Etienne-Cummings as well. He has always been available for advice and support.

I thank the other two members of my dissertation committee, Philippe Pouliquen and Najim Dehak. Philippe has been very helpful over the years and has provided a wealth of information and assistance in the lab. Najim has also been a great source of support and insights. I am grateful for their service on the committee.

I truly appreciate the fact that I got to spend my days in the lab with Andrew Cassidy, Joseph Lin, Recep Ogzun, Thomas Murray, Tomás Figliolia, Sean McVeigh,

ACKNOWLEDGMENTS

Kayode Sanni, Gaspar Tognetti, Guillaume Garreau, Kate Fischl, Martín Villemur, Christos Sapsanis, Jeff Craley, Alejandro Pasciaroni, Valerie Rennoll, and Jonah Sengupta. It was wonderful to work, play, and learn with you. I am grateful for the wisdom of Pedro Julián and Philippe Pouliquen who provided their expertise to us all. Ralph Etienne-Cummings and the members of his lab were all very helpful during my time at school as well.

Much of the work in this dissertation was accomplished in collaboration with other groups. I thank the Advanced Processor Technologies group at the University of Manchester, particularly Alan Stokes, Luis Plana, Sergio Davies, Steve Temple, and Stephen Furber, for all their assistance in working with the SpiNNaker hardware platform. In addition, the technical support of IBM researchers, particularly Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, and Paul Merolla, was invaluable when working with the TrueNorth hardware platform. I acknowledge the discussions with Kaitlin Fair from the Georgia Institute of Technology which were helpful in pursuing novel ways of working with the TrueNorth. Andrew Dykman also contributed to implementing and documenting the 8-bit TrueNorth work presented here. Collaboration with Kate Fischl was invaluable for working with many aspects of the TrueNorth platform and other related projects in the lab.

I thank Alejandro Pasciaroni for creating some peripherals for the ARM Cortex M0 architecture we worked on together and for helping me debug modules I worked on. I also thank him for designing the board for connecting the SpiNNaker and the

Parallella together which will enable fast communication between the two devices. I thank Kate Fischl for helping to assemble the board as well.

I thank the Electrical and Computer Engineering department staff, particularly Janel Johnson, Nicole Aaron, Cora Mayenschein, Debbie Race, and Barbara Sullivan, for their assistance with many varied tasks over the years. I also thank Ruth Scally from the Center for Language and Speech Processing for administrative assistance.

Powerlifting was a big part of my life while at Johns Hopkins, so I thank some friends who helped make that enjoyable. At school it was great to train and discuss research with Paul Stanton, Charles Jonassaint, Michael Carlin, and Thomas Murray. I also thank all the friends who joined my wife and me at other gyms later on.

My research could not have been accomplished without generous fellowships from the Electrical and Computer Engineering Department, the Bodmer family, and the Johns Hopkins University Applied Physics Laboratory. I thank Dennis Ryan in particular for his work on maintaining the Applied Physics Laboratory fellowship. I am also grateful for research funding from the NSF grant INSPIRE SMA 1248056 through the Telluride Workshop on Neuromorphic Cognition Engineering, the NSF grant SCH-INT 1344772, an ONR MURI N000141010278, and the DARPA UPSIDE project HR0011-13-C-0051 through BAE Systems.

I thank my wife, Amanda Mendat, for selflessly and enthusiastically reviewing this entire manuscript and offering numerous helpful suggestions.

I thank my family for graciously accepting the fact that I spent many weekends

working on classes and research, even while visiting. They have all been very supportive during the entire process. From grandparents to aunts, uncles, and cousins I am very appreciative of everyone. I thank my mother-in-law, Diane Peterson, and her husband, Ken Peterson, for all their encouragement and general love and support. My sister-in-law, Rebecca Pickering, and her husband, CJ Pickering, have likewise been invaluable sources of positivity along the way. Their son, Jackson Pickering, has already been a source of light in the world. I thank my brother, Benjamin Mendat, and my sister, Julie Mendat, for all their fun banter and support their whole lives. My parents, Amy and Stephen Mendat, have always meant the world to me, and I am so very thankful for their devotion. Finally, my wife Amanda has been there for me every step of the way and I cannot thank her enough. I love them all.

Dedication

This thesis is dedicated to my wife, Amanda. I love you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The semiconductor industry has followed Moore's Law[1] for decades. Gordon Moore originally stated that the transistor count in a given chip area doubles approximately every year,[1] but he changed the estimate later on to say that the doubling would occur approximately every two years.[2] These exponential improvements in computing capabilities have led to a massive increase in the amount of data[3] used in research today. Researchers create more complicated models every day, and more data are collected to train these models.

On the other hand, processor clock speed increases have been slowing in recent years,[4] so progress in improving single-core computing power has decelerated. Parallel processing is an obvious way to mitigate the possible end of Moore's Law, and with the current emphasis on power-efficient microprocessors[5] it is possible to create massively-parallel systems that consume much less power than in the past.

However, even mainstream multiprocessors created today consume lots of power and still cannot think about general complex problems as well as humans can despite decades of advancement. The human brain is full of neurons that consume minimal power,[6] only totaling about 20 W for the entire brain. Despite that low power budget, the brain can solve many problems that traditional computers still have trouble solving.

Advancements in machine learning (particularly in deep learning) have made impressive progress on the computing side in recent years[7] over a wide variety of fields including speech recognition, natural language processing, object recognition, genomics, and medicine to name a few. For example, the state of the art techniques in object recognition these days generally utilize deep networks, often convolutional neural networks.[8,9] Human poses can be estimated through the use of convolutional neural networks[10] as well, and extending these principles further creates the ability to perform action recognition on videos.[11] In fact, it is possible to perform action recognition with neural networks[12] and other techniques[13] without the use of a camera, relying on only a few single dimensional active acoustic signals and taking advantage of the Doppler effect.

Autonomous driving has become an extremely hot area lately given that many car manufacturers have been working on integrating early versions of that technology into their vehicles. Other companies are contributing as well. For example, NVIDIA has an autonomous driving group that has demonstrated fully autonomous driving on

public roads of all kinds including highways and rural routes with no lane markings, utilizing convolutional neural networks.[14] These general concepts have been adapted for use in portable robots with specialized hardware for terrain navigation as well.[15]

It is out of the scope of this thesis to adequately cover all the areas where neural networks have improved upon state of the art results in machine learning tasks. However, it is clear that neural networks have impacted many fields in a profound way. These tasks are often ones where humans excel. Image recognition, speech understanding, driving, etc., are all areas within which the human brain is adept at understanding and functioning but computing systems have only recently been able to catch up.

On a superficial level neural networks function similarly to how the brain works. Spikes of electrical activity travel through the brain's neurons via axons of one neuron connected to dendrites of another. When conditions are right, typically when enough spikes reach excitatory connections at a neuron's dendrites, the neuron sends out more spike(s) along its axon that can travel to other neurons in the brain. Neural networks typically include nodes that aggregate inputs that are summed before a nonlinearity is applied to that input.[7] This nonlinearity can be thought of in very general terms as being analogous to the decision for whether a node ("neuron") in the brain spikes given its input.

Neuromorphic engineering[16–18] attempts to emulate the brain and other aspects of biology to create hardware/software architectures that generally consume less power

than conventional techniques and often employ a high level of parallelism just as the neurons in the brain all work simultaneously. The field encompasses a wide range of projects and goals that span from modeling biology as accurately as possible to borrowing concepts from biology to improve conventional methods without being so strict about how close they are to the real world.

For example, one project that uses neurons to implement mathematical functionality as well as to model areas of the brain is called Nengo.[19] Based on the Neural Engineering Framework (NEF),[20] Nengo provides a useful way to both get started modeling neural systems to accomplish a goal as well as a way to build more complicated systems. Modeling neuromorphic systems in software is useful, but without low-power hardware it is difficult to use these techniques to reduce power consumption in an application.

There are many low-power neuromorphic systems with various levels of impact in the field. This thesis focuses on three of them, namely the SpiNNaker, Parallella, and TrueNorth. The SpiNNaker is the Spiking Neural Network Architecture and was created at the University of Manchester.[21–23] The system consists of chips, each containing 18 ARM968 cores running at about 200 MHz, each capable of 32-bit fixed-point math. These chips are connected via a fast mesh network, and two main boards are currently available for research: a 4-chip board and a 48-chip board. The larger board contains over 800 cores that can be used to perform parallel computations, and Nengo has been implemented on this hardware to accelerate the simulation

of networks of biologically-plausible neurons.[24,25] These large boards can be linked together to form massive networks of ARM cores for parallel computations.

The Parallella[26] is an open-source board that contains a Xilinx Zynq7000 series system-on-chip[27] containing two ARM A9 cores and a field-programmable gate array (FPGA) in addition to an Adapteva Epiphany coprocessor[28] which integrates 16 floating-point processors running at 1 GHz. While the architecture is less neuromorphic than the SpiNNaker in a sense due to the fact that the Parallella is less distributed and runs faster cores capable of more computations, the Parallella is nevertheless an interesting platform for software architecture exploration and comparisons with the SpiNNaker.

The IBM Neurosynaptic System,[29,30] also called the TrueNorth chip, is the final piece of neuromorphic hardware programmed in this thesis. Unlike the SpiNNaker and the Parallella, the TrueNorth consists of networks of hardware neurons that can be configured independently. Instead of programming general-purpose processors to emulate neurons the hardware can only consist of neurons, and the neuron parameters are directly programmed to perform a task. This type of specialized architecture leads to very minimal power consumption but also creates limitations when programming the chip to perform generalized tasks. However, there are plenty of applications the TrueNorth is well-suited for due to having over 1 million neurons and over 268 million synaptic connections that are all individually programmable.

The first two devices described above, the SpiNNaker and the Parallella, contain

von Neumann processing units. However, they are not programmed in a typical manner due to their parallel nature. The SpiNNaker can only communicate via special packets that can be sent around the board through the use of programmable routers located on each chip, and in this thesis the Parallella processing units communicate by writing to and reading from specific areas of shared memory. SpiNNaker packets can additionally be dropped if they experience deadlock, so there are challenges with these architectures beyond standard parallel programming techniques that apply to a single computer with reliable message passing.

On the other hand, the TrueNorth was created specifically to overcome limitations of typical von Neumann architectures,[29] specifically the von Neumann bottleneck[31] which arises from the CPU having access to the memory (and thus the data and instructions) through one bus. Most data must go through that bus (omitting details such as cache designs and other parameters), so the bus limits the speed at which computations can occur. The TrueNorth architecture does not have that singular bottleneck due to its distributed nature and in particular its memory that is colocated with its processing units. The TrueNorth stores its memory states (neuron membrane potentials, parameters, etc.) near its processing elements (neurons) in each core. In addition, the very nature of the TrueNorth cores themselves means that programming the system is a new challenge because most algorithms are not designed to be executed in parallel, let alone on neurons with limited computing capabilities.

The challenges associated with designing algorithms and implementing them on

these parallel neuromorphic hardware platforms are well worth the cost because these devices have inherent advantages in terms of power and speed as compared to typical general-purpose computing devices. Just as graphical processing units (GPUs) provide speed advantages over central processing units (CPUs) but have limitations and different programming considerations required when implementing algorithms, neuromorphic processing units have their own tradeoffs in the pursuit of increasing capabilities and decreasing power consumption. Researchers are now creating cognitive processing units[17] (CogPUs) to more creatively architect solutions to problems using neuromorphic techniques. These new massively-parallel architectures also create new challenges for chip designers. With so many simultaneous processing units that are sometimes non-homogeneous it can be difficult to analyze the best use of space when creating a microprocessor, but new theory has provided some direction[32] for tackling this type of optimization.

As CogPUs become closer to biology they are more likely to provide more pronounced benefits in power consumption and processing capabilities, particularly as related to the types of problems with which computers struggle and humans excel. These improvements come with more challenges, though, because as the systems get closer to biology, algorithms enter uncharted territories as researchers need to become more creative and design new ways of performing tasks using the new neuromorphic tools at hand without knowing how the brain actually executes these procedures in practice.

It is obvious that mimicking the brain has potential for improving circuit design because of the efficiencies the brain displays. However, there are other reasons as well. For example, cognitive science has applied[33] probabilistic models to many cognitive processes. The processes themselves are highly probabilistic in nature, and it only makes sense that the human brain can cope with these models in a probabilistic way. Taking things one step further, the brain even has an innate ability to determine the form a model should take on.[34] Rather than having to be told whether data should be arranged in a list, hierarchy, ring, etc., the human brain can look at various situations and quickly determine a good way to describe the situation in the context of a model. Of course there are more complicated examples of data for which the organization is not obvious, but for many cases even small children can categorize and model the data effectively. For example, given varying objects of different sizes, colors, and shapes, humans can very quickly come up with a way to organize the shapes based on those characteristics, but machines still struggle to deal with these types of open-ended model structure tasks. They typically need to be programmed to fit the data to a given structure rather than determine the structure on their own, but it would be very important both to the field of cognitive development as well as AI to better understand this type of capability.

It has been shown that in various situations the brain performs close to a statistically optimal way. One area of examples of this type of optimal decision is in sensorimotor control.[35] Humans face everyday optimization and estimation tasks

when they interact with the world, and there is uncertainty inherent to each of these types of actions. When people view objects the estimation of these objects' locations and speeds are subject to noise due to the visual system in the brain. The same idea applies to touching objects – the size, shape, and location of objects are noisy measurements in the brain. As most people have noticed on numerous occasions, human hearing is particularly noisy when it comes to estimating the location of objects emitting sounds.

Modeling the stochastic nature of perception in the environment therefore plays a large role in human behavior.[35] One example of a real-life situation is estimating the location of a tennis ball during a game by combining information about the predicted distribution of the location of the opponent's hit and visual information showing a noisy estimate of the ball's actual position and velocity once the ball is struck.[35] One experiment done to support this idea[36] is that people were asked to estimate the location of a cursor on a screen in relation to their hand position. Uncertainty was added to the cursor by making its location fuzzy (depicted as a point cloud), and participants had a mental model (prior distribution) of where the cursor was likely to be based on the number of times the cursor showed up in various locations in previous trials. As a result they were able to predict the location in a manner consistent with using Bayesian statistics to perform the same task.

There are many other examples of this type of behavior,[35] and the same phenomenon occurs when information from two different senses are combined. Studies

have been developed to show that combining uncertain visual and tactile information to estimate an object's height results in behavior close to what Bayesian statistics predicts,[37] and the same situation occurs when combining visual and auditory information to predict an object's location.[38]

Clearly the brain has the capacity to effectively deal with uncertainty in everyday situations, but it also performs complicated tasks. As described earlier, neural networks have only recently been doing as well as humans on specific tasks with great amounts of data required to train them. On the other hand, the human brain can much more quickly learn to perform these tasks and is clearly better at generalizing to new situations. Machine learning has only begun to scratch the surface of what brains are capable of doing, so there are plenty more things to be learned from studying the brain and also from adapting related ideas to algorithms and hardware.

People have only begun to determine how groups of neurons work together in small animals, and less progress has been made in understanding the intricacies of the human brain. However, there are some examples in neuroscience showing that spiking neuron activities match predictions of Bayesian statistics, suggesting that some spike encodings in actual biology may represent probabilistic quantities. For example, when owls localize sounds they are biased toward being accurate straight ahead and typically underestimate angles on the sides. This behavior is consistent with a Bayesian model, and particular decodings of neural populations in the owl's auditory system are consistent with quantities in the model.[39] In another case, neurons in monkey

cortexes fire consistently with the probability that certain eye movements will result in more reward juice.[40]

Other work posits that the brain may weigh possible decision choices using a common technique from statistics - by expressing the choices as a ratio of their probability values. More specifically, the logarithm of the so-called "likelihood ratio" may be formed in the brain as a result of sensory input and stored for later decision making.[41] In fact, this thesis explains how the formation of such a log-likelihood ratio can be used to make decisions about the world on neuromorphic hardware in a way that could potentially be done in the brain.

The thesis begins in Chapter 2 by describing one main class of statistical models called Bayesian networks that can be used to describe situations in the world. The brain must by definition be able to construct a cohesive view of the world in order to make intelligent decisions, and Bayesian networks are one way to create these types of knowledge. The chapter then goes on to explain how these models can be trained so that they represent actual phenomena before describing how decisions can be made (called "inference") using these networks.

Chapter 3 takes these concepts further and describes how these decisions can be made using spiking neurons on parallel neuromorphic hardware (SpiNNaker). Descriptions of the entire automated software flow are included as the process goes from a representation of the network automatically to inference running on the hardware using computations that could be accomplished in the brain. Results of this com-

bined hardware/software flow using the smaller 4-chip SpiNNaker are described in Chapter 4 for various models, and a scalable network structure is introduced that is used in this chapter and later chapters to benchmark inference performance and scalability.

Chapter 5 discusses the work required to implement these networks and inference on the larger 48-chip SpiNNaker as well as how to integrate the same type of computations on the Parallella hardware. That way the small and large SpiNNaker board results can be compared to the Parallella results. A complexity analysis of the algorithms used to process the network and set it up on the SpiNNaker is also explored. Finally, a heterogeneous architecture was created by connecting the large SpiNNaker and the Parallella via Ethernet, and performance results are shown.

The TrueNorth is described in Chapter 6. Very large-scale vector-matrix multiplication architectures using spiking neurons are shown using multiple techniques, each having different tradeoffs. One of those architectures is used to implement natural language processing tasks including detecting word similarities and solving analogies using the TrueNorth and this neural architecture. A technique for performing multiplications stochastically and selecting particular neurons to perform computations with is also shown along with an example image processing pipeline task as an application.

A way to perform cognitive audio-visual beamforming and localization is presented in Chapter 7. The system is cognitive because it enables neuromorphic algorithms

to be developed to localize specific sounds based on their spectral characteristics just as the human brain can focus on specific types of sounds at the expense of others. Theory is presented and then two actual hardware systems running these algorithms are examined. One consists of a fully integrated audio-visual spherical array and the other consists of a more inexpensive spherical array coupled with a basic camera. Performance of both systems are shown and compared.

Finally, Chapter 8 presents the design of an ARM Cortex M0 architecture for massive neuromorphic chips created in the lab. The main goal of the design is to implement an image processing pipeline for wide area aerial photographs. This pipeline includes tasks such as non-uniformity correction for the raw pixel data coming out of the imager, debayering of the incoming pixels to form standard RGB images, dewarping the images so that despite movement of the sequence of images each image is rotated to have a common fixed background, detecting moving objects in the image, and tracking those objects. The chips contain some conventional implementations for performing computations, but there are many processing units that are quite unconventional to enable low power consumption in a neuromorphic manner. Coordinating all these processing units in parallel can be a challenging task, so the M0 architecture provides a way to bootstrap the system and enable a fully programmable interface to utilize these processing units to their full potentials. Although the chips are designed to perform image processing, these units can be used for many other purposes as well.

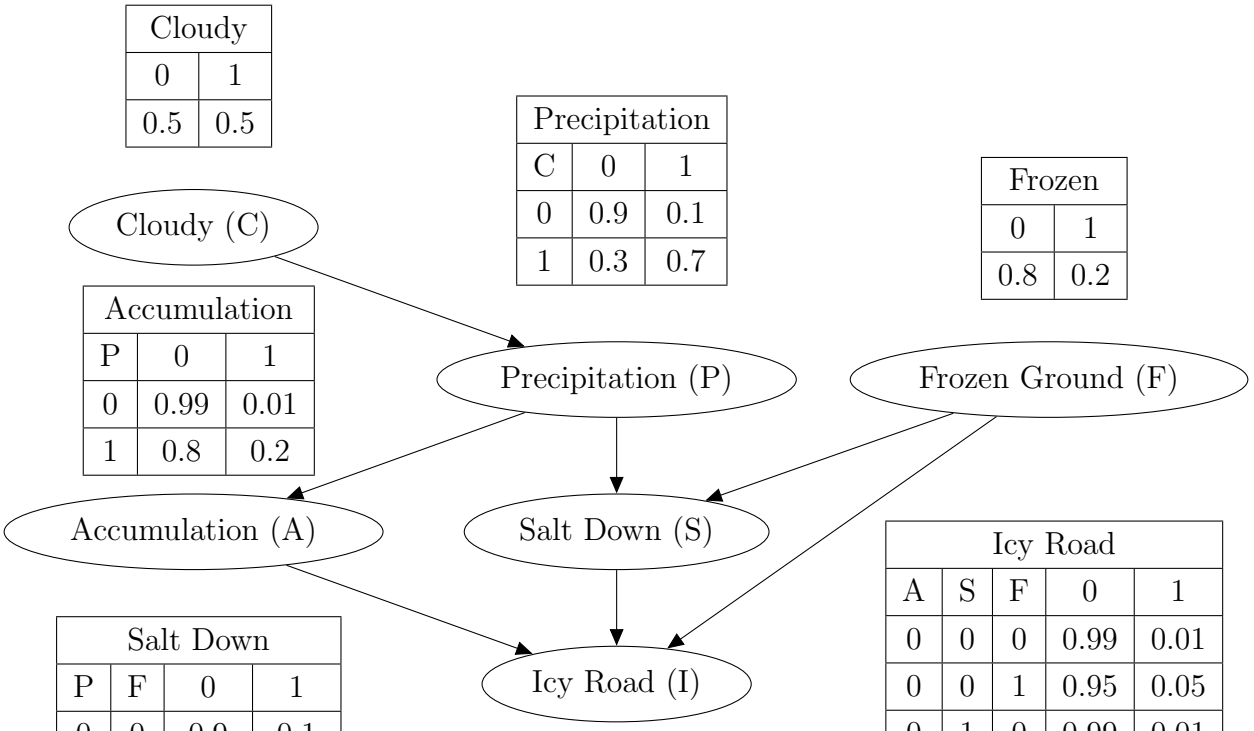
The common theme in all this work is that all these systems are brain-inspired,

massively parallel unconventional processing systems. Useful systems utilizing neuromorphic principles can be created whether they utilize actual spiking neurons or borrow concepts from the brain's capabilities. Research such as this has the potential to continue to improve the state of the art in many areas, particularly as it becomes harder over time to continue increasing mainstream processor clock speeds.

Chapter 2

Bayesian Networks, Learning, and Inference

Bayesian networks are used to model a wide variety of systems in the world, ranging from simple classification tasks all the way to analyzing complex human behavior. Bayesian networks are directed graphical models where each node in the network represents a random variable and the arrows represent dependencies between those variables in a probabilistic model.[42–45] Every node contains a description of its conditional probability distribution (CPD) given its parents (other nodes that point toward the current node). When the nodes are discrete variables with categorical distributions, the conditional distribution is stored as a CPD table where each row corresponds to a different configuration of the node's parents and the values in that row describe the probability of the node's values occurring given those parent values.



Cloudy	
0	1
0.5	0.5

Precipitation		
C	0	1
0	0.9	0.1
1	0.3	0.7

Frozen	
0	1
0.8	0.2

Accumulation		
P	0	1
0	0.99	0.01
1	0.8	0.2

Salt Down			
P	F	0	1
0	0	0.9	0.1
0	1	0.85	0.15
1	0	0.5	0.5
1	1	0.05	0.95

Icy Road				
A	S	F	0	1
0	0	0	0.99	0.01
0	0	1	0.95	0.05
0	1	0	0.99	0.01
0	1	1	0.95	0.05
1	0	0	0.35	0.65
1	0	1	0.01	0.99
1	1	0	0.65	0.35
1	1	1	0.15	0.85

Figure 2.1: Icy Road Network. This example Bayesian network models a probability distribution over random variables associated with precipitation on road surfaces and whether the road is icy.

After a graphical model is established the parameters of the model can be learned from data (see Section 2.1). These parameters describe conditional probability values of nodes given their parents. So the model describes the probabilistic relationships between variables, and each of these relationships has parameters that are learned from data.

Figure 2.1 shows an example Bayesian network called the Icy Road Network which happens to contain only binary nodes. It describes a probability distribution over six variables concerning precipitation and whether the ground is icy depending on other conditions such as temperature and whether salt was placed on the road. Arrows are drawn from parents to children and indicate which variables are directly and conditionally dependent on others. These dependencies are often referred to as "causal relationships" which in aggregate describe the independence structure of the network.[42,44] The CPD tables are shown surrounding the network, and these particular CPD tables were invented for illustrative purposes. For example, a node with no parents such as Cloudy has a prior probability distribution of being 0 or 1. On the other hand, the Precipitation variable has a CPD table that depends directly on the value of Cloudy.

The Icy Road Network can be described intuitively as follows. The cloud cover (Cloudy) and whether the ground is frozen (Frozen Ground) are conditions that do not directly result from the other variables (they are not "caused" by the others). On the other hand, the probability of Precipitation depends on the cloud cover (clouds "cause" precipitation to become more or less likely to occur). Significant accumulation (Accumulation) depends on whether precipitation is occurring at all, and whether salt is administered (Salt Down) depends on whether precipitation is present and the ground is frozen. Finally, icy conditions (Icy Road) depend on whether significant accumulation occurs, salt is on the road, and the ground is frozen.

The nodes in the network collectively describe a joint probability distribution, and the arrows indicate how the joint distribution can be factored in terms of CPDs that are learned from data. Of course the general rules of conditional probability apply so the distribution can be factored many different ways, but the arrows indicate a factorization that can utilize the CPD tables described directly by the model. For example, the Icy Road Network explicitly describes the following joint distribution factorization:

$$\mathrm{P}(A, C, I, F, P, S) = \mathrm{P}(C)\mathrm{P}(F)\mathrm{P}(P|C)\mathrm{P}(A|P)\mathrm{P}(S|P, F)\mathrm{P}(I|A, S, F). \tag{2.1}$$

Learning, or training, must be done in order for the model to be useful, and this process is described in Section 2.1. Training establishes the parameters for every CPD table in the network based on data collected so that the model describes reality as well as it can.

Once the model is trained, useful tasks can be accomplished by performing inference as described in Section 2.2. While performing inference the values of some variables may be fixed as evidence, and then the probability distribution over the remaining variables given the fixed evidence is determined. Inference is where real predictions, classifications, and decisions can be made using the model.

For example, if there is a model describing noise in measurements taken using a sensor as well as the underlying physical process that governs the data themselves, it

is possible to perform inference in order to determine the most likely phenomena that led to those measurements. In order to do so, the measurements are fixed as evidence in the model, and inference is done over the remaining variables to determine the probability distribution of the original data given the measurements collected.

The structure of the network includes the choice of variables and the values they can have as well as the connections between nodes. These choices are generally made or influenced by subject matter experts who have knowledge regarding which relationships are causal. However, these decisions are sometimes altered by other factors including the difficulty of performing learning or inference, so the design of the model is generally an iterative process where it is improved based on the challenges encountered when evaluating how well it works in practice.

The rest of the sections in this chapter describe learning and inference including some basic examples. These concepts are expanded to include a brain-inspired way of performing inference, and comparisons between standard techniques and this neuromorphic technique are shown. Later chapters will take that technique further by exploiting parallelism and brain-inspired hardware architectures to improve the speed, parallelism, and power consumption when performing inference.

2.1 Learning

Learning encompasses a variety of different techniques ranging from the simple to the complex,[42,43,45] and new techniques are described for novel networks regularly. The appropriateness and effectiveness of these techniques are extremely dependent on the actual model in question as well as the data themselves that are available for training. There are two main classes of learning techniques that cover most of these cases: maximum likelihood estimation and Bayesian estimation. Neither of them is perfect for all situations, and they both have various pros and cons that can make one or the other advantageous over the other for particular models and datasets.

Extremely briefly, Bayesian estimation provides a convenient framework to incorporate prior beliefs about random variables in a distribution.[42,43] Including prior beliefs can be helpful when there is insufficient data to create a reasonable estimation of a distribution's parameter. A classic example is the case of flipping a fair coin three times. If the coin comes up heads three times most people would not expect the coin to always come up heads, and the reason they do not expect that result is because they are thinking about their prior beliefs on what is reasonable. On the other hand, maximum likelihood parameter estimation depends more directly on the data, and if the data are skewed due to having a small number of samples, nothing is explicitly built into the framework to take that into account.

Although Bayesian estimation techniques have attractive properties such as incorporation of prior distributions it can be difficult to get the math to "work" when

designing a strategy for parameter estimation. This means that there are a limited number of distributions that are easy to utilize in the Bayesian framework, and as a result some distributions are used in these models for mathematical expediency that do not match the data or real beliefs of experts.

Because this thesis is focused more on inference (see Section 2.2) and all the graphical models are of the same general class unless specifically mentioned, this section will focus on learning in Bayesian networks with discrete categorical distributions at each node. Another assumption made here is that all the data are available for learning, which means there are no random variables for which data are missing. These assumptions make maximum likelihood estimation straightforward, and that process is described below.[42,43,45,46]

Using the Icy Road Network depicted in Figure 2.1, assume there are N instances of training data. Each value is a vector containing the value of each of the 6 random variables in the model. The first step is to create the likelihood function because maximum likelihood estimation involves maximizing the likelihood of the data given the parameters (CPD table entries). This likelihood is the joint distribution (Equa-

tion 2.1) and is shown below:

$$\prod_{d=1}^{N} \mathrm{P}(A = a_d, C = c_d, I = i_d, F = f_d, P = p_d, S = s_d) = \prod_{d=1}^{N} \mathrm{P}(C = c_d)\mathrm{P}(F = f_d)\mathrm{P}(P = p_d|C = c_d)\mathrm{P}(A = a_d|P = p_d)\cdot \mathrm{P}(S = s_d|P = p_d, F = f_d)\mathrm{P}(I = i_d|A = a_d, S = s_d, F = f_d). \tag{2.2}$$

In the above equation, d indexes the N data points, and the variables a_d, c_d, i_d, f_d, p_d, and s_d are the values of the variables A, C, I, F, P, and S in data point d. The right-hand side of the equation is the factorized version described explicitly by the model (see Equation 2.1).

Next, change the notation so that $\mathrm{P}(C = i)$ where $i \in \{0, 1\}$ is C_i, $\mathrm{P}(P = i|C = j)$ where $i \in \{0, 1\}$ and $j \in \{0, 1\}$ is P_{ij}, and so on:

$$\prod_{d=1}^{N} \mathrm{P}(A = a_d, C = c_d, I = i_d, F = f_d, P = p_d, S = s_d) = \prod_{d=1}^{N} \left[\prod_{i} C_i^{I(c_d=i)} \prod_{i} F_i^{I(f_d=i)} \prod_{i,j} P_{ij}^{I(p_d=i,c_d=j)} \prod_{i,j} A_{ij}^{I(a_d=i,p_d=j)} \cdot \prod_{i,j,k} S_{ijk}^{I(s_d=i,p_d=j,f_d=k)} \prod_{i,j,k,l} I_{ijkl}^{I(i_d=i,a_d=j,s_d=k,f_d=l)} \right]. \tag{2.3}$$

Here $I()$ is the indicator function that is 0 when the value passed to I is false and 1 when the value passed to $I()$ is true. Then let $N_{C=i}$ be the number of times node C takes on the value i in the data, $N_{P=i,C=j}$ be the number of times in the data node

P has the value i and node C has the value j simultaneously, and so on:

$$\begin{aligned}\prod_{d=1}^{N} \mathrm{P}(A = a_d, C = c_d, I = i_d, F = f_d, P = p_d, S = s_d) = \\ \prod_{i} C_i^{N_{C=i}} \prod_{i} F_i^{N_{F=i}} \prod_{i,j} P_{ij}^{N_{P=i,C=j}} \prod_{i,j} A_{ij}^{N_{A=i,P=j}}. \\ \prod_{i,j,k} S_{ijk}^{N_{S=i,P=j,F=k}} \prod_{i,j,k.l} I_{ijkl}^{N_{I=i,A=j,S=k,F=l}}.\end{aligned} \tag{2.4}$$

Because the logarithm function is a monotonically increasing function, one can take the log of the likelihood function to create the log-likelihood function and not affect the results of the maximization procedure:

$$\begin{aligned}\log\left[\prod_{d=1}^{N} \mathrm{P}(A = a_d, C = c_d, I = i_d, F = f_d, P = p_d, S = s_d)\right] = \\ \sum_{i} N_{C=i} \log C_i + \sum_{i} N_{F=i} \log F_i + \sum_{i,j} N_{P=i,C=j} \log P_{ij} + \\ \sum_{i,j} N_{A=i,P=j} \log A_{ij} + \sum_{i,j,k} N_{S=i,P=j,F=k} \log S_{ijk} + \\ \sum_{i,j,k.l} N_{I=i,A=j,S=k,F=l} \log I_{ijkl}.\end{aligned} \tag{2.5}$$

Let the log-likelihood (Equation 2.5) be referred to as $\mathcal{L}(\mathbf{X}, \theta)$ where $\mathbf{X}$ is the set of all the training data (N vectors of length 6 in this example) and θ represents the CPD table entries ($C_i, F_i, P_{ij}, A_{ij}, S_{ijk}$, and I_{ijkl}). The likelihood must be maximized, but this maximization must be constrained so that the row of each CPD table sums

to 1. Therefore, the following contraints must hold:

$$\sum_i C_i = 1, \tag{2.6}$$

$$\sum_i F_i = 1, \tag{2.7}$$

$$\sum_i P_{ij} = 1 \quad \forall j, \tag{2.8}$$

$$\sum_i A_{ij} = 1 \quad \forall j, \tag{2.9}$$

$$\sum_i S_{ijk} = 1 \quad \forall j, k, \tag{2.10}$$

$$\text{and} \quad \sum_i I_{ijkl} = 1 \quad \forall j, k, l. \tag{2.11}$$

In summary, the goal is to maximize $\mathcal{L}(\mathbf{X}, \theta)$ subject to Equations 2.6-2.11. A simple way to do this is to utilize Lagrange multipliers,[42,46] so the following will be maximized:

$$\begin{aligned} \mathcal{L}(\mathbf{X}, \theta) - \lambda_1 \left(\sum_i C_i = 1 \right) - \lambda_2 \left(\sum_i F_i = 1 \right) - \lambda_3 \left(\sum_i P_{ij} = 1 \right) \\ - \lambda_4 \left(\sum_i A_{ij} = 1 \right) - \lambda_5 \left(\sum_i S_{ijk} = 1 \right) - \lambda_6 \left(\sum_i I_{ijkl} = 1 \right). \end{aligned} \tag{2.12}$$

Maximizing Expression 2.12 involves taking the partial derivatives with respect to each of the parameters. Note that each of the parameters only shows up in a single term in $\mathcal{L}(\mathbf{X}, \theta)$ and only one of the constraints. Therefore, taking each partial derivative only leaves a simple result. For example, the following is the result when

the partial derivative of Expression 2.12 is taken with respect to parameter C_i and set equal to 0:

$$\frac{N_{C=i}}{C_i} - \lambda_1 = 0 \tag{2.13}$$

$$C_i = \frac{N_{C=i}}{\lambda_1}. \tag{2.14}$$

Combining the constraint in Equation 2.6 with Equation 2.14 yields the following:

$$\sum_i \frac{N_{C=i}}{\lambda_1} = 1 \tag{2.15}$$

$$\lambda_1 = \sum_i N_{C=i}. \tag{2.16}$$

Therefore,

$$C_i = \frac{N_{C=i}}{\sum_i N_{C=i}}, \tag{2.17}$$

which means that in order to estimate the probability of the node Cloudy being true, one just counts how many times node C is true in the data and divides by the total number of times node C is true or false (the number of total data points in the training data).

The same basic process holds for other nodes in the graph as well, but nodes having parents (and therefore conditional probability parameters) are slightly different. Take, for example, node P which describes whether there is any Precipitation in the model.

Following the same process, the partial derivative of Expression 2.12 with respect to P_{ij} is below:

$$\frac{N_{P=i,C=j}}{P_{ij}} - \lambda_3 = 0 \tag{2.18}$$

$$P_{ij} = \frac{N_{P=i,C=j}}{\lambda_3}. \tag{2.19}$$

Combining Equation 2.19 with the constraint in Equation 2.8 gives the following result:

$$\sum_i \frac{N_{P=i,C=j}}{\lambda_3} = 1 \tag{2.20}$$

$$\lambda_3 = \sum_i N_{P=i,C=j}. \tag{2.21}$$

Therefore,

$$P_{ij} = \frac{N_{P=i,C=j}}{\sum_i N_{P=i,C=j}}, \tag{2.22}$$

which again means that counting provides the estimate for the probability of Precipitation given whether the sky is Cloudy in this model. For a given value of the node Cloudy the counts for Precipitation are collected. For the case where Cloudy is true the probability of Precipitation being true is the number of times in the data where Cloudy is false and Precipitation is true divided by the total number of times in the

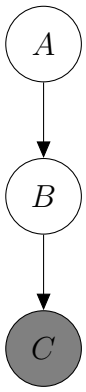
data where Cloudy is false. That process must then be repeated for all the other cases (of course taking into account the fact that each conditional distribution must sum to 1 so some values can be trivially filled in once others are obtained).

Learning in discrete Bayesian networks with categorical CPD tables is straightforward when all the data are available and involves keeping track of simple counts in the training data. However, there are many other cases that complicate matters. Situations where the values of some nodes are unknown in the training data are common and can be dealt with by utilizing other techniques. In addition, utilizing different probability distributions in the model (other than categorical) create other learning procedures even when utilizing the same maximum likelihood estimation framework. The process is similar in that the likelihood function is maximized, but sometimes it is difficult to derive a closed-form solution for learning. This thesis does not cover the myriad other situations that can arise in learning, but this section is intended to give the reader a basic understanding of how learning in the models covered in this thesis can occur.

2.2 Inference

Once the model is trained it can be used to estimate the probability of other events occurring. Some questions are trivial given the network structure. For example, asking for the probability of significant accumulation given that precipitation is

present is simple because that information is stored directly in the CPD table for the Accumulation node (in this case the answer is 0.2). However, other questions can also be asked that are not as obvious to answer. For example, one can ask for the probability that the ground is frozen given that the road is icy. Or one can query the probability that salt was put down on the road given that it is icy. Furthermore, it is possible to determine the most likely configuration of some variables given that others might have specific values.



$A=0$	$A=1$
0.3	0.7

	$B=0$	$B=1$
$A=0$	0.2	0.8
$A=1$	0.9	0.1

	$C=0$	$C=1$
$B=0$	0.4	0.6
$B=1$	0.2	0.8

Figure 2.2: Simple ABC Bayesian Network. This directed network contains three nodes, each of which are binary. Node C is observed (known) and inference can be performed on nodes A and B. In this example $C = 0$. The CPD table for each node is shown to the right of the network itself.

There are multiple ways to attack the challenge of performing inference.[42,43,45] Different techniques work better for some networks than others, and some techniques only apply to specific types of networks. Ideally an exact solution is sought (see

Section 2.2.1), but sometimes that is not a tractable task and approximate solutions (see Section 2.2.2) can be used instead.

In order to describe a few important inference procedures and to contrast their differences a simple binary Bayesian network has been created. Figure 2.2 contains the ABC model as well as its associated CPD tables. This model contains three random variables, A, B, and C, each of which are binary. Node C is an observed node (shaded) which means its value is known. The factorization built into the model is the following:

$$\mathrm{P}(A, B, C) = \mathrm{P}(A)\mathrm{P}(B|A)\mathrm{P}(C|B). \tag{2.23}$$

Assume this network is already trained, so the CPD tables have already been specified based on the training data collected for this network. Each CPD entry described here is listed with the probability of the node being '0' first and the probability of being '1' second.

Since C is known (assume for this example that its value is 0), inference can be performed to determine the probability of nodes A and B taking on the values 0 and 1. More explicitly, the values of $\mathrm{P}(A = 1|C = 0)$ and $\mathrm{P}(B = 1|C = 0)$ can be determined given that the following probability distributions are specified by the trained model: $\mathrm{P}(A)$, $\mathrm{P}(B|A)$, and $\mathrm{P}(C|B)$.

2.2.1 Exact Inference

The simplest way to perform exact inference is to use the laws of conditional probability and marginalization of probability distributions. Since the ABC network in Figure 2.2 is so small, does not contain very many connections, and does not have a large set of values each node can be, it is straightforward to perform these computations. Focusing on the conditional distribution of node A given the evidence at node C, the laws of conditional probability provide the following starting point:

$$\mathrm{P}(A = 1|C = 0) = \frac{\mathrm{P}(A = 1, C = 0)}{\mathrm{P}(C = 0)}. \tag{2.24}$$

The top and bottom probability values can be calculated by marginalizing the joint distribution of the model:

$$\mathrm{P}(A = 1|C = 0) = \frac{\sum_{b=0}^{1} \mathrm{P}(A = 1, B = b, C = 0)}{\sum_{a=0}^{1} \sum_{b=0}^{1} \mathrm{P}(A = a, B = b, C = 0)}. \tag{2.25}$$

Since the network contains only binary nodes the joint distribution $\mathrm{P}(A, B, C)$ is a table with $2^3 = 8$ entries. However, as the size of the network increases the table grows exponentially and quickly becomes unwieldy. For example, even a modest network with 100 binary nodes would yield a table with 2^{100} entries which would not fit in the combined memory of all the computers in existence. Luckily, though, the hierarchical network structure provides a scalability to the model by providing local factors that

are of manageable size. Therefore, terms can be expanded by using the factorization shown graphically in the model and in Equation 2.23:

$$\mathrm{P}(A=1|C=0) = \frac{\sum_{b=0}^{1} \mathrm{P}(A=1)\mathrm{P}(B=b|A=1)\mathrm{P}(C=0|B=b)}{\sum_{a=0}^{1}\sum_{b=0}^{1} \mathrm{P}(A=a)\mathrm{P}(B=b|A=a)\mathrm{P}(C=0|B=b)}. \quad (2.26)$$

Probability values can be found in the CPD tables shown in Figure 2.2. Filling them into the above equation yields the following result:

$$\mathrm{P}(A=1|C=0) = \frac{0.7 \cdot 0.9 \cdot 0.4 + 0.7 \cdot 0.1 \cdot 0.2}{0.3 \cdot 0.2 \cdot 0.4 + 0.3 \cdot 0.8 \cdot 0.2 + 0.7 \cdot 0.9 \cdot 0.4 + 0.7 \cdot 0.1 \cdot 0.2}. \quad (2.27)$$

So $\mathrm{P}(A=1|C=0) \simeq 0.79$ and $\mathrm{P}(A=0|C=0) \simeq 0.21$. $\mathrm{P}(B=1|C=0)$ and $\mathrm{P}(B=0|C=0)$ can be computed in a similar manner.

Performing exact inference in this naive way can be easily done in this simple network, but as the network complexity increases the number of multiplications and summations that must be achieved increase immensely and may take a very long time to compute or can even become intractable. Luckily there are some computational efficiencies that can be taken advantage of to make the process better. For example, Equation 2.26 can be simplified in the following manner by moving the summations inward:

$$\mathrm{P}(A=1|C=0) = \frac{\mathrm{P}(A=1)\sum_{b=0}^{1} \mathrm{P}(B=b|A=1)\mathrm{P}(C=0|B=b)}{\sum_{a=0}^{1}\mathrm{P}(A=a)\sum_{b=0}^{1} \mathrm{P}(B=b|A=a)\mathrm{P}(C=0|B=b)}. \quad (2.28)$$

In this network there is not much gain from moving the summations in, but as the network size increases that technique becomes essential. For example, in a network with 50,000 nodes, many nodes are likely not connected to very many other nodes. In those situations summations relating to the factors for those nodes can be moved very far through the exact inference expressions and can save a great deal of computation.

This general idea of moving summations inward forms the basis of some general algorithms used to perform inference in directed acyclic graphs.[42,45] The basic case of this is called *variable elimination*. Some tweaks on the variable elimination concepts yield the *sum product* algorithm, and by turning the summations into maximization operators and keeping track of which values in the model yield those maximum values the most likely configuration of the model can be determined. That tweak is the *max product* algorithm. Further changes by dividing the problem into locally-held beliefs and allowing new information (updated values of random variables in the model) to flow yield the popular *belief propagation* procedure.

Despite these improvements in computation complexity, networks can still become intractable to perform exact inference on. This is often due to a combination of networks having so many local connections and so many nodes in general that all the local computations still take a very long time to compute. Other times exact inference is not preferred because the application demands faster inference results than exact inference can provide. Alternative options to perform approximate inference are described in Section 2.2.2, and this thesis focuses on these types of solutions in

a neuromorphic way by performing approximate inference using spiking neurons in parallel on brain-inspired hardware.

2.2.2 Approximate Inference

There are myriad ways to perform approximate inference, but perhaps the most obvious algorithm is *loopy belief propagation*.[42] The procedure is very similar to belief propagation as briefly described in Section 2.2.1. However, whereas belief propagation is guaranteed to converge to the correct result in two passes of sent messages in directed acyclic graphs, loopy belief propagation applies in situations where there are cycles in the graph. The procedure works essentially the same way as belief propagation except that messages are passed until the updates become small enough for convergence to occur. Convergence is not guaranteed for all networks, but it can nevertheless be successfully applied for some models.

Another class of inference techniques comprises various optimization techniques to approximate the distribution of interest.[42] These techniques, including various forms of *variational inference*, essentially provide a framework with which to perform inference on a distribution that is more tractable than the original distribution of interest.

Instead of changing the distribution itself it is also possible to approximate the original distribution by creating *particles* that in aggregate converge to the inference queries in question. These algorithms are called *particle methods*, and they comprise

a wide variety of *sampling* techniques.[42] The concept of aggregating values and computing their histogram will be revisited many times in this thesis as it forms the basis for the approximate inference techniques discussed here.

Some particle methods are very straightforward such as *rejection sampling*, where in directed acyclic graphs node values are chosen starting from prior nodes (nodes with no parents). Then, traveling down the tree of the model (following the arrows), node values are sampled until all the nodes have a value. Each of these samples is created based on the distribution at that node given the other values already sampled in the network. In other words the CPD table at each node is used to generate each sample. If any sample does not match the evidence (for example $C = 0$ in the ABC network shown in Figure 2.2) the current set of samples are thrown out. Once a sample is created for each node in the network the configuration of the nodes of interest are stored. These values are aggregated, and their relative frequencies of occurrence represent the probability of those nodes taking on those values.

An improvement to rejection sampling is called *likelihood weighting*.[42] Rather than allow some samples to be thrown out which wastes computations, this technique forces the evidence to be its observed value. However, samples are then weighted by the probability of the evidence being true given the other sampled values in the network.

Importance sampling allows for sampling from a different distribution than the one of interest[42] which can make inference tractable. It can also be used to focus on areas of a distribution that are unlikely and therefore rarely produce samples using

other techniques.

Other approximate inference techniques are Markov Chain Monte Carlo (MCMC) algorithms. All MCMC techniques[47] rely on the construction of a Markov chain[48] (see Figure 2.3). Markov chains are simply graphical models where as time goes on each node only depends on the value of the previous node. In Figure 2.3, each random variable in the chain $(Y_1, Y_2, Y_3, ..., Y_N)$ represents a complete configuration of another graphical model. For example, consider using an MCMC technique on the ABC network described in Figure 2.2. In that situation for the MCMC techniques described in this thesis, Y_1 would refer to a configuration of the variables A, B, and C, and Y_2 would refer to another configuration of the variables A, B, and C at a later time in the evolution of the chain and the sampling process.

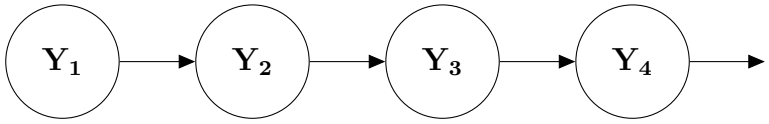


Figure 2.3: Typical Markov Chain.

Samples from the Markov chain are created over time, meaning that another configuration of the original graphical model follows from the one before it.[42,49] This Markov chain is designed so that over time the samples are generated from a *stationary distribution* the chain has converged to. The Markov chain samples can then be used to perform approximate inference by estimating the conditional distribution of the nodes of interest given the evidence in the original model. The following sections

describe first how a typical MCMC algorithm called Gibbs sampling can be used in Bayesian networks to perform inference (see Section 2.2.2.1). Then a way to perform brain-inspired computations to achieve the same goal is explained in Section 2.2.2.2.

2.2.2.1 Gibbs Sampling

Gibbs Sampling[42,49] is an MCMC technique that is a special form of the Metropolis-Hastings algorithm. As described earlier, each node in the Markov chain (see Figure 2.3) represents the full configuration of the original network. The first node in the Markov chain corresponds to a random initialization of the nodes in the original model (except for the observed nodes which are fixed to their known values). Then, during each sampling iteration each unobserved node in the network is sampled one at a time. Assuming that there are N nodes $X_1, X_2, ..., X_N$ having current values $x_1, x_2, ..., x_N$ in the network, they are sampled according to the following distribution:

$$x_i \sim \mathrm{P}(x_i | x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_N), \tag{2.29}$$

where i goes from 1 to N, and each variable x_j is the most recently-sampled value for that node X_j. Each iteration consists of running through all N nodes, so each iteration consists of N steps in the Markov chain.

This Gibbs sampling construction ensures that the Markov chain has a unique stationary distribution that is proportional to the distribution of interest,[50] which in this

case is the joint distribution described by the original Bayesian network. Therefore, by aggregating all the samples over time and viewing their histograms, it is possible to perform inference tasks because these relative frequencies of values occurring estimate the probability of the nodes being those values given the known evidence in the model.

Now, note that the distribution described in Expression 2.29 can often be greatly simplified because the conditional probability of a node given its Markov Blanket is independent of all the other nodes in the network.[42] Therefore, the distribution in Expression 2.29 only requires computations done using the node x_i's Markov Blanket.

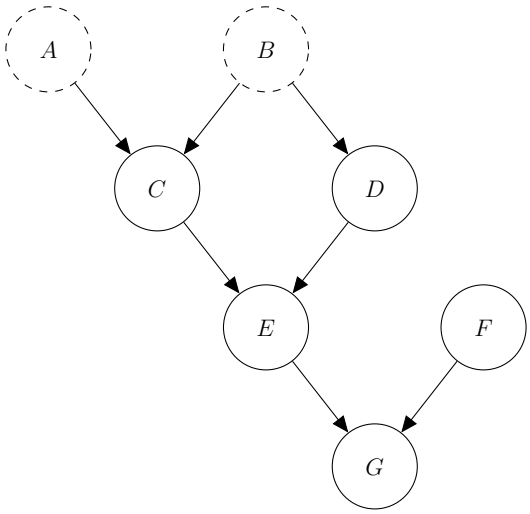


Figure 2.4: Example Markov Blanket in a Bayesian network.

For example, consider the network shown in Figure 2.4. This network depicts the Markov Blanket for node E. The Markov Blanket consists of the node's parents, children, and coparents (nodes that are parents of the same node as the node in

question). Therefore, nodes A and B are not part of the Markov Blanket and can be ignored when sampling E using Gibbs sampling.

Put another way, consider sampling node E using Expression 2.29. Following the rules of conditional probability the following is true:

$$\mathrm{P}(E|A,B,C,D,F,G) = \frac{\mathrm{P}(A,B,C,D,E,F,G)}{\mathrm{P}(A,B,C,D,F,G)}. \tag{2.30}$$

Based on the conditional independence structure of the model, the joint distribution can be factored in the following manner:

$$\mathrm{P}(A,B,C,D,E,F,G) = \mathrm{P}(A)\mathrm{P}(B)\mathrm{P}(C|A,B)\mathrm{P}(D|B)\mathrm{P}(E|C,D)\mathrm{P}(G|E,F). \tag{2.31}$$

Thus, continuing along from Equation 2.30, E can be sampled from the following distribution:

$$\begin{aligned}
\mathrm{P}(E|A,B,C,D,F,G) &= \frac{\mathrm{P}(A,B,C,D,E,F,G)}{\mathrm{P}(A,B,C,D,F,G)} \\
&= \frac{\mathrm{P}(A)\mathrm{P}(B)\mathrm{P}(C|A,B)\mathrm{P}(D|B)\mathrm{P}(E|C,D)\mathrm{P}(G|E,F)}{\sum_{e\in\mathcal{D}(E)}\mathrm{P}(A)\mathrm{P}(B)\mathrm{P}(C|A,B)\mathrm{P}(D|B)\mathrm{P}(E|C,D)\mathrm{P}(G|E,F)} \\
&= \frac{\mathrm{P}(A)\mathrm{P}(B)\mathrm{P}(C|A,B)\mathrm{P}(D|B)\mathrm{P}(E|C,D)\mathrm{P}(G|E,F)}{\mathrm{P}(A)\mathrm{P}(B)\mathrm{P}(C|A,B)\mathrm{P}(D|B)\sum_{e\in\mathcal{D}(E)}\mathrm{P}(E|C,D)\mathrm{P}(G|E,F)} \\
&= \frac{\mathrm{P}(E|C,D)\mathrm{P}(G|E,F)}{\sum_{e\in\mathcal{D}(E)}\mathrm{P}(E|C,D)\mathrm{P}(G|E,F)},
\end{aligned} \tag{2.32}$$

where in the above equations $\mathcal{D}(E)$ is the set of values node E can be. The only

nodes involved in the computation come from the Markov Blanket, as expected. In addition, note the factors involved in the computation. The factors are exactly the factors that can be formed with the members of the Markov Blanket, and they are also the only factors that involve the node that is to be sampled. This last point that the factors all include the node to be sampled is exactly why these nodes form the Markov Blanket. They are the only ones that do not drop out of the equation for the conditional probability of the current node given all the others.

The process of determining the update equations for Gibbs sampling is easily automated if the distributions are all discrete. For each node the Markov Blanket is determined, and then the factors completely included in the Markov Blanket are multiplied together in the numerator and denominator. Then the denominator is marginalized to remove the dependence on the current node, and the division is completed. The denominator is a single value after the sum, and the numerator can take on a number of values, each corresponding to a member of $\mathcal{D}(E)$.

On the other hand, if the conditional distribution for the current node given its parents is not discrete, the denominator turns into an integral and the result of Equation 2.32 is a continuous conditional distribution. This distribution may or may not be possible to express in closed form.

Once the Gibbs update distributions are calculated for each node in the network the sampling process can run. At each time step a new sample is generated for each node in order given the current values of all the other nodes. When Gibbs

sampling is used to sample nodes that are parameters for other distributions in the network there is a *burn-in* period before the model settles down and converges to a good likelihood for the model,[42] but in this work the model is assumed to be already trained and therefore all the samples can be used for inference from the beginning of the sampling process. Instead of converging to a better configuration of the random variables, in this work the sampling process is simply used to traverse the probability distribution that is already set in order to estimate the inference distributions in question. In addition, the samples are commonly correlated, so if an application depends on uncorrelated samples every nth sample is often taken. However, in the situations described in this thesis the samples are only viewed in aggregate to build a distribution so this factor is not important.

Now, going back to the simple 3-node discrete Bayesian network (the ABC network) shown in Figure 2.2, determining the sampling equation for each node using the form of Expression 2.29 is straightforward. Since there are three nodes and one is known, two distributions are required. These two distributions are the following: $\mathrm{P}(A|B=b,C=0)$, where b is the most recent sample for B, and $\mathrm{P}(B|A=a,C=0)$, where a is the most recent sample for A.

These sampling distributions can generally be greatly simplified by only considering the terms that include nodes in the Markov Blanket of the current node. There-

fore, the following simplification can be made:

$$\mathrm{P}(A|B=b,C=0)=\mathrm{P}(A|B=b) \tag{2.33}$$

because the Markov Blanket of A only contains B. On the other hand, $\mathrm{P}(B|A=a,C=0)$ cannot be simplified in this manner because both A and C are in the Markov Blanket of B.

Before the algorithm starts each node is initialized to a random value except for the observed nodes which are fixed. During every iteration each unobserved node is sampled one at a time, and the number of times each node takes on each value is recorded. Every sample relies on the most current value of every other node in the network. In this example network the following two distributions (in simplified form) are used for generating new samples:

$$\mathrm{P}(A|B=b) \text{ and } \mathrm{P}(B|A=a,C=0). \tag{2.34}$$

To elaborate, take the first sampling distribution:

$$\mathrm{P}(A|B=b)=\frac{\mathrm{P}(A)\mathrm{P}(B=b|A)}{\sum_{i=0}^{1}\mathrm{P}(A=i)\mathrm{P}(B=b|A=i)}. \tag{2.35}$$

If $b = 0$ then the following distribution is the sampling distribution for A:

$$\mathrm{P}(A = 0|B = 0) = \frac{0.3 \cdot 0.2}{0.3 \cdot 0.2 + 0.7 \cdot 0.9} \simeq 0.09 \tag{2.36}$$

$$\text{and} \quad \mathrm{P}(A = 1|B = 0) = \frac{0.7 \cdot 0.9}{0.3 \cdot 0.2 + 0.7 \cdot 0.9} \simeq 0.91. \tag{2.37}$$

Otherwise $b = 1$ and the following is the sampling distribution for A:

$$\mathrm{P}(A = 0|B = 1) = \frac{0.3 \cdot 0.8}{0.3 \cdot 0.8 + 0.7 \cdot 0.1} \simeq 0.77 \tag{2.38}$$

$$\text{and} \quad \mathrm{P}(A = 1|B = 1) = \frac{0.7 \cdot 0.1}{0.3 \cdot 0.8 + 0.7 \cdot 0.1} \simeq 0.23. \tag{2.39}$$

The sampling distributions for node B, $\mathrm{P}(B|A = a, C = 0)$, are calculated in a similar manner, in this case depending on the most recent sample for node A.

2.2.2.2 Neural Sampling

Neural sampling[51,52] is similar to Gibbs sampling in that it involves aggregating samples of random variables in an MCMC manner to perform inference in Bayesian networks. However, Neural sampling is currently limited to models with binary variables. On the other hand, Neural sampling is more biologically realistic than Gibbs sampling because it is designed to mimic neurons in the brain. Each random variable in the network is modeled as being a neuron with a membrane potential that influences the node's probability of spiking, and each neuron has a refractory period that

affects the frequency with which it can spike.

The first step in setting up the network is to convert the Bayesian network into an equivalent neuron network that can be used to perform sampling. This is done by determining the Markov Blanket for each node and creating new types of connections. For example, in the simple ABC network described in Figure 2.2, node B has nodes A and C in its Markov Blanket.

The new neural network structure is shown in Figure 2.5. This structure is set up in a specific manner so that the Markov Chain that is processed in this MCMC technique converges to the correct distribution over time. Each node is binary, so there are four possible permutations of values that the Markov Blanket can take on. These are represented by four variables, $\alpha 00, \alpha 01, \alpha 10$, and $\alpha 11$. The nodes $l00$, $l01$, and $l10$ are inhibitory nodes, and they fire when all the nodes connected to them are 0.

There is a pattern in the connections in the structure here. Whenever a node is supposed to be 0 for a particular Markov Blanket value permutation, that node in the Markov Blanket is connected to the l-node associated with that combination. On the other hand, when the node is supposed to be 1 it is connected to the α-node for that combination. That is why for the $\alpha 00$ node, neither A nor C is connected directly. Both of them are 0 for the 00 combination, so they are both connected to $l00$ which is instead connected to $\alpha 00$.

It has already been mentioned that the inhibitory l-nodes turn on when everything

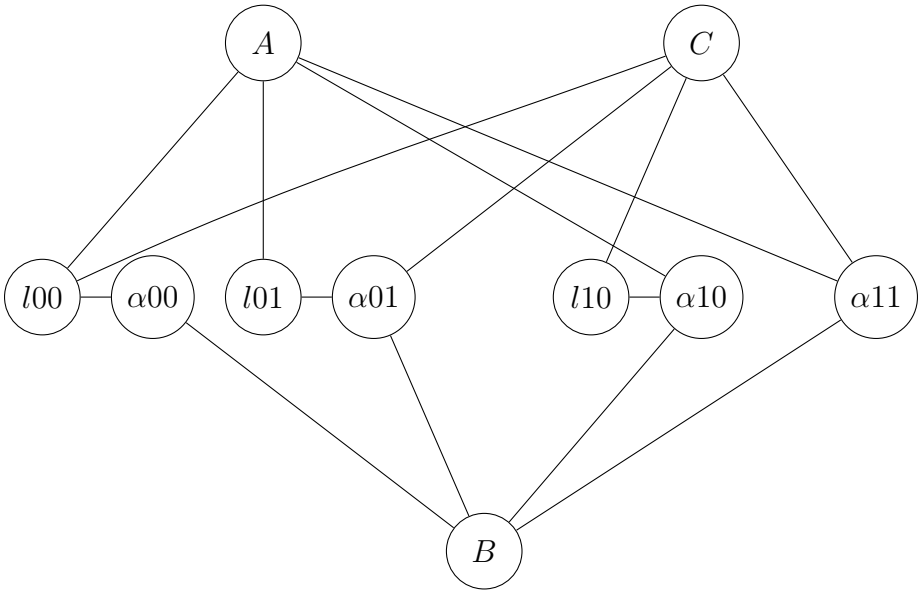


Figure 2.5: A conversion to neural network structure for node B assuming that it has two nodes, A and C, in its Markov Blanket.

connected to them is 0. However, for the α nodes the situation is different. These nodes each have a certain probability of firing, and this probability tends toward 0 whenever any of the nodes connected to them are 0.

However, when all the nodes connected to an α-node are 1, the node has a different probability of firing. This probability depends on the membrane potential, $u_i(t)$, at a particular time t of the neuron representing that node:

$$u_i(t) = \log \frac{\mathrm{P}(X_i = 1|\mathbf{X}_{\setminus i})}{\mathrm{P}(X_i = 0|\mathbf{X}_{\setminus i})}. \tag{2.40}$$

In this notation X_i is the ith random variable in the model containing the nodes $X_1, X_2, ..., X_N$. In Figure 2.5 the variable X_i is the node B, or the node for which

the Markov Blanket is examined. $\mathbf{X}_{\setminus i}$ represents all the nodes in the network that are not the current node, so $\mathbf{X}_{\setminus i}$ can mathematically be simplified to be the Markov Blanket of the current node X_i. This membrane potential value $u_i(t)$ is fed into a sigmoid function to determine the probability of that α-node firing:

$$p(\text{firing}) = \sigma(u_i(t) - \log \tau), \tag{2.41}$$

where $\sigma(x) = (1+e^{-x})^{-1}$. Here, τ is a constant chosen for the entire sampling process which represents how long the effects of a spike last. The spike creates a refractory process in the current neuron as well as a change in the membrane potential of other neurons that lasts on the order of 5 ms to 100 ms.[51,52] So τ represents both of these effects in milliseconds, and in this thesis τ has been fixed to be 20 ms.

Remember, though, that these nodes are represented by neurons. So instead of having this firing probability (Equation 2.41) be applicable all the time for each α-neuron, it only applies once τ ms has elapsed since the last time the neuron has fired. This is because the neuron has to go through its refractory process before it can fire again.

In reality, some neurons can fire a rapid burst of spikes and others must wait longer amounts of time in between spikes. This can be accommodated by using a modified version of this framework,[51,52] but in this work everything is restricted to the fixed refractory period case for simplicity and accuracy. The MCMC process

has only been proven to converge to the correct distribution for the fixed refractory period case despite the fact that local computations are correct in the flexible case and empirical results can be good.

So the way each main neuron (not the auxiliary or inhibitory ones) works is that if it has not fired in the last τ timesteps it can fire with the probability given in Equation 2.41. Once it fires this neuron cannot fire again until enough time has elapsed. However, from the time the neuron fires all the way through the amount of time it cannot spike, the neuron represents the value 1 in the network. This is why the value of τ is included in the firing probability (the firing probability must essentially be divided by τ since once a neuron fires it stays on for a time period of length τ).

This type of network structure is created for each neuron in the original Bayesian network, and all neurons are additionally connected together in the same manner as they are in the original network. Then the converted network can be sampled over time in a manner similar to Gibbs sampling. An order is chosen for the neurons, and they are updated over time. The number of times each neuron is on (including both actual spikes as well as the refractory periods) is tallied, and from those numbers the distribution can be approximated via a histogram after the burn-in period has elapsed.

Note that the entire network in Figure 2.5 essentially functions to choose the distribution from which node B samples. The excitatory and inhibitory connections

influence which of the α nodes can send a spike to node B which then makes node B spike. Thus the essential pieces of information are the node's Markov Blanket and the values of the nodes in that Markov Blanket for every node in the original Bayesian network.

Now, going back to the example ABC network shown in Figure 2.2, it is straightforward to set up the sampling distributions. Looking at node A of the model and considering Equation 2.40 as well as the Markov Blanket of node A, the following membrane potential is calculated:

$$u_a(t) = \log \frac{\mathrm{P}(A = 1|B = b)}{\mathrm{P}(A = 0|B = b)}. \tag{2.42}$$

So depending on the value of node B the membrane potential at A varies. Using that membrane potential the probability of the node firing must be calculated:

$$\mathrm{P}(\text{firing}) = \sigma(u_a(t) - \log \tau).$$

For example, in Equation 2.42, if $B = 0$ and $\tau = 20$ then the values for the numerator and denominator from Equation 2.36 and Equation 2.37 can be substituted. The results are the following:

$$u_a(t) = \log \frac{0.91}{0.09}$$

$$\text{and} \quad \mathrm{P}(\text{firing}) \simeq 0.34.$$

The same process must be completed for the other unobserved nodes in the network, so for this example the probability of firing for node B must be calculated in a similar way.

The probability of firing at each node only applies when the neuron is not in its refractory period. When a neuron fires it becomes refractory for τ iterations, so it cannot spike again until the refractory period is over.

2.2.3 Simple Inference Results

A general MATLAB framework was developed for describing Bayesian networks as well taking those networks and analyzing them in preparation for performing sampling. This analysis includes tasks such as determining which nodes are in each node's Markov Blanket, determining the sampling distributions automatically from the graph structure and the CPD tables, etc. This framework was also extended and improved later on in this thesis to work with neuromorphic hardware as described later. Gibbs sampling and Neural sampling were both implemented in MATLAB on top of this general framework so that arbitrary inference queries can be asked of various models.

To provide an idea about how these algorithms work in practice, inference was done on the simple ABC model in Figure 2.2 using four different techniques – exact inference, Gibbs sampling, Neural sampling, and another implementation of Gibbs sampling using Kevin Murphy's BayesNet Toolbox.[53]

Table 2.1 shows the probability of nodes A and B having the value 1 given that node $C = 0$ for each of the four techniques. The sampling code all ran for 50,000 iterations, and the results are reported from just one run of each algorithm. "BNT Gibbs" is the BayesNet Toolbox implementation of Gibbs sampling.

	$A = 1$	$B = 1$
Exact	0.787	0.183
BNT Gibbs	0.785	0.187
Gibbs	0.792	0.180
Neural Sampling	0.794	0.183

Table 2.1: Inference results using four different techniques.

Figure 2.6 shows what the first 1000 iterations look like for node B using this thesis' Gibbs and neural sampling implementations. In Gibbs sampling the node can potentially spike at any time, but in neural sampling once a spike occurs the node's value must remain 1 until τ iterations have passed. Note that despite these differences the two implementations produce the value 1 approximately the same amount of time overall.

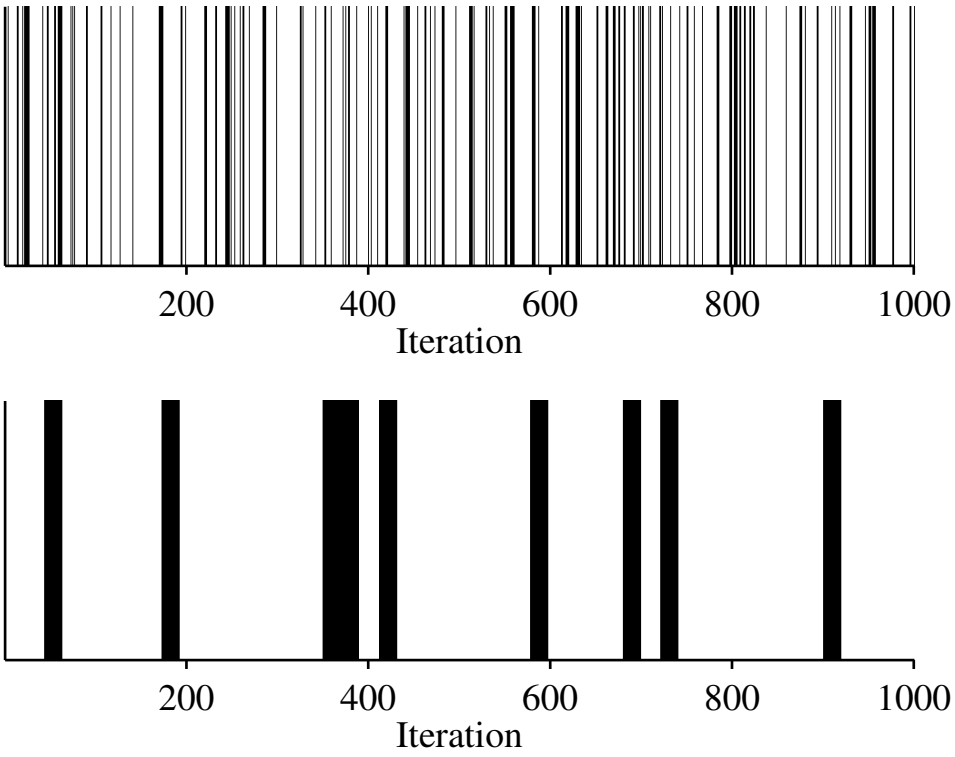


Figure 2.6: Samples at node B in the simple ABC network. The top of the graph comes from the Gibbs sampling implementation and the bottom is from the neural sampling implementation.

Chapter 3

Parallel Neural Sampling on SpiNNaker

The Spiking Neural Network Architecture (SpiNNaker)[21–23] was created at the University of Manchester. This board is a flexible platform that allows for massively-parallel event-based computation. There are two different boards worked on during this thesis which are shown in Figure 3.1. One contains four chips and the other is larger and contains 48 chips. Each chip contains contains 18 ARM968 cores clocked at about 200 MHz (configurable) which are capable of 32-bit fixed-point arithmetic. Each core has 32 kB of instruction tightly-coupled memory (ITCM) and 64 kB of data tightly-coupled memory (DTCM), and the 18 cores per chip all share one 128 MB block of SDRAM per chip. Multicast packet routing is supported in addition to point-to-point message passing, and the messages are sent through a routing network

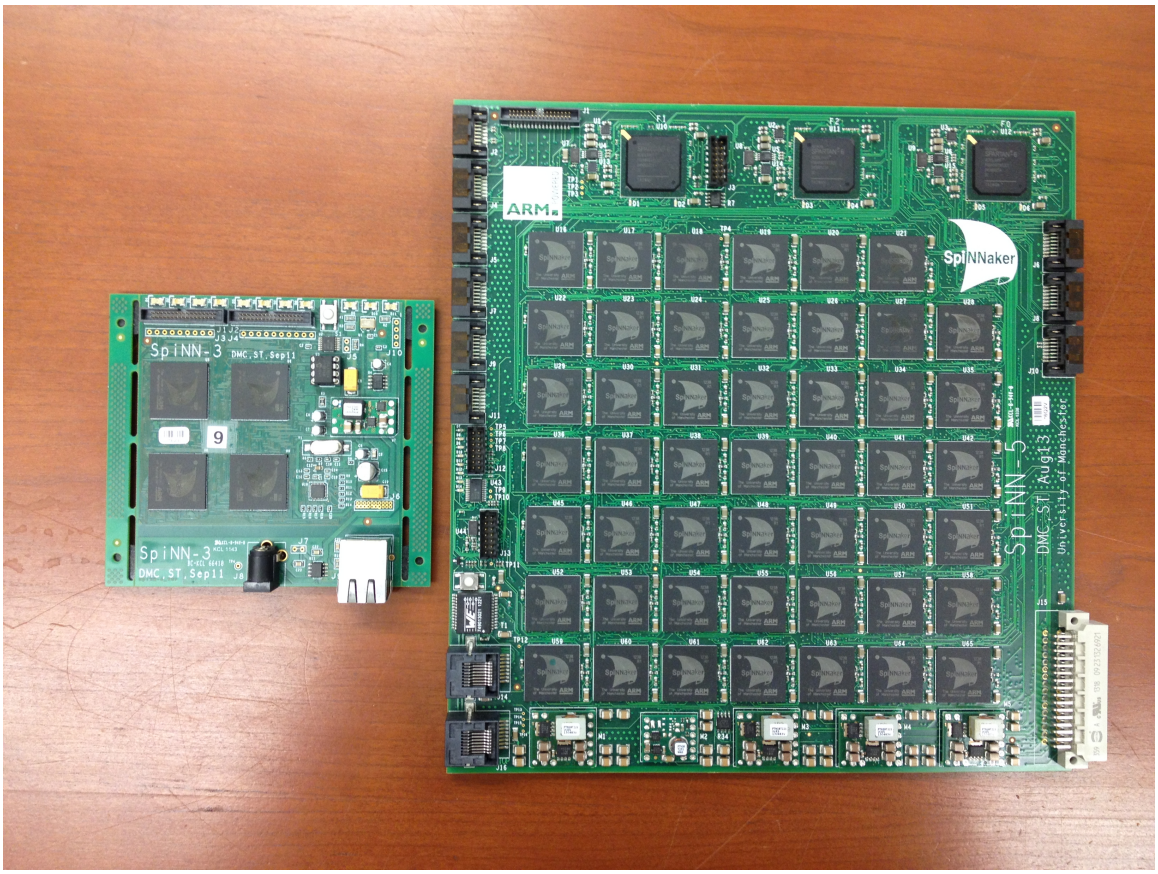


Figure 3.1: Two SpiNNaker boards. The left board is a smaller, 4-chip board with up to 72 available cores. The right board has 48 chips for up to 864 available cores.

operating at up to 1 gbps.[54]

These boards are designed to be programmed in an event-based manner so that each core shuts down when it is not actively processing. Then, when an event (incoming packet, timer tick, etc.) occurs it is either added to a queue until the current task is completed or interrupts the current process if the event has higher priority. The smaller 4-chip board runs on only 1 W of power.

This entire architecture was designed for the purpose of simulating spiking neurons in a realistic manner, so Neural sampling is a good candidate to be implemented on

this architecture. Each core only runs at 200 MHz, but with the fast communication interconnect it is possible to distribute loads in an interesting manner. Just as neurons do simple local computations and pass the results to other neurons, each core on this chip can do local computations and pass them on to other cores so that the whole board can do a great deal of work in aggregate.

Therefore, the goal of this portion of the thesis is to implement the Neural sampling algorithm on the SpiNNaker and see how well it performs. The limits of the board are tested, and the Neural sampling algorithm is implemented on large networks to see whether it works well in practice.

The following sections describe the theory and mechanics of transforming a Bayesian network into the proper type of neural network to perform sampling. There is also a discussion of the way the code and data are organized on the board as well as the process of actually performing sampling in the context of the SpiNNaker. The entire flow of processing the network and getting it on the SpiNNaker is an automated process. Once the network is described in a text file and the known values (evidence) in the network are specified, the process can begin. Various MATLAB functions were created which build the network structure from the text description, specify parameters, calculate sampling distributions for the SpiNNaker implementation, place the nodes on the hardware, generate routing paths, boot the SpiNNaker, send data to the SpiNNaker, and communicate via Ethernet with the SpiNNaker to retrieve the results of simulations once sampling is completed.

Some of the work in this chapter has been previously published[55,56] by the author of this thesis.

3.1 Automated Network Analysis

The first steps toward performing sampling on the SpiNNaker involve analyzing the probabilistic graphical model file. This analysis happens without further intervention by the user, and all the parameters required for setting up all the data structures on the SpiNNaker are established. The sections that immediately follow describe the process of preparing the network for sampling.

3.1.1 Converting the Network for Neural Sampling

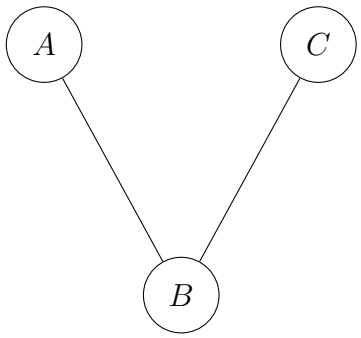


Figure 3.2: The converted Markov Blanket network used to simplify computations.

The addition of the extra neurons in Figure 2.5 clearly adds to the complexity of the simulation in terms of runtime as well as memory. Instead of updating only the nodes that were in the original Bayesian network, the auxiliary and inhibitory

nodes also must be sampled. However, by looking at the original network it is pretty clear what that structure accomplishes. The inhibitory and auxiliary neurons (l- and α-neurons, respectively) are used to select which probability distribution should be used for the sampling of the bottom node (in this case node B). The α-neurons are either on or off with a probability that only applies when the Markov Blanket matches the correct pattern, and then the node at the bottom is simply designed to follow the actions of the α-neurons. Since only one α-neuron is permitted to be on at a time, all the extra neurons are essentially just choosing the distribution from which node B is sampled.

As a result, the network has been simplified in this work to that of Figure 3.2. This network is much simpler in terms of the number of neurons that need to be sampled, but it performs the same function. In this case the main neuron, node B, must be smarter, but all the inhibitory and auxiliary neurons are not sampled.

With the simplified network structure the main node needs to take care of the work of the auxiliary neurons. So with this modified architecture, the main node needs to determine the values of the nodes in the Markov Blanket. It also must store a table of conditional probability distributions for itself given its set of Markov Blanket values. Then, depending on the collective Markov Blanket value, the main node performs the same function the corresponding auxiliary α-neuron would. So if the main neuron already fired it remains in the state 1 and will not fire again, and if it has not fired recently the main neuron fires with the probability the auxiliary

neuron would have fired.

So this network is not exactly the same as the original network, but it attempts to emulates the same basic functionality by using less resources and reducing the number of nodes that need to be simulated. The smaller number of node updates translates directly into faster execution of the sampling algorithm.

3.1.2 Parallelization and Colorization

Gibbs sampling requires that, during each iteration, each node in the model is sampled from one at a time.[49] The order can change from iteration to iteration, but nodes must be updated one after another. In the general case, samples are generated from each node in order, and this process is repeated as many times as necessary for adequate convergence. However, implementing this algorithm on parallel hardware such as the SpiNNaker requires parallelizing the sampling process in order to distribute computations across computational units to save time and power.

One can think of this issue in the following manner. Imagine there is a node that is in state 1 with very low probability and is in state 0 with very high probability given the current state of the network, and let the current state of the network have a very high probability. If all the nodes are updated in parallel, the most likely update is that all the nodes stay in their high probability states. Now, imagine that it is the case that if that main node changes to its low-probability state, state 1, that some of the rest of the network is more likely to take on a different state. If that is the case and

if all nodes are updated simultaneously, if that one node changes state, that change is erroneously not reflected in the sampling probabilities for the other nodes until the next time around. Therefore, the other nodes will most likely stay the same until the next iteration, and then that original node will go back to its higher-probability state as a result.

On the other hand, if the nodes are updated in a specified order, the change in the main node will be reflected in the sampling probabilities of all the other nodes that come after it, and this change may remain for a while. So there is clearly a difference between sampling all the nodes in parallel and sampling them one at a time, and Gibbs sampling has been proven to converge in the case where all the nodes are updated in order. Therefore, one cannot just parallelize the whole network and expect the sampling to work correctly.

However, it is still possible to parallelize Gibbs sampling without affecting the convergence.[57] The technique is to split the graph into different groups of nodes, or colors, that can each be updated in parallel. If there are N colors, then the simulation updates each of the nodes in each of the N colors in parallel, does the same for the next color, and so on. This process continues until convergence.

The way to determine which nodes can be updated in parallel is intuitive. Basically, any node that is completely independent of the rest of the nodes in a given color (given its Markov Blanket) can be added to that color and updated along with the rest of that color in parallel. It is clear that if a node is in any other node's Markov

Blanket that the two nodes cannot be in the same color. Other than that restriction, though, the nodes can be updated in parallel.

Essentially, there is still a fixed order between nodes that directly depend on each other's values, and parallelizing the nodes that do not depend on each other does not affect the reasoning behind having Gibbs sampling ordered. Therefore, it is feasible to parallelize the sampling computations on neuromorphic hardware such as the SpiNNaker.

3.1.3 Node Organization on the SpiNNaker

One of the first considerations regarding performing sampling on the SpiNNaker is the location of each node on the physical hardware. Obviously some performance gains can be had by placing nodes closer together that need to communicate frequently, but that is explored later on in this thesis with the implementation on the larger 48-chip SpiNNaker. This section describes earlier work which is a simpler design on the 4-chip board.

In this basic location implementation the nodes are simply arranged around the board in an ordered manner that has nothing to do with the actual structure of the network. First, the nodes are arranged in order of color with the original ordering preserved within each color group. Then the nodes are distributed, in that order, in a pattern where core 1 is filled on all the chips, then core 2 is filled on all the chips, and so on up to and including core 16 on all the chips. If there are still unallocated

nodes left then the process starts over from the beginning.

Because this technique does not take into account spatial locality and thus locality of spikes, many spikes must be sent between chips rather than being confined to one chip or even core. Since this information must be sent between chips each time the nodes must be updated, the amount of information sent around the board is larger than if locality was preserved and nodes close to one another were contained within the same cores, for example.

Now, recall that the board is set up to easily allow for event-based processing. The architecture and software application programming interface (API) are both designed to encourage this style of programming, and Neural sampling, once parallelized, will work similarly to the way neurons work in the real world. Each neuron needs to accept inputs from those around itself and then, with a certain probability, spike. Therefore, there are a number of items each neuron must keep track of in order for this system to work. The following is a list of important values each core must know:

Number of colors in the network This is necessary so each core knows how many times it must iterate through neuron updates for each color before moving on to the next time step.

Color of each node Nodes of the same color can be updated simultaneously, so the color of each node must be known.

Nodes on current core Each core must know what nodes it is simulating.

Node values Each neuron must know whether it is an observed node with a specific value (evidence) or an unobserved node and that node's current sampled value.

Markov Blanket Each neuron must know which nodes are in its Markov Blanket (MB) as well as their current values so that it can store the proper values when spikes come in and determine the distribution with which it should produce samples.

Nodes whose Markov Blanket includes the current node Each neuron must know where to send its spikes. These other nodes all have the current node in their Markov Blankets, so they must be informed of the current node's value.

Firing probabilities For each combination of MB values the neuron must know what its probability of firing (sampling distribution) is.

3.2 Code Organization and Data/Event Flow

In the original version of this implementation, each core had its own custom-generated C code that ran on the board. MATLAB was used to automatically convert the original Bayesian network to its neural equivalent and to determine all the parameters needed for each core to keep track of all the nodes it is simulating as well as the associated information each node needs to know.

Creating, compiling, and then sending the compiled files, one for each individual core, to the SpiNNaker is a slow process, and it does not really make sense to do things that way. When all the data are written out in custom code for each core, instruction memory is wasted by having extra lines of code. Therefore, an improved technique has been implemented since then which is described in the following sections.

3.2.1 Putting Data on the Board

The better way to send code and data to the board is to just have one C file that is compiled and sent to the board. The C code is compiled with an ARM cross-compiler (it is compiled on an x86 host machine but compiled for the ARM cores located on the SpiNNaker). This compiled file is then sent around the board to all the cores using functions included in some Perl code written by the Advanced Processor Technology (APT) group at the University of Manchester. This Perl code also starts an application written by the APT group that waits for SpiNNaker Datagram Protocol (SDP) packets on a particular port. When it receives these messages it displays them, providing a sort of standard output for the C programs which is useful for debugging and all sorts of display purposes.

These commands are all automated in a shell script that is generic and relies only on parameter files created by the MATLAB framework that reads in all the trained network files and sets everything up. The script also programs the SpiNNaker with the IP address of the host computer so that it knows where to send packets for display

as well as the packets containing the results of the sampling simulations. Once these basic commands are executed the board is bootstrapped using a configuration file provided by the APT group. The cores come online and the first core per chip to come online is designated as a "monitor" core which handles administrative tasks. The other cores on the chip can be used for general computations by the programmer.

The SDRAM for each chip is shared among all the cores on that chip. Therefore all the necessary data for each core are placed into the SDRAM, and then the code distributed to each core simply reads from its corresponding block of SDRAM to initialize everything it needs to know. The transmission of data to the SDRAM is controlled by Perl functions written by the APT group. Once all the data are transmitted another Perl function sends the board a message telling the cores to start executing their compiled code.

3.2.2 Communication

There are multiple ways to send messages around the SpiNNaker board, and these are described next. Section 3.2.3 describes how these messages are processed and used during the execution of parallel code on the board.

One way to communicate on the board is by sending multicast messages, which is the quickest way to send a message. Only 32 bits of data can be sent in each message, but the messages can be sent in multiple directions at once as it leaves each router on the board. In fact, a multicast message can be sent to all cores on the board if

desired.

There is a router located on each chip, and the routers are connected to each other in a particular way. The chips are arranged in a mesh grid, and each chip is connected to all its neighbors except those in the NW and SE directions (6 total connections going in and out of each chip).

The simplest way to set up a route for a multicast message is to give that particular route a key. Then, at each chip the direction a packet with that key designation should go is selected from a programmable list of router entries. So if the packet should turn right when it hits a particular chip, that behavior can be coded into that core's routing table. In addition, if the packet should stop at a certain chip so that it can be received by a certain core on that chip, that can also be coded into that chip's routing table.

There are 1024 different entries allowed in each routing table, so the number of paths a multicast packet can take is finite. However, for both the 4-chip and 48-chip boards this amount of entries covers all the possible destinations a packet could possibly go. This does become an issue when more than one 48-chip SpiNNaker board is used though.

Another way to send messages on the board is by using SDP packets. These packets allow messages to be sent to arbitrary locations without having to explicitly set up routing tables on each chip along the path of the route. The downside to this type of message is that the message takes longer to arrive. An advantage is that messages of sizes up to 256 bytes can be sent using this technique.

Another interesting feature of SDP packets is that they can be used to send messages over the network via the Ethernet port on the board. This is a very useful property for getting data back from the board so that results can be stored on the host PC for analysis. This capability is also used for sending messages mid-simulation back to the host PC for debugging and general display purposes.

3.2.3 Interrupts and Event-Based Programming

This section describes the flow of programs on the board including how executables deal with messages that are received.

The SpiNNaker is designed to be a parallel, event-based architecture. Utilizing this event-based programming methodology yields great power savings over running code sequentially because the ARM cores go to sleep when they are idle, so the increased complexity of coding these parallel algorithms can be worth tackling.

In addition, the event-based paradigm allows the cores to work together to solve a problem which is one of the main draws of the device. Each core is not very fast on its own, but when they are used together to do smaller computations in parallel that communicate with each other this architecture can be useful. Running completely isolated code in parallel is not the best use of this board design though because that could be done much faster on a more powerful cluster (power consumption would be worse, but in terms of speed that type of system would be better than the SpiNNaker). One of the main benefits of the SpiNNaker is its fast built-in communication network,

so collaborative problems are better suited to its architecture.

Programs start off in the same basic manner as any sequential application. However, there are five different event types that can be used to generate an interrupt in the current program execution. When an interrupt is triggered, an associated callback function is called in order to deal with the event that occurred. For example, if a message is received then a function can be established that will save the incoming data so the program can utilize that information later on.

The five events on the SpiNNaker are the following:

1. MC packet received - This is the event that is triggered when a multicast (MC) packet arrives at the current core. Typically the callback function associated with this event would save the incoming data to a buffer so it can be used later on. Exiting this function as quickly as possible is beneficial to ensure that no incoming messages are dropped if the queue becomes full.

2. DMA transfer done - This is the event triggered when a direct memory access (DMA) operation has completed. So if the program initiates a DMA transfer from the shared SDRAM to the data memory for a particular core, this event happens when the data have been successfully copied. This is an important event because it ensures that the data have arrived before the program tries to access the local copy, and it frees up the processor to perform other work instead of blocking while data are transferred.

3. Timer tick - This event is triggered over and over again with a certain time interval between (in microsecond units) which is set by the programmer. This associated callback is typically the main processing function used when running jobs on the SpiNNaker. A normal procedure is to have a processing function run every so often at each timer tick, and messages are sent at the end of that task. When every core gets its incoming messages the data are stored, and then the timer tick occurs again and triggers the next round of processing.

   The reason why the timer tick is usually used for data processing is that the architecture was mainly designed to perform simulations of neurons in real-time. This means that the computations for each neuron should update every millisecond, and all the messages (spikes) should be sent and stored within that time frame as well. When this paradigm is adapted to other problems this timer tick period can be altered depending on how long the computations take to run at each time step. These ticks are synchronized on one board, but independent boards are not perfectly synchronized. Therefore, algorithms that scale to larger collections of boards must be tolerant of this situation.

4. SDP packet received - This event is triggered when an SDP packet is received at a given core. Depending on what is running on all the cores this packet can either be from another core or from an outside host via Ethernet. Just as for the multicast packet received event, the callback for this event is often used to record the received data or perhaps reply to the message, especially in the case

when the packet comes from the host PC which is requesting data.

5. User event - This even is triggered by the application, so it is very free-form and can be used for a variety of custom purposes.

Events can be set to have different priorities. Therefore, when an event occurs it is added to a priority queue, and the ones with the highest priority are dealt with first by calling their associated callback functions. The highest-priority events are often MC message received events because these messages may disappear if they are not dealt with in time and more messages arrive. However, this functionality can be changed depending on the requirements of any particular application running on one of the ARM cores.

3.2.4 Code Organization for Neural Sampling

As described in Section 3.2.3, the SpiNNaker is designed to be programmed in an event-based manner. So the first thing that is done is to set up three functions that each run when a corresponding interrupt is triggered:

1. Timer tick: This is the main function that runs at each time step and does the actual generation of samples on the board. The first thing it does is to copy all the incoming spike data to a different buffer so it will not be contaminated by other new spikes from other neurons. Then this function generates new samples for all the nodes of the current color. Spikes are sent to other nodes that need

to know what this core's node values are. Finally, bookkeeping is done to keep track of the node values over time which enables computing probability values at the end, and the simulation moves on to the next color's nodes. Once all the colors have been exhausted, a new timestep is started and the process is repeated for all the colors until the simulation has run for its full duration.

2. Multicast packet received: This function runs whenever a core receives a multicast packet (spike) from other cores on the board. When this function runs, the core receiving the data needs to log the incoming spike which tells the receiving core what the sending node's current value is.

3. SDP packet received: This function runs whenever a message comes from the PC controlling the board. This functionality is described in the next section that details how data are sent back to the PC, but the majority of these packets are transmitted when the simulation is over and information is collected about the sampling process that was just completed.

Most of the work is done in the Timer tick function. The frequency with which this function is run depends on the size and complexity of the network being simulated. As the networks grow larger this function must run less frequently because each iteration takes longer to run as the number of generated samples, messages passed, and messages received grow. Unfortunately the SpiNNaker requires this value to be hard-coded, so it is necessary to tweak this parameter until the function runs as often

as it can without the SpiNNaker crashing or too many packets are dropped.

Each of the callback functions is assigned a priority, and ones with higher priority can interrupt others with lower priority. The multicast packet received function has the highest priority so that no incoming packets are lost due to computational delays. The timer tick function is medium priority and can be interrupted by the multicast packet received function in order to temporarily stop and save an incoming spike. Finally, the SDP packet received callback function has the lowest priority because once the simulation starts it is only used when gathering results at the end of execution. These packets are instructions from the host computer controlling the SpiNNaker and they simply instruct various cores to provide the number of spikes sampled from each node and related information.

3.2.5 Getting Data Back

Once the simulation has completed, the data have to be sent somewhere so they can be stored. Since the SpiNNaker board in its bare form is only connected to a computer via an Ethernet port, packets must be sent back to the PC controlling the board. SpiNNaker supports UDP communication, so that is used for sending the data back.

When the simulation completes, some UDP messages indicating so are sent back to the PC and displayed using software written to listen on a particular port. Then the user must manually tell the PC to send packets to the SpiNNaker that request

information about the results.

The MATLAB software that requests results sends UDP packets crafted so that they go to the chip/core combination that corresponds to where the current node is handled on the board for the simulation. Once that core receives the message it sends back the number of times in the simulation that particular neuron had value 1 along with the number of timesteps in the simulation. With those two pieces of information the conditional probability of that node being 1 given the evidence can be determined by doing a simple division operation. In addition, having the number of timesteps is useful to ensure that each node was sampled the proper number of times during simulation.

3.2.6 Summarized Flow for Neural Sampling

The first thing each core does is load all the parameters for the Neural sampling simulation. These parameters include how long the simulation runs, which nodes are on the current core, which routes outgoing packets need to take, etc.

The routing tables are then populated, and the event callbacks are established. The timer tick callback is where the math is done to determine whether each neuron on each core should spike at the current simulation timestep. After all the neuron states are calculated, those states are transmitted to other relevant cores via multicast packets. When a core receives a message (spike) containing the state of another neuron, that core's MC packet received callback is called to store the data for the

next timestep in the simulation.

This process continues until the simulation has completed, at which point a UDP packet is sent to the host PC that tells a program created by APT to display a message saying that the simulation is done.

After the SpiNNaker indicates that the simulation has completed, UDP packets are sent from the host PC using MATLAB that tell the SpiNNaker which information is requested. For example, in order to determine how many times neuron number 1 has spiked during the simulation, MATLAB sends a UDP packet to the chip/core location on the board that has that information, and then that core responds back with the number of times neuron 1 spiked.

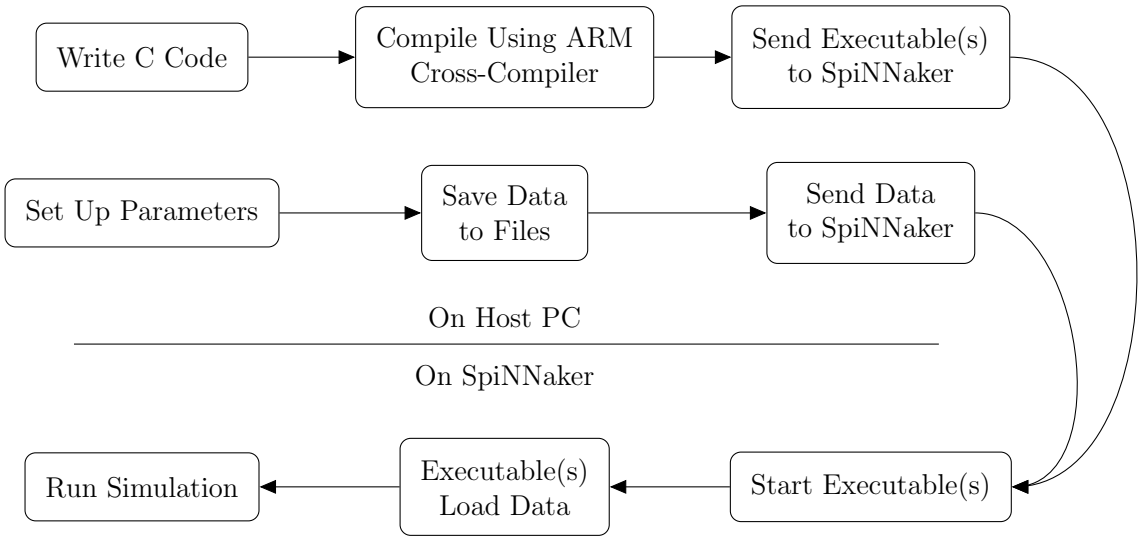


Figure 3.3: Overview of the SpiNNaker flow.

These data are aggregated on the PC in MATLAB, and the results are saved. In fact, MATLAB is used a lot on the host PC for controlling and communicating

with the SpiNNaker. Although the SpiNNaker itself is programmed in C, the original Bayesian networks are all currently described in text files, and MATLAB is used to read in those files and automatically convert those networks into valid networks of neurons for use with the Neural sampling algorithm. MATLAB code is used to calculate the firing probabilities for all the neurons as well as for the colorization of the graph (see Section 3.1.2).

MATLAB is even used to call bash scripts that compile the C code for the application as well as send data to the SpiNNaker and get everything up and running. In effect, MATLAB provides the complete user-facing interface with the board by controlling many of the underlying technologies so everything can be run automatically and directly from MATLAB. However, when tweaks are made to the algorithm on the SpiNNaker or data are sent to different SDRAM locations, the C code or bash scripts are modified. But to run simulations and change basic parameters the only changes are made in MATLAB, so complete experiments can be run in MATLAB throughout the entire process.

Figure 3.3 shows the overview for creating a software system that works on the SpiNNaker. The top portion of the diagram contains tasks that are done on the host PC, and the bottom part describes things that occur on the SpiNNaker itself. All the steps described in this section are completed once the algorithm has already been detailed. Of course, many of the steps are not independent of each other. Instead, many things are done in parallel during the development and testing process.

For example, the C code might be developed at the same time the parameters are established because doing both of those things simultaneously makes testing more feasible.

Chapter 4

Sampling Results on 4-Chip SpiNNaker

This chapter describes the results of running Neural sampling on the smaller 4-chip SpiNNaker. Multiple small networks were created and tested on the board to show the basic functionality, and then larger networks were run in order to determine the performance of the SpiNNaker as compared to other implementations. In addition, results from running Gibbs sampling on the PC in MATLAB and running Gibbs sampling on the 4-chip SpiNNaker are presented.

Some of the work in this chapter has been previously published[55,56] by the author of this thesis.

4.1 Chest Clinic Network

The classic Chest Clinic Network is a small example Bayesian network used to illustrate the basic functionality of some inference algorithms. The papers initially describing Neural sampling[51,52] show some experimental results for Neural sampling on this network, so this work shows that similar results are achieved by using the SpiNNaker implementation. The network was assumed to be trained already and the CPD tables were matched to the ones expressed in the Neural sampling paper[51] describing that network.

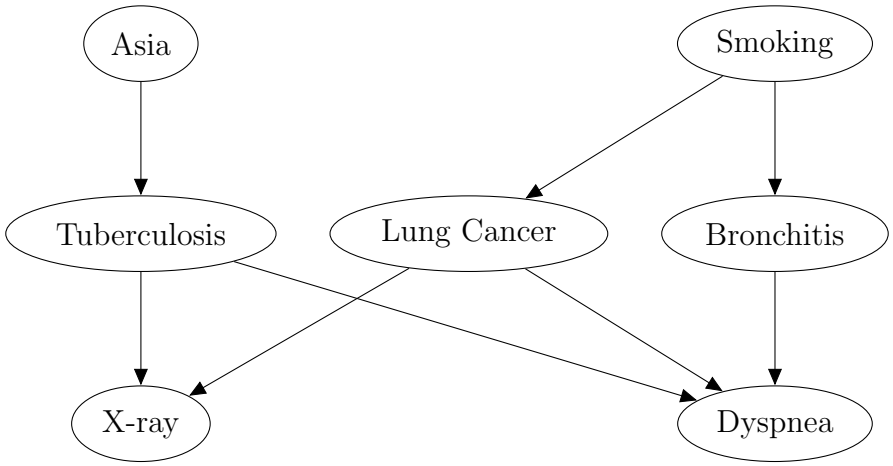


Figure 4.1: The Chest Clinic Network.

The network is shown in Figure 4.1. This network describes the probability of having a few different illnesses given symptoms and other factors. The illnesses are tuberculosis, lung cancer, and bronchitis, and some conditions that relate to the probability of having any of these conditions are whether the person visited Asia, is a smoker, has a positive X-ray, and/or is experiencing dyspnea. The network structure

also suggests that the three illnesses are caused by visiting Asia and smoking. Then, the onset of these illnesses in turn cause X-rays to become positive or dyspnea to occur.

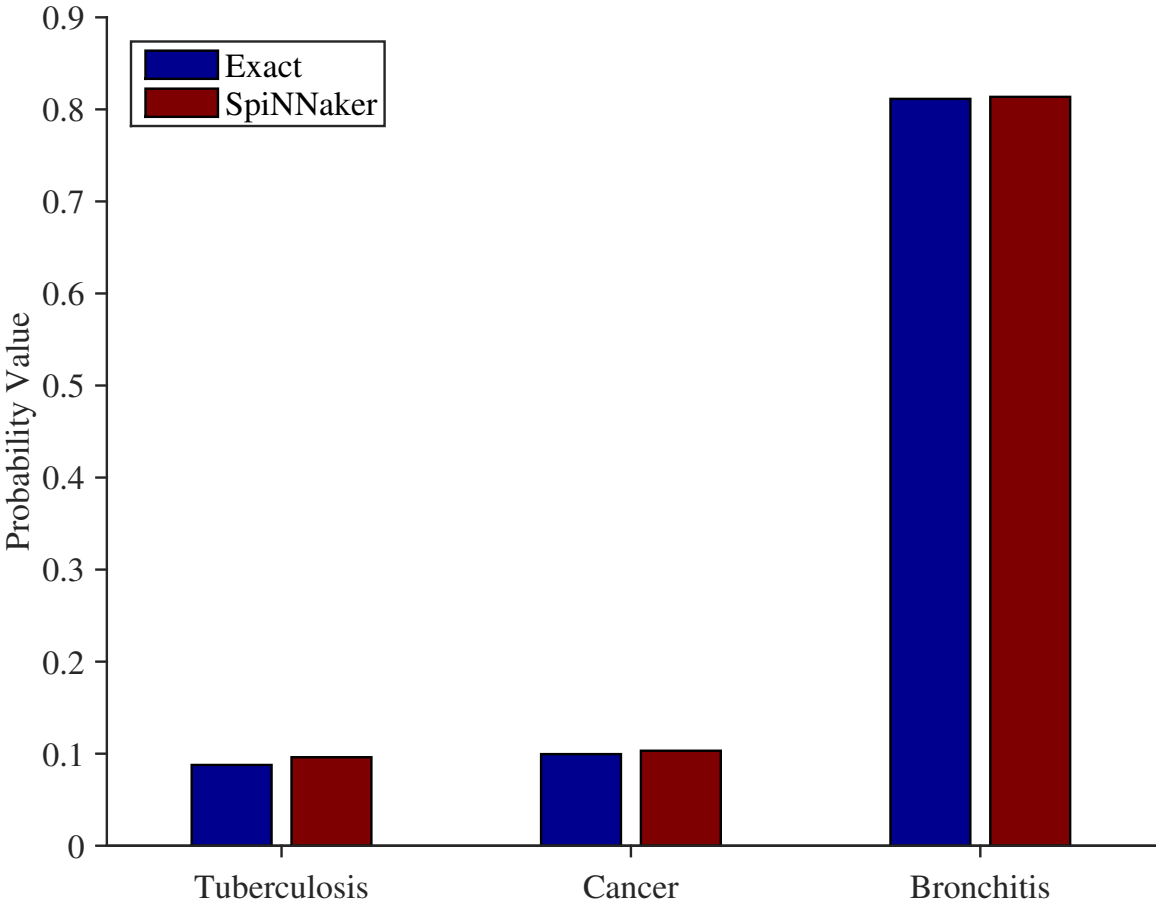


Figure 4.2: Conditional probability values for tuberculosis, cancer, and bronchitis without having an X-ray given that the patient is experiencing dyspnea and has recently visited Asia.

There are two examples described below. One is the case when no X-ray diagnosis has been made, and the second case is when the X-ray is a positive X-ray. When the X-ray has not been taken it is most likely the case that the person simply has bronchitis rather than a more serious illness. On the other hand, once a positive

X-ray has occurred the person is more likely to have a serious condition.

In both of these examples the known evidence is that the person is experiencing dyspnea and has recently visited Asia. For the no X-ray situation exact inference was run in MATLAB and compared to the results of running Neural sampling on the same network using the SpiNNaker. These results can be seen in Figure 4.2. The exact inference results are the blue bars and the Neural sampling results are the red bars. For all the examples in this section Neural sampling was run for 50,000 iterations.

As mentioned earlier, each of these results is a conditional probability value. For example, the values for Tuberculosis represent $\mathrm{P}(T = 1|A = 1, D = 1)$. The other values are similar except that they are substituted in for the first parameter, so Cancer is $\mathrm{P}(C = 1|A = 1, D = 1)$.

In the second example the X-ray has a positive result in addition to the evidence already present in the previous example. The positive X-ray indicates that a more serious illness than bronchitis may be present, so there should be an increased probability of tuberculosis or cancer. These inference results are shown in Figure 4.3.

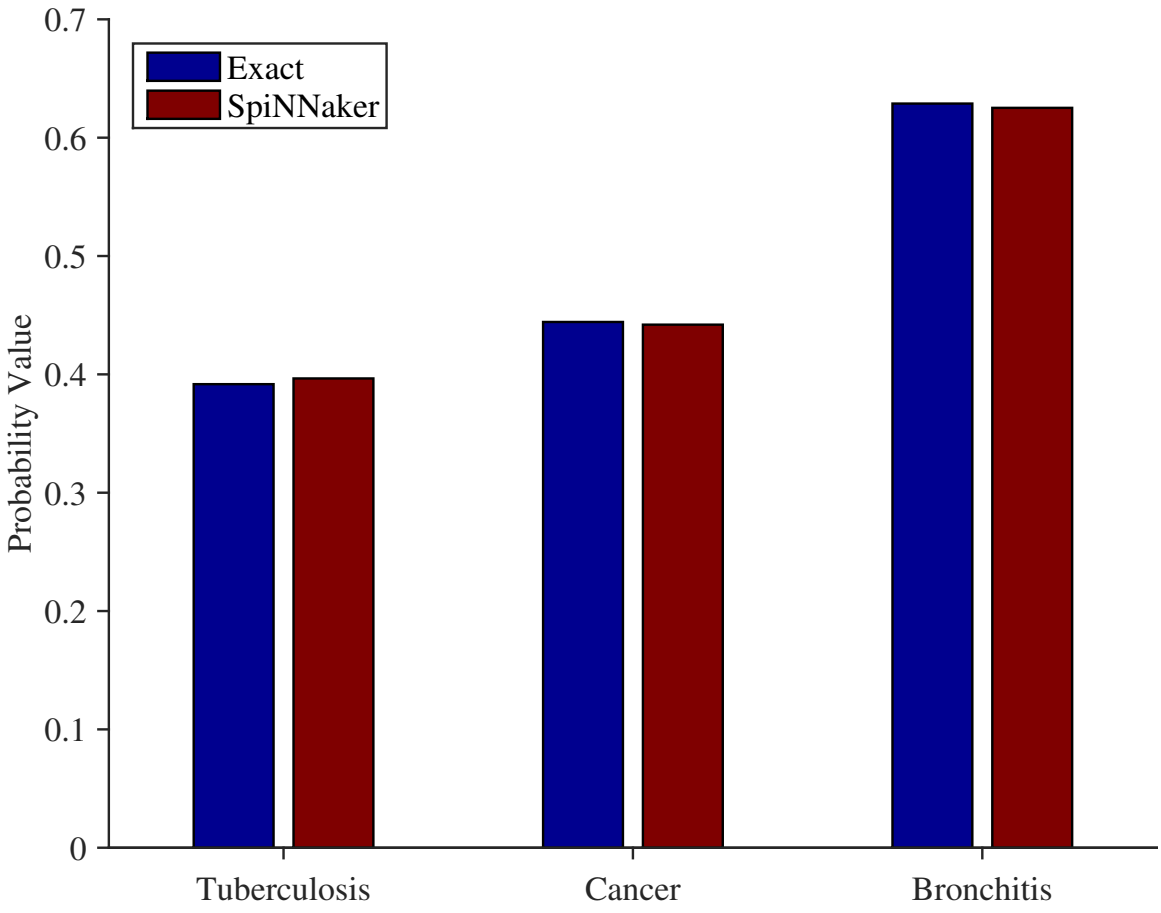


Figure 4.3: Conditional probability values for tuberculosis, cancer, and bronchitis given that the patient visited Asia, has dyspnea, and has had a positive X-ray. Note that the probability of more serious illness has gone up compared to Figure 4.2.

4.2 Icy Road Network

The Icy Road Network was described in detail back in Chapter 2 and shown in Figure 2.1. In this section inference results using both exact sampling in MATLAB and Neural sampling with 50,000 iterations on the SpiNNaker are presented.

Two situations are considered here. In the first situation the road is known to be icy and the inference task is to determine the probability of the ground being below

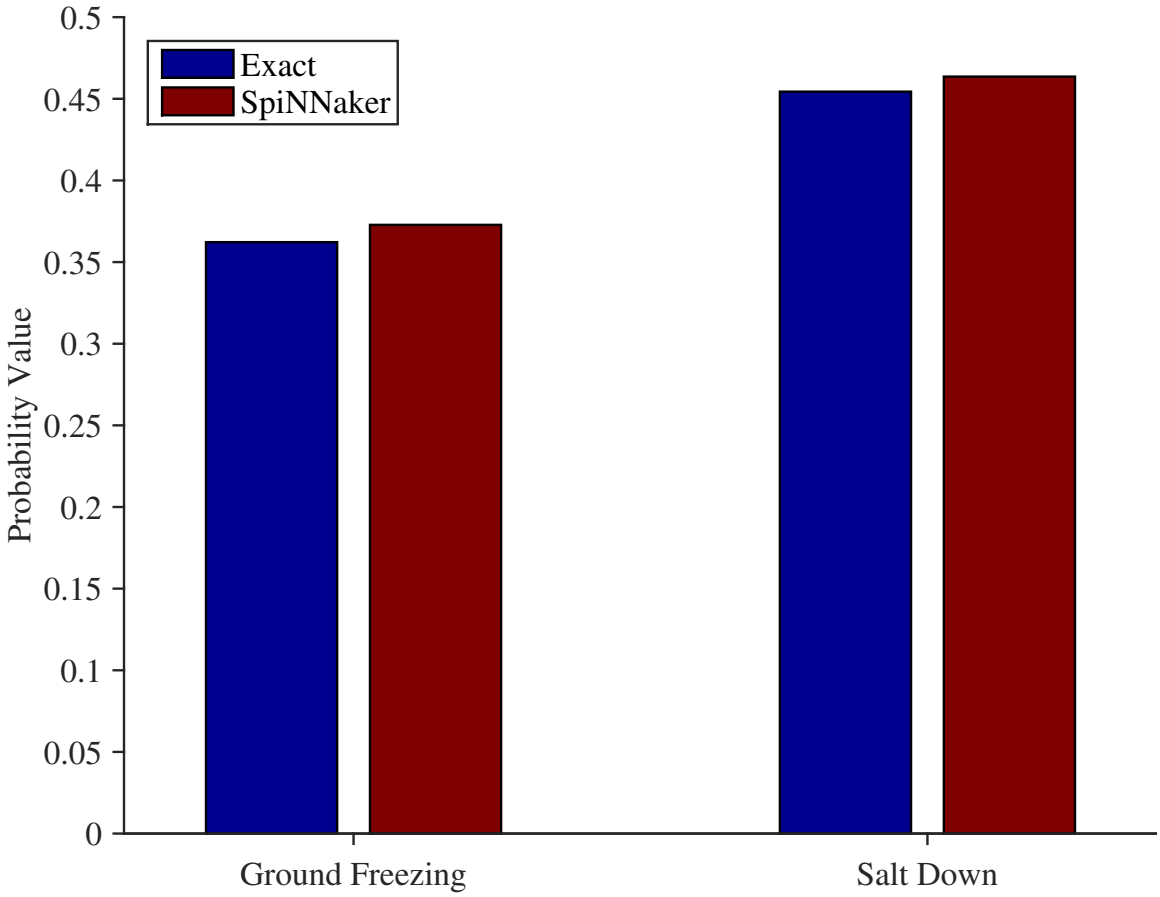


Figure 4.4: Separate conditional probability values for the ground being below freezing and salt being on the road given that the road is icy.

freezing given that the road is icy as well as the probability that salt was placed on the road given that the road is icy. These results are shown in Figure 4.4.

The second situation is that the road is icy and there is no significant amount of precipitation. These results are depicted in Figure 4.5.

In both Figure 4.4 and Figure 4.5, the road is icy, and the difference between the two is that the latter describes the situation where there is also no significant precipitation. The results are intuitive. If the road is icy there is a certain probability

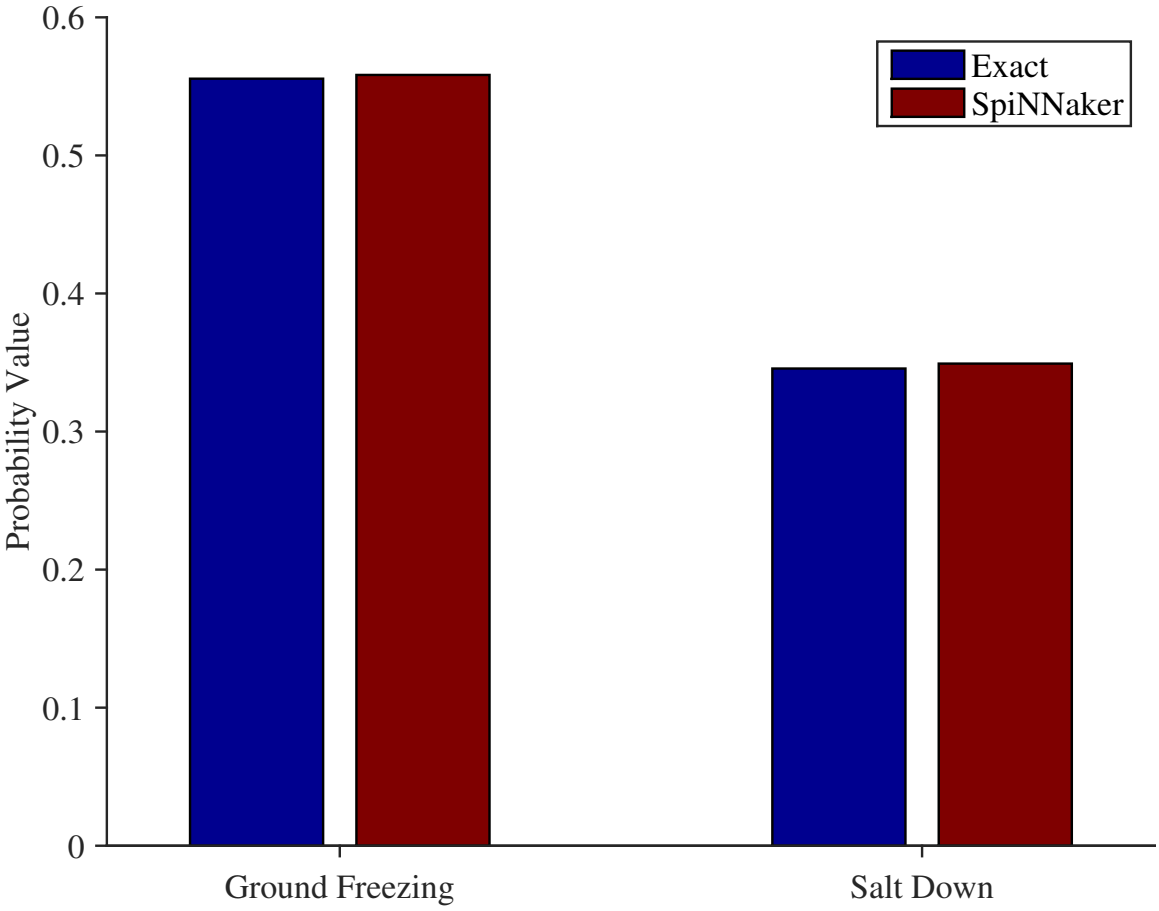


Figure 4.5: Separate conditional probability values for the ground being below freezing and salt being on the road given that the road is icy. This is the same plot as Figure 4.4 except that the extra condition of having no significant precipitation is added.

of the ground being below freezing. However, if evidence is added that says there is no significant precipitation, then it is more likely that the ground must be below freezing because otherwise it is unlikely that the road would be icy.

On the other hand, the reasoning behind the probability of salt being down is more subtle. When evidence of no significant precipitation is added in, that also means that there is a higher probability of no precipitation at all, which leads to a lower probability of salt being down because it would be less useful.

The main point of this section, though, is that Neural sampling on the SpiNNaker again faithfully reproduces the results achieved by performing exact inference on these basic networks.

4.3 Larger Networks and Scalability

Larger networks were simulated in order to determine how well the Neural sampling architecture on the SpiNNaker scales to bigger problems (networks). The goal was to determine how many nodes can fit onto the four-chip board as well as to see how well the speed of inference scales as the number of nodes increases.

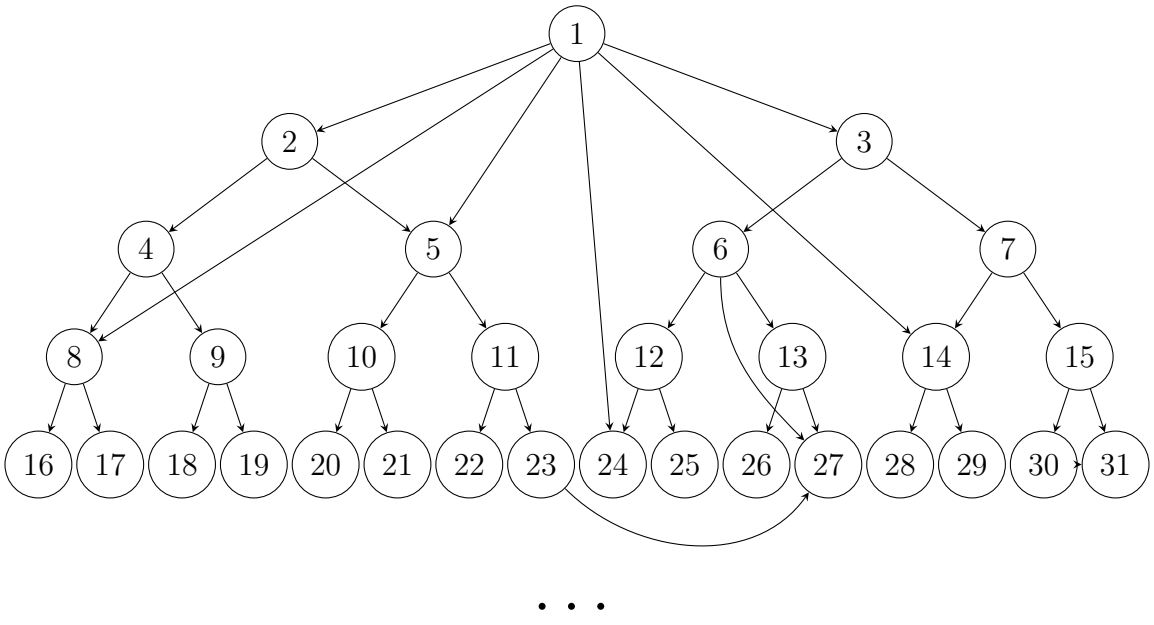


Figure 4.6: Basic Large Network Structure.

These networks were constructed in a way that makes it is easy to add more nodes

to the base structure, but this does not restrict the topology of Bayesian networks that can be simulated using this technique. This structure was mostly chosen for its simplicity and ease of creation, but other network structures could have been used for these experiments as well.

The network is basically a binary tree of binary nodes where the arrows in the network all point from the top down to the bottom. There are a few extra arrows added in so that the network is not completely regular (see Figure 4.6). These networks will be referred to as networks with k layers in this thesis, and networks of k layers contain $2^k - 1$ total nodes.

In order to see how faithfully the SpiNNaker simulation performs the Neural sampling algorithm, a comparison was done between the MATLAB PC implementation and the SpiNNaker version. Figure 4.7 shows the results which are presented as a mean absolute difference between the inferred probability values of the nodes in the network on each architecture.

First, the network was given identical evidence on each platform, and then inference was run on all the other nodes using the two different hardware/software architectures. The number of levels in the network was varied to see how well the PC and the SpiNNaker simulations match for various network sizes. The number of layers here was varied from 6 layers up to 15 layers ($2^6 - 1 = 63$ nodes up to $2^{15} - 1 = 32,767$ nodes).

Each simulation was run for 50,000 iterations. As the number of nodes in the

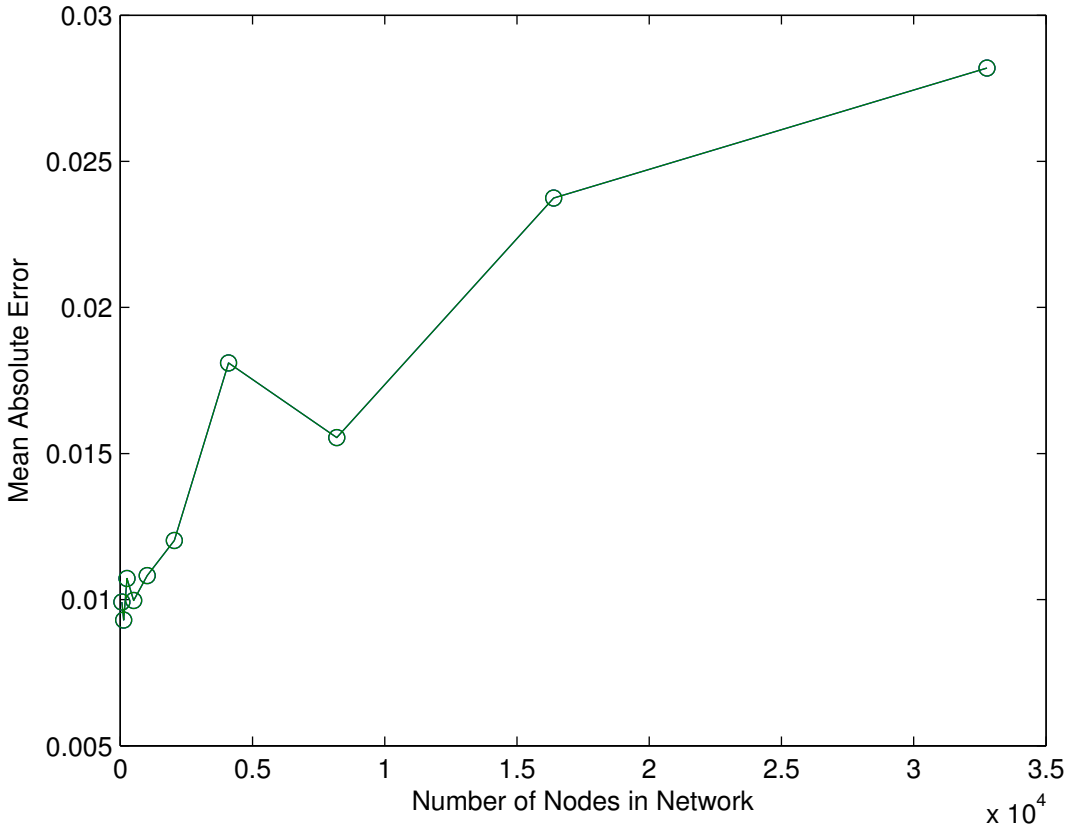


Figure 4.7: Mean Absolute Error Values between PC and SpiNNaker for binary tree-structured Bayesian networks.

network increases, the results on the two different architectures diverge more. This is expected because as the number of nodes increases the more complicated the probability distribution of the model becomes. Since samples are aggregated over time and the accuracy of MCMC techniques depends on adequately exploring the relevant areas of the probability distribution, having a larger network and not running sampling for more iterations leads to a decrease in accuracy. However, the mean absolute error values suggest that the SpiNNaker implementation reasonably matches that of

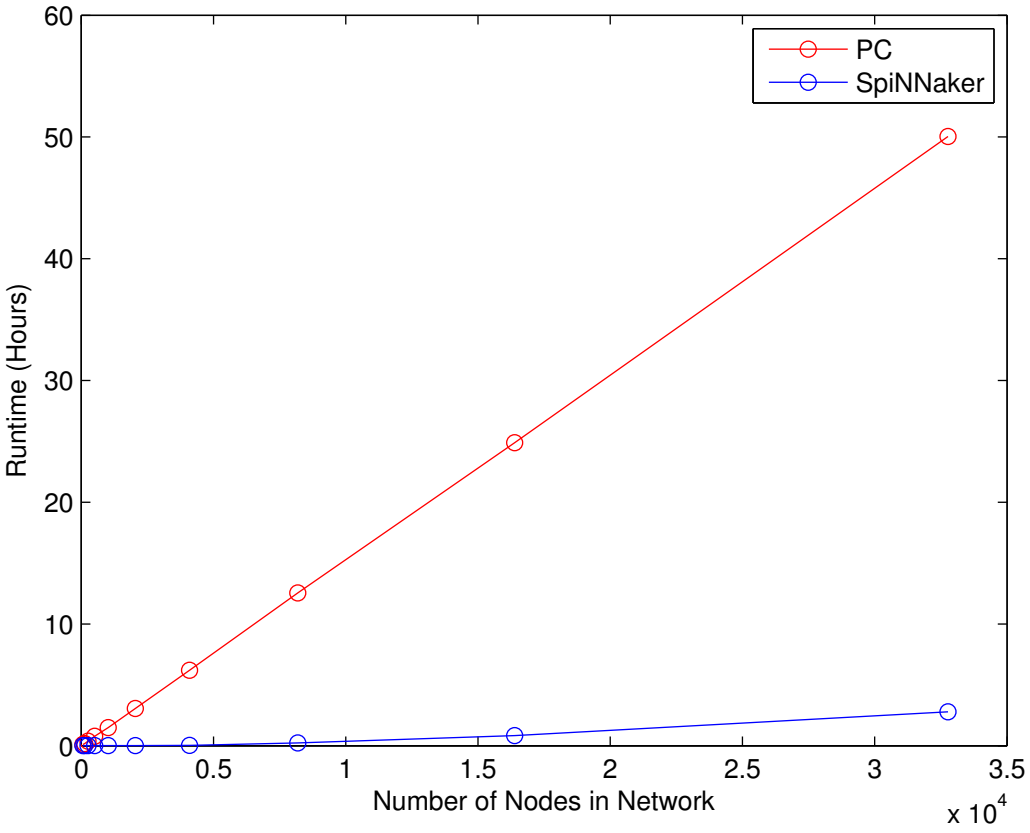


Figure 4.8: Runtimes for Neural sampling in hours depending on the number of nodes in the network using the 4-chip SpiNNaker architecture.

the PC using MATLAB.

A comparison between the SpiNNaker and the PC implementation was also performed to see how well each of them scale in terms of speed as the number of nodes increases (See Figure 4.8). The network sizes were the same as those used for the accuracy comparison ($2^6 - 1 = 63$ nodes up to $2^{15} - 1 = 32,767$ nodes).

It is not surprising that the PC simulation has a mostly linear slowdown as the number of nodes increases. The amount of work done per node in MATLAB is

essentially fixed as the number of nodes increases because there is no network communication and all the nodes are simulated sequentially, so the expected slowdown is approximately linear. On the other hand, the SpiNNaker implementation does slow down at a higher than linear rate as the number of nodes grows large. This reduction in speed is likely due to the fact that the communication network on the SpiNNaker has to deal with a much greater number of messages that are passed between nodes as the network size increases.

However, it is clear that parallelizing the sampling algorithm yields very significant speedups in execution time, even when only considering the 4-chip SpiNNaker. Expanding to the larger SpiNNaker board yields even greater improvements and will be described later in this thesis.

4.4 Comparison to Gibbs Sampling

Gibbs sampling was coded in MATLAB in order to better test the accuracy of the Neural sampling algorithm running on the SpiNNaker. In addition, results using Gibbs sampling from Kevin Murphy's Bayes Net Toolbox[53] (BNT) were compared to both of these implementations developed for this thesis.

The comparisons were done on the binary tree-like networks described in Figure 4.6. For a given network size the three implementations were run for various amounts of sampling iterations. The results for a network of 127 nodes are shown

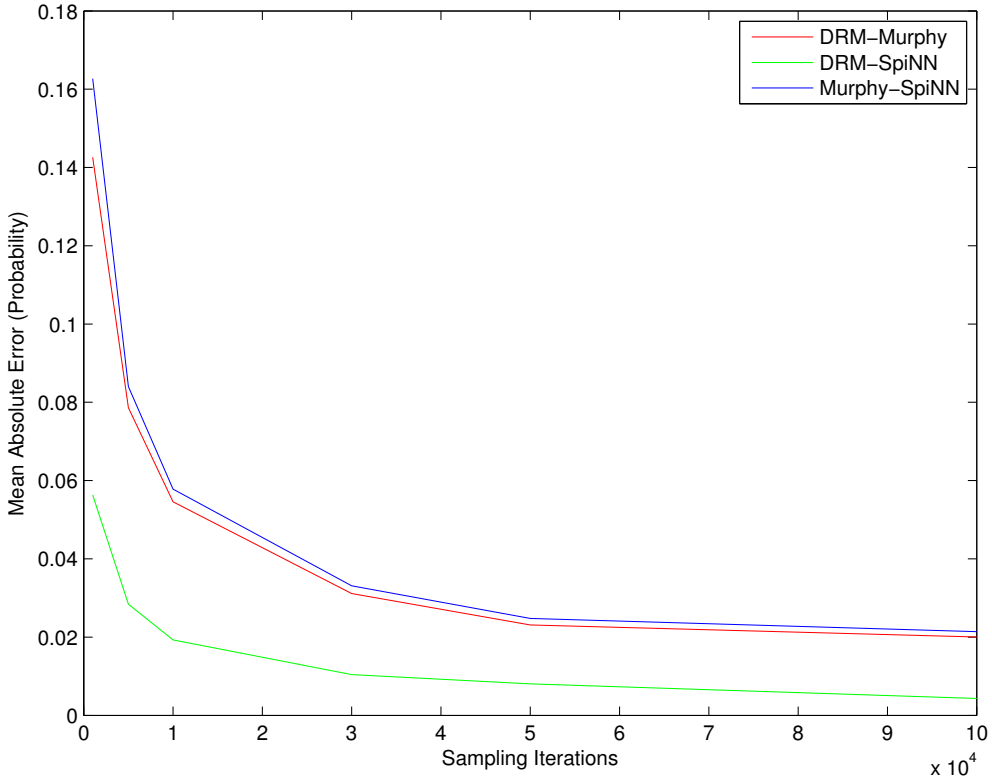


Figure 4.9: Mean Absolute Error (MAE) values between different sampling algorithms for a binary tree-like network with 7 layers (127 nodes) as a function of the number of sampling iterations. DRM stands for the author's implementation of Gibbs sampling. Murphy stands for the implementation of Gibbs sampling found in Kevin Murphy's BNT Toolbox. SpiNN stands for the Neural sampling algorithm running on the SpiNNaker.

in Figure 4.9. As expected, the absolute error values decrease as the number of iterations increases. It is unclear why Kevin Murphy's BNT implementation does not match the other two implementations as closely as they match each other, but all the results show a similar trend in the reduction of their differences as the number of sampling iterations grows.

The runtimes of all three algorithms were also recorded from these preliminary

results. These speeds are shown in Figure 4.10. This comparison was run on a small binary tree-like network of 1023 nodes (10 layers), and all the algorithms run on a PC were run using an Intel Core-i7 (Sandy Bridge) quad-core laptop running Ubuntu. DRM Gibbs sampling is a MATLAB implementation of Gibbs sampling so it is the slowest out of the three implementations. Kevin Murphy's BNT Gibbs sampling implementation is written in C as a MATLAB MEX function so it is faster than the DRM MATLAB implementation. Finally, Neural sampling on the SpiNNaker is the fastest due to its parallel architecture. Each of the runtime lines increase linearly as the number of sampling iterations increases because that is a linear increase in the runtime complexity.

The 4-chip SpiNNaker architecture runs Neural sampling over 250 times faster than Gibbs sampling running in MATLAB and about 25 times faster than the same algorithm implemented in C. The main tradeoff here is that the algorithms on the PC are running at a very high clock rate but are sequentially coded in a single thread whereas the SpiNNaker implementation only runs on ARM processors running at 200 MHz. However, the 4-chip SpiNNaker these experiments used consists of 64 of these ARM cores that were used in parallel for computations.

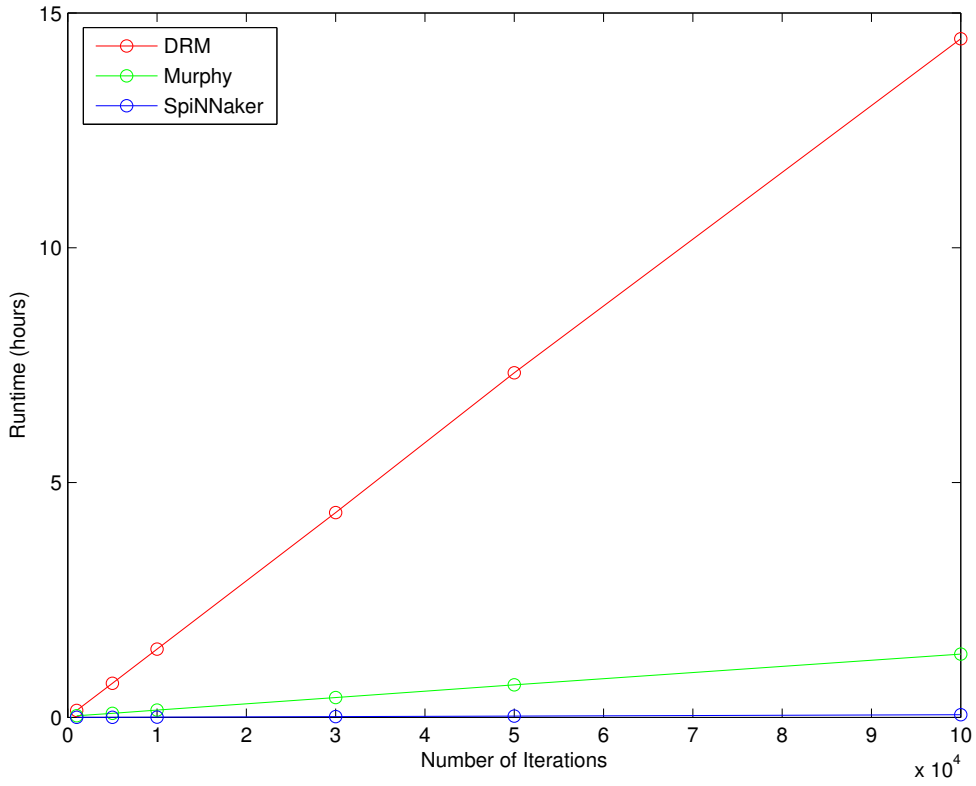


Figure 4.10: Runtimes in hours for all three algorithms with a binary tree-like network of 1023 nodes as the number of sampling iterations increases. DRM and Murphy are Gibbs sampling implementations, and SpiNNaker is Neural Sampling on the SpiNNaker.

4.5 Discrete Gibbs Sampling on 4-Chip SpiNNaker

This section describes the development of parallel Gibbs sampling in discrete Bayesian networks with categorical distributions on the 4-chip SpiNNaker as well as the sequential, single-threaded version implemented in MATLAB. So the implemen-

tation of this particular type of Gibbs sampling is very similar to Neural sampling except that each node can take on multiple values of a categorical distribution rather than being restricted to only having two possible values.

The organization of the code for Gibbs sampling on the SpiNNaker is very similar to that of the Neural sampling design described in Section 3. The same framework is used to load the network from text files and load the data on the SpiNNaker except for minor tweaks to accommodate more values for each random variable in the model. Of course the data are set up slightly differently because the tables containing the sampling distributions depending on the nodes' Markov Blankets are larger due to the fact that the nodes can take on more than two values. The sampling code accounts for this same difference and so does the code that aggregates all the samples at the end to compute the probability of the node taking on its values.

Another big difference in the code is that unlike Neural sampling there is no log ratio and no sigmoid computation done to determine the sampling distribution at each node during each iteration (see Equation 2.40 and Equation 2.41). These differences are due to the fact that the nodes do not have membrane potentials and refractory periods that affect the cadence of spikes coming from each node. For reference the Gibbs sampling distribution is described in Expression 2.29.

The following sections describe example discrete Bayesian networks for which inference was performed on the SpiNNaker in parallel and compared to inference results run on a PC.

4.5.1 Student Network

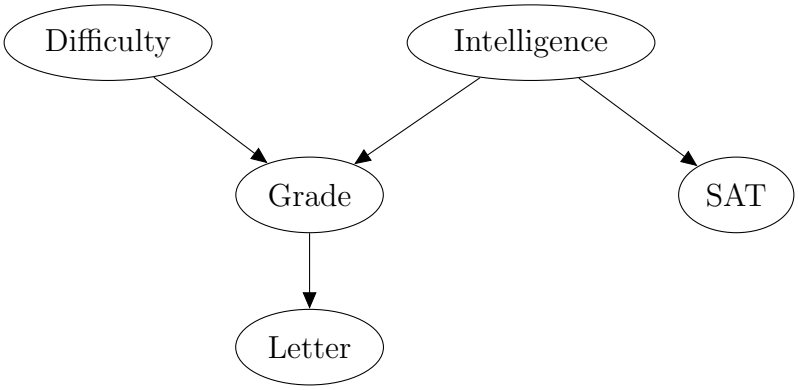


Figure 4.11: Student Network.

One example discrete Bayesian network is shown in Figure 4.11. This basic model describes various factors relating to the performance of a student.[42] Here the grade directly depends on the difficulty of the class and the student's intelligence. The student's SAT score directly depends on his/her intelligence, and the likelihood of getting a positive recommendation letter from the teacher depends directly on the grade earned in the class. Each node is binary except for the Grade node, since the grade can be A, B, or C. The Difficulty can be easy or hard, the intelligence can be low or high, the SAT score can be low or high, and the Letter can be weak or strong.

The first test was to focus on the PC implementation of Gibbs sampling using MATLAB, and it was compared to exact inference to ensure that it works correctly. These results can be seen in Figure 4.12. In both cases the difficulty of the class was set to be hard, and the recommendation letter was varied from strong to weak. The left side shows the inference results when the recommendation letter is strong. In

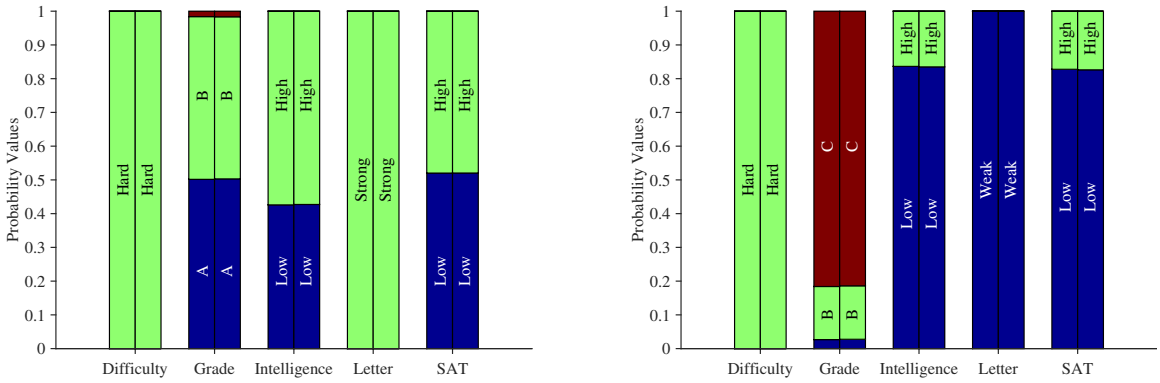


Figure 4.12: Conditional probability values determined from performing inference on the student network where the evidence is that the difficulty is hard and the letter is varied from being strong on the left to weak on the right. The two bars in each column correspond to the results of doing Gibbs sampling for 100,000 iterations and exact inference, respectively.

that situation, the student is likely to get an A or a B in the class and has about a 50% chance of having a high SAT score. In addition, his/her intelligence is more likely to be high instead of low. On the other hand, when the recommendation letter is weak instead of strong, the likelihood of getting an A in the class is inferred to be very low, and the most likely grade is a C. The student's SAT score is likely low, and the student also is more likely to have lower intelligence.

Each inference result has two columns for each node, and they correspond to performing Gibbs sampling for 100,000 iterations on the left and exact inference on the right. Thus, these results provide some evidence that the Gibbs sampling algorithm on the PC works correctly.

Networks were then run in parallel on the 4-chip SpiNNaker and compared to a new run of sequential Gibbs sampling on the PC using MATLAB. See Figure 4.13 for the results. Inference was performed in the same model as earlier with the same

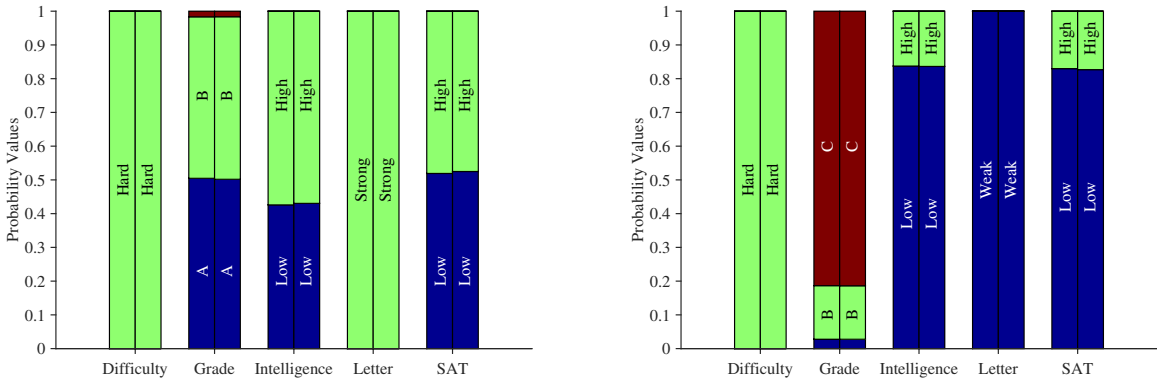


Figure 4.13: Conditional probability values determined from performing inference on the student network where the evidence is that the difficulty is hard and the letter is varied from being strong on the left to weak on the right. The two bars in each column correspond to the results of doing Gibbs sampling for 100,000 iterations on the PC and in parallel on the SpiNNaker, respectively.

fixed evidence, and the SpiNNaker results are very similar to Gibbs sampling on the PC. In addition, since the experiments were similar to those in Figure 4.12, the two graphs achieved on the SpiNNaker can be directly compared to those that only ran on the PC as shown earlier. This means that the parallel Gibbs implementation on the SpiNNaker also matches well with the exact inference results.

4.5.2 ALARM Network

The ALARM (A Logical Alarm Reduction Mechanism) network[58] was used to monitor and diagnose patients based on 16 findings and 13 intermediate variables. There were 8 possible diagnoses in the network, and the model is comprised of 37 discrete variables.

Variables in the network include items such as cardiac output, central venous

pressure, pulmonary capillary wedge pressure, respiratory rate, and left ventricular end-diastolic volume. Some diagnoses include hypovolemia (decreased blood plasma volume), anaphylaxis (severe allergic reaction), and pulmonary embolism.

The network is available online through the bnlearn R package[59] website[60] and was already trained, so code was written in MATLAB to convert the network parameters to the format used in the rest of this thesis, and approximate inference using Gibbs sampling was performed on the network using both MATLAB and the SpiNNaker.

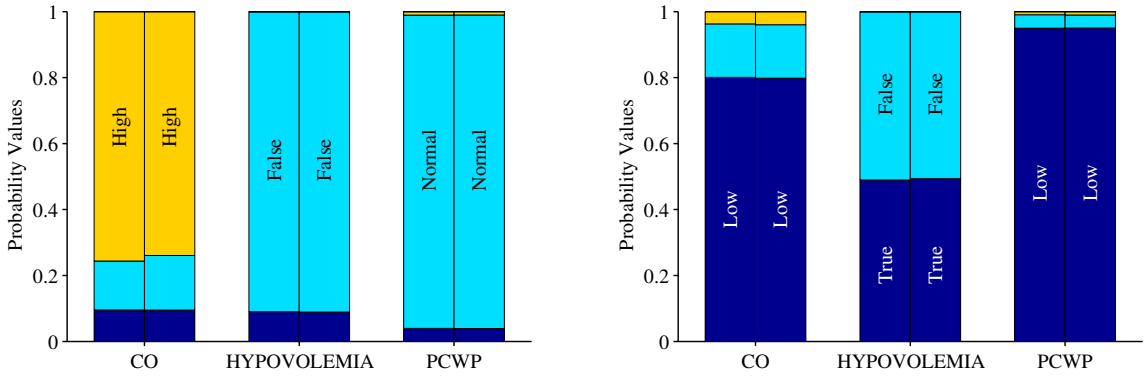


Figure 4.14: Inference results for the ALARM network, a Bayesian network composed of discrete random variables. Two situations were examined: good (left) and bad (right) health. Approximate inference was performed on the SpiNNaker and on a PC using MATLAB. Inference was done for all nodes, but three are highlighted here: CO (cardiac output), HYPOVOLEMIA, and PCWP (pulmonary capillary wedge pressure). The left bar for each variable indicates the inference results from the PC and the right bar corresponds to the SpiNNaker results.

Two experiments were done, both with the same observed nodes. In the first experiment here called "good health," the left ventricular end-diastolic volume (LVED) was fixed to be normal rather than low or high, and the node indicating left ventricular failure was fixed to be false rather than true. In the second experiment called

"bad health," the LVED was fixed to be low and the left ventricular failure was fixed to be true.

Inference results can be seen in Figure 4.14. Inference was performed on all variables in the network, but the figure focuses on certain variables for clarity. For the good health case the cardiac output was most likely to be high whereas in the bad health case the output was most likely to be low. Hypovolemia was likely to be false in the good health situation and equally likely to be true or false in the bad health situation. Finally, the pulmonary capillary wedge pressure was most likely to be normal in the good health situation and low in the bad health case. The results from running the MATLAB code on the PC are very similar to those running directly on the SpiNNaker.

4.5.3 Child Network

The Child network (Figure 4.15) was created during a project involving the Great Ormond Street Hospital for Sick Children to diagnose congenital heart disease in newborns during the first few days after birth.[61] This network consists of 20 nodes describing various measurements such as CO_2 levels, lung blood flow, and chest X-ray results.

The model is available online[60] with all parameters trained, so the same code described in Section 4.5.2 was used to convert the network for use with the SpiNNaker framework developed here. Evidence was chosen to be fixed, and approximate

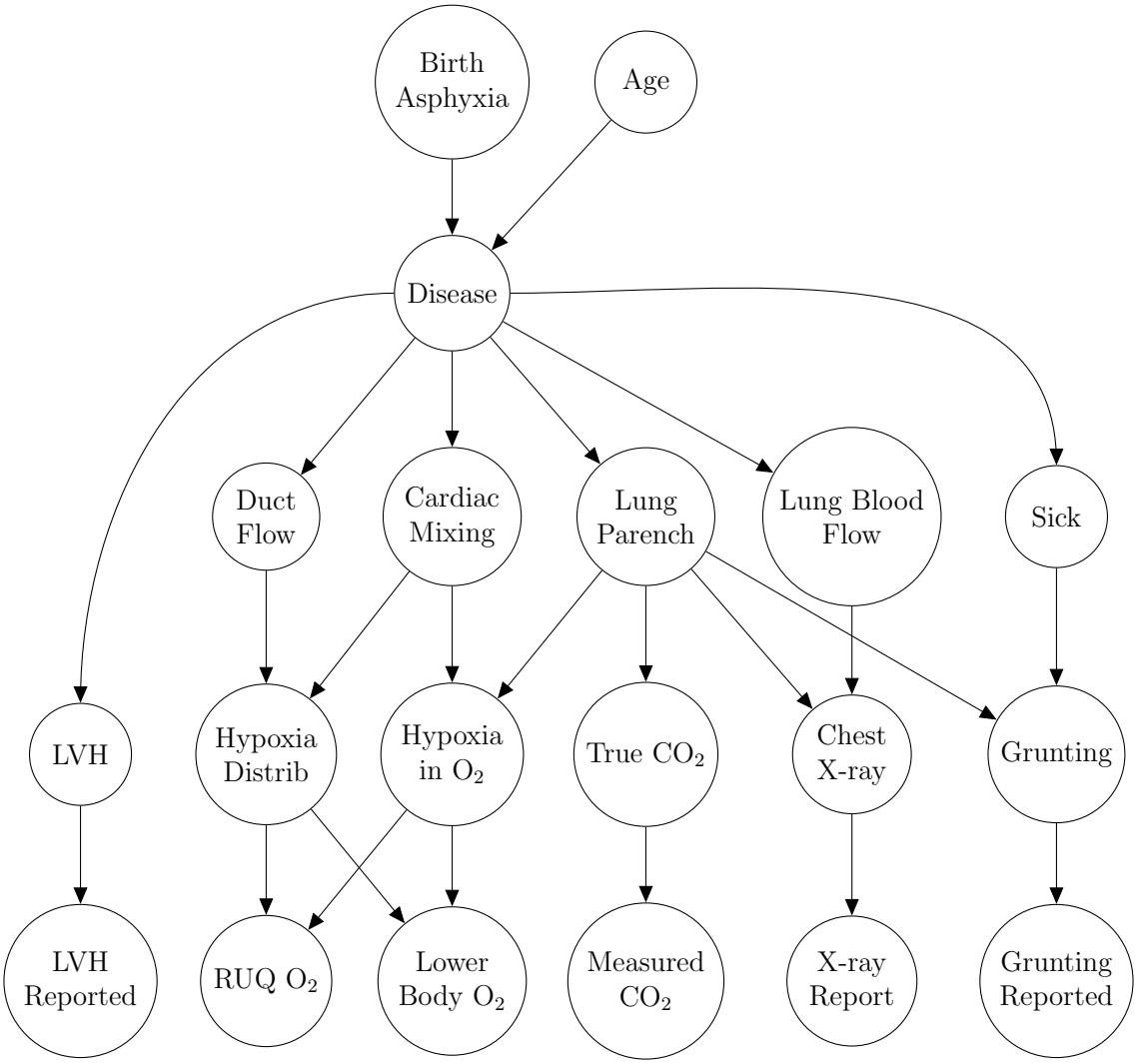


Figure 4.15: The Child Network was used to diagnose congenital heart disease by taking into account various random variables such as X-rays, CO_2 measurements, etc.

inference using Gibbs sampling was performed on both the PC and the SpiNNaker.

Two situations were examined, one here called "good health" and the other here called "bad health." The good health situation included two pieces of evidence: the lung flow was high and the baby was not grunting. For the bad health case the lung flow was low and the baby was grunting.

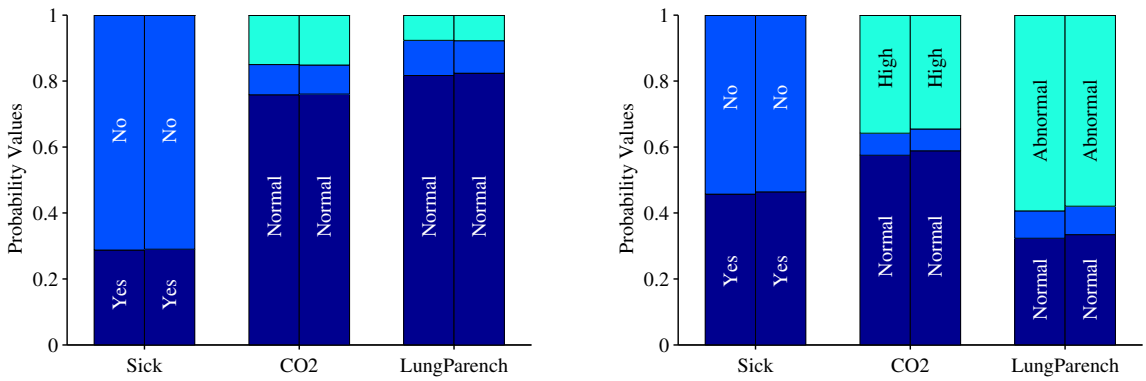


Figure 4.16: Inference results for the child network. Two situations were examined: good (left) and bad (right) health. Inference was done over all nodes, but three nodes are shown here: whether the child is sick, CO_2 levels, and the status of the lung parenchyma. The left bar for each variable indicates the inference results from the PC and the right bar corresponds to the parallel SpiNNaker implementation results.

The inference results are shown in Figure 4.16. Inference was done on all nodes in the network, but a few specific variables are highlighted in the figure. The infant was more likely to be sick when it was in the bad health experiment and more likely to have high CO_2 levels. Finally, the lung parenchyma (functional tissue) condition was more likely to be abnormal in the bad health case. The parallel inference on SpiNNaker results are again similar to those achieved using the MATLAB implementation on the PC.

Chapter 5

Migrating to 48-Chip SpiNNaker and the Parallella

The inference results shown in Chapter 4 were all run on the small 4-chip SpiNNaker for which 64 ARM cores were used to perform approximate MCMC inference in parallel. Neural sampling was ported to the larger 48-chip SpiNNaker as described in this section for a big performance increase due to the ability to use 768 cores for parallel sampling.

This chapter also details the Parallella, another parallel computing device, in Section 5.2. Neural sampling was ported to the Parallella and its performance was compared to both SpiNNaker devices. Finally, a heterogeneous architecture was created where the SpiNNaker and Parallella work together to perform Neural sampling which allows for the tradeoffs between the two platforms to be explored together.

Some of the work in this chapter has been previously published[55,56] by the author of this thesis.

5.1 Migration to 48-Chip SpiNNaker

Each core on the larger SpiNNaker board is set up basically the same as each core on the smaller board. Therefore, most of the required changes are related to the fact that there are more locations on the board where nodes are physically located. As a result the code to send the data to the board was changed as well as the code laying out the nodes and determining the routes that packets take when the nodes communicate with one another. Destination-based routing can still be used because there are 768 cores used in this situation and there are 1024 possible router entries per core, so there is plenty of room to store the route for each destination on the board. Each core's router is configured to know where to send packets destined for particular endpoints.

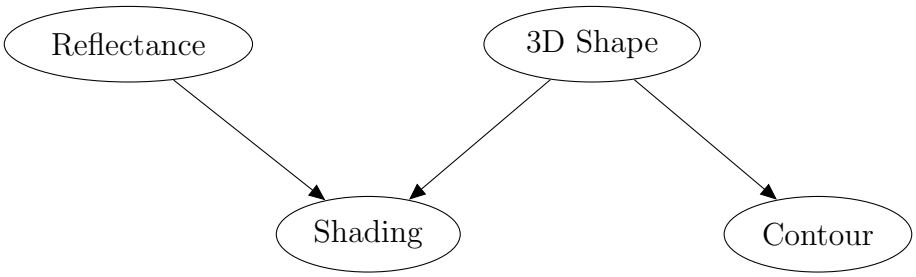


Figure 5.1: Binary graphical model for visual perception of two side by side shapes. The reflectance and 3D shape of the objects influence the perceived shading and contour of the objects when processed by the human visual system.

In order to perform a simple test of the migration to the larger SpiNNaker board, an experiment was reproduced from one of the original Neural sampling papers.[51] In this visual experiment[62] two objects were touching each other side by side. These objects could either both be cylindrical or they could both be flat. They could both have identical reflectance values or one value could be larger than the other but the object was still flat horizontally across the surface. The combination of the shapes of the objects and the reflectance values influence human perception of the objects' shading as well as the contours of the objects. See Figure 5.1 for the binary graphical model describing this model.

Let R denote the random variable describing the reflectance, S denote the shading, C denote the contour, and D denote the 3D shape of the objects. In this experiment the subjects know the type of shading (S) and contour (C), and the goal is to infer the reflectance (R) and 3D shape (D) of the objects. As can be seen from the hierarchical structure shown in Figure 5.1, the joint distribution can be factored in the following manner: $\mathrm{P}(R, D, S, C) = \mathrm{P}(R)\mathrm{P}(D)\mathrm{P}(S|R, D)\mathrm{P}(C|D)$.

Inference was performed in two different situations given the trained parameters detailed in the Neural sampling paper.[51] The shading and contour were both known. In both cases the shading was fixed while the perceived contour was chosen to be flat for one situation and cylindrical for the other.

Inference on two different conditional probability distributions was achieved. For case 1 inference calculates $\mathrm{P}(R = \text{different}|S = \text{linear}, C = \text{cylindrical})$ while for case

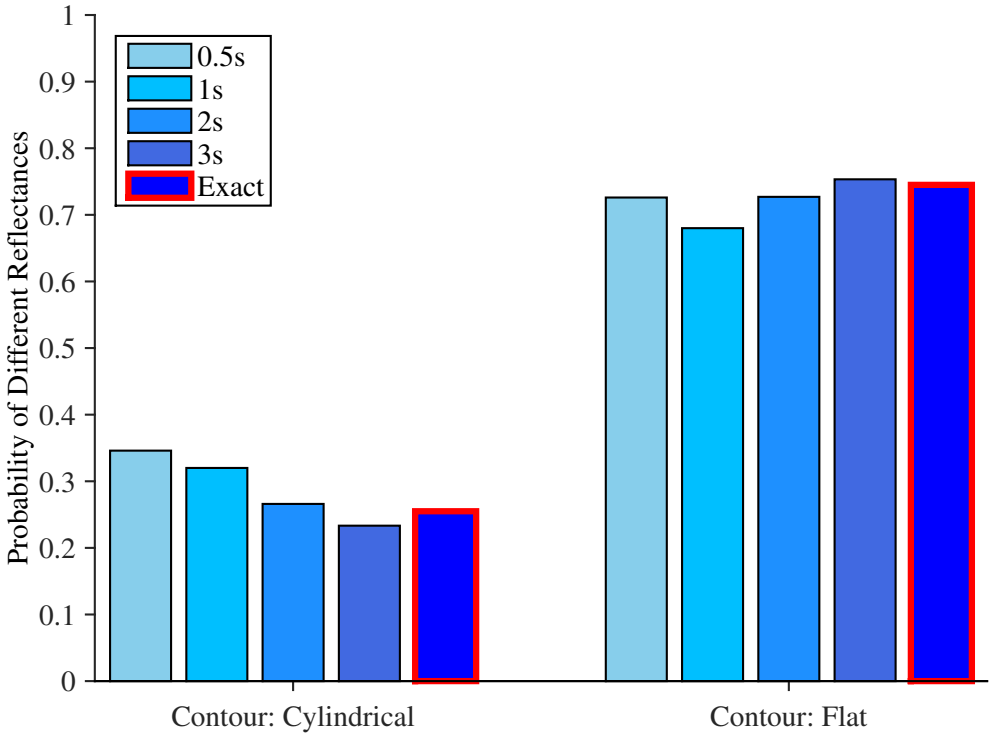


Figure 5.2: Inference (Neural sampling) in visual perception experiment on SpiNNaker hardware. The evidence for both sections includes linear shading, and on the left the perceived contour was cylindrical while on the right the perceived contour was flat. Neural sampling was run four times for each example (cylindrical vs. flat), and the amount of time the algorithm was run varied from 0.5 seconds up to 3 seconds.

2 inference determines $\mathrm{P}(R = \text{different} | S = \text{linear}, C = \text{flat})$.

This model is simple enough that exact inference is trivial, so the ground truth is exact inference. In addition, the model was run on the SpiNNaker so that approximate inference was performed for each of those two cases. The results are shown in Figure 5.2, which compares exact inference with Neural sampling running in MATLAB on a PC and Neural Sampling running on the SpiNNaker. The networks of spiking neurons were run on the SpiNNaker for varying amounts of time, and the

longer the networks were run the better they converged to the ground truth.

Binary tree-like networks of the type described in Figure 4.6 were also run on the large SpiNNaker of sizes up to 18 layers (262,143 nodes) so that scalability of the system could be explored. The larger SpiNNaker has 12 times more cores available for performing parallel sampling than the smaller SpiNNaker has and therefore the 48-chip SpiNNaker can perform sampling much faster than the smaller board can. These results are reported in the next few sections along with those from related architectures, but the first results can be found in Figure 5.4 along with those of the Parallella which is introduced next. All the runtime comparisons performed are using 50,000 iterations of neural sampling unless otherwise stated.

5.2 Parallella

The open-source Parallella[26] board combines two ARM A9 cores and an FPGA inside the Xilinx Zynq7000 series system-on-chip[27] as well as 16 floating-point processing units packaged as the Adapteva Epiphany coprocessor.[28] Each floating-point unit has a 1 GHz clock speed as compared to the approximately 200 MHz clock speed for the ARM cores in the SpiNNaker, but the Parallella has less cores available. See Figure 5.3 for an image of one of the two Parallella boards used in this thesis. The board is contained within the original acrylic case provided by Parallella. The two wires at the top of the image are used to power the fan that is currently located

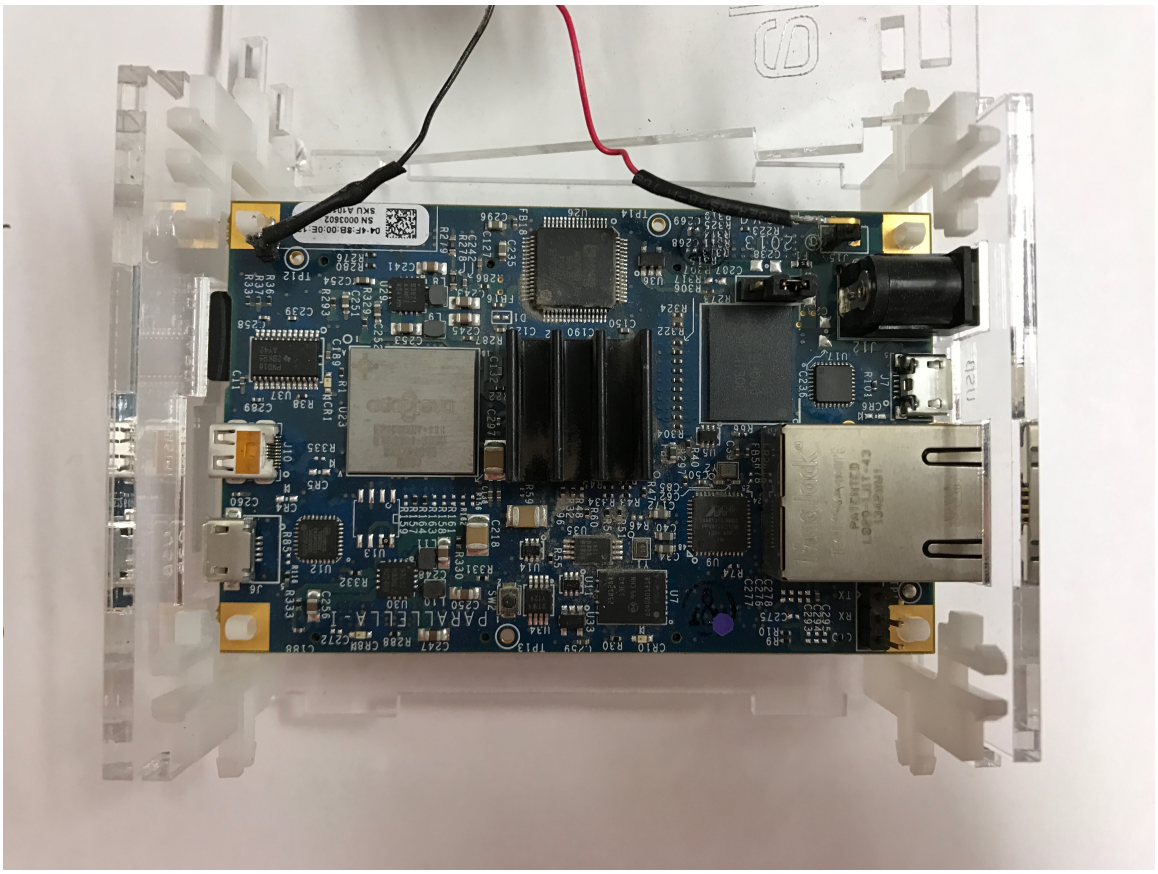


Figure 5.3: Parallella.

on the top of the box. The silver chip toward the left of the board is the Epiphany coprocessor, and the Zynq chip is located underneath the black heatsink in the center.

Both the ARM chips and the Epiphany can be programmed using OpenCL.[63,64] OpenCL allows people to create kernels which execute in parallel on various hardware platforms (GPUs, CPUs, some FPGAs, and some custom hardware such as the Parallella). In addition, the ARM cores run Ubuntu off an SD card and can be programmed in various ways including using C/C++ which was the main technique used for this thesis. The floating-point units on the Epiphany coprocessor can also

be programmed using C++ using the Epiphany Software Development Kit (ESDK) developed specifically for this hardware.

Neural sampling was implemented on the Parallella. The same MATLAB framework that was developed for the SpiNNaker code was used with some modifications. The code used to parse the network and determine the parameters stored for each node remained the same, but changes were made to the code controlling the placement of the nodes on the board because the Adapteva Epiphany chip where sampling was run has 16 nodes arranged in a 4x4 grid whereas the SpiNNaker has 64 nodes that were used to generate samples arranged 16 to a chip in a 2x2 chip arrangement.

The code used to communicate with the Parallella was also different than the code used to talk to the SpiNNaker. The SpiNNaker relies on UDP communication to load data onto the board whereas the Parallella runs Ubuntu and can therefore be sent data using the standard Linux Secure Copy (SCP) command. The host application on the Parallella loads the data from the files and places data into the Epiphany's external memory so there is a maximum amount of space for everything. The memory is divided into 16 sections, one for each Epiphany core, and the data for each core is placed into that section. Then the host application tells the Epiphany that everything is ready and it can begin sampling.

On the Epiphany side each core has access to the entirety of this memory, so a special section was allocated for each core to receive messages. The offset for the starting address of this location was the same on each core so that the other cores

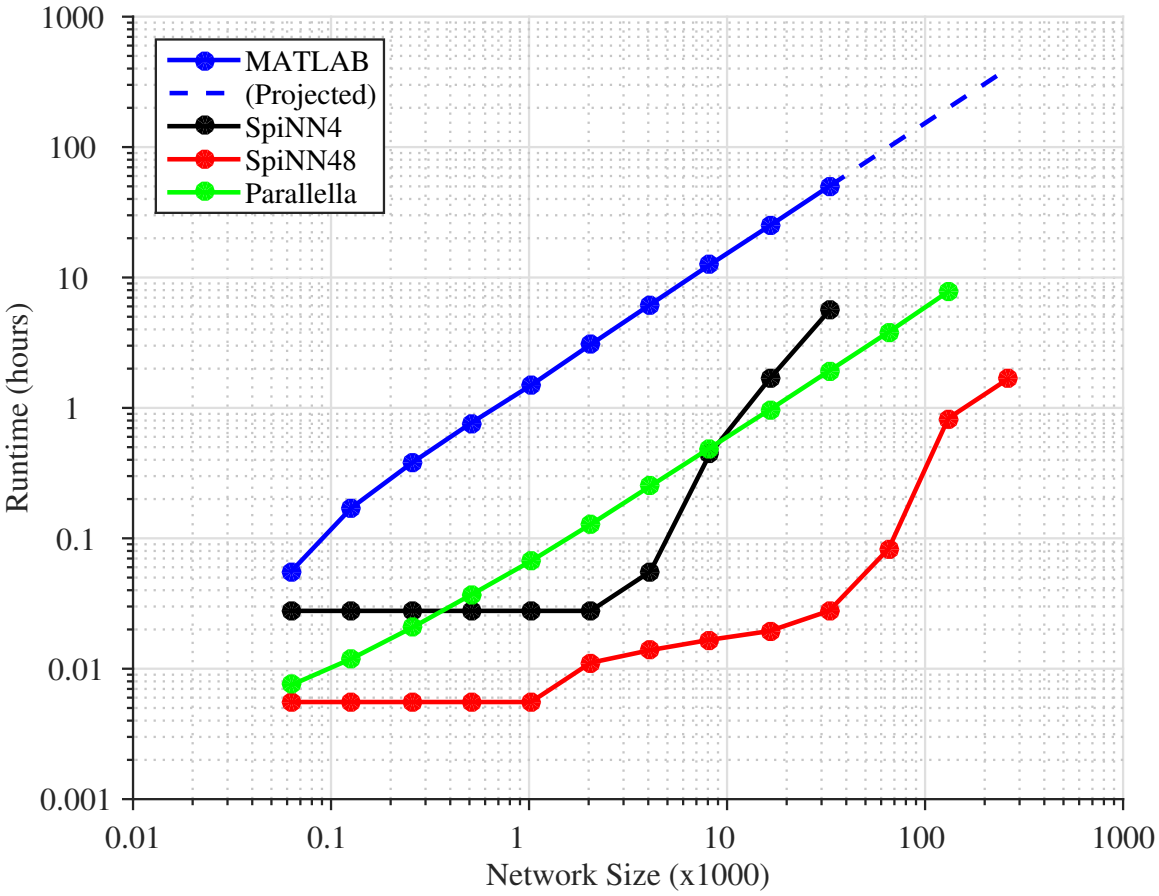


Figure 5.4: Runtimes for 50,000 iterations of the Neural sampling algorithm on four platforms: single-threaded MATLAB on a Core i7 PC, 4-node SpiNNaker, 48-node SpiNNaker, and 16-node Parallella.

would know where to put the message data when it was sent out to other boards. Then the only other information needed is the cores and offsets from the starting address to send the data to, so those were stored for each node. Instead of sending a message from chip to chip as in the case of the SpiNNaker, the incoming message sections of memory were simply populated instead.

Rather than using a global timer tick to synchronize all the cores as was done with the SpiNNaker, the Epiphany ESDK provides for barriers to synchronize all the

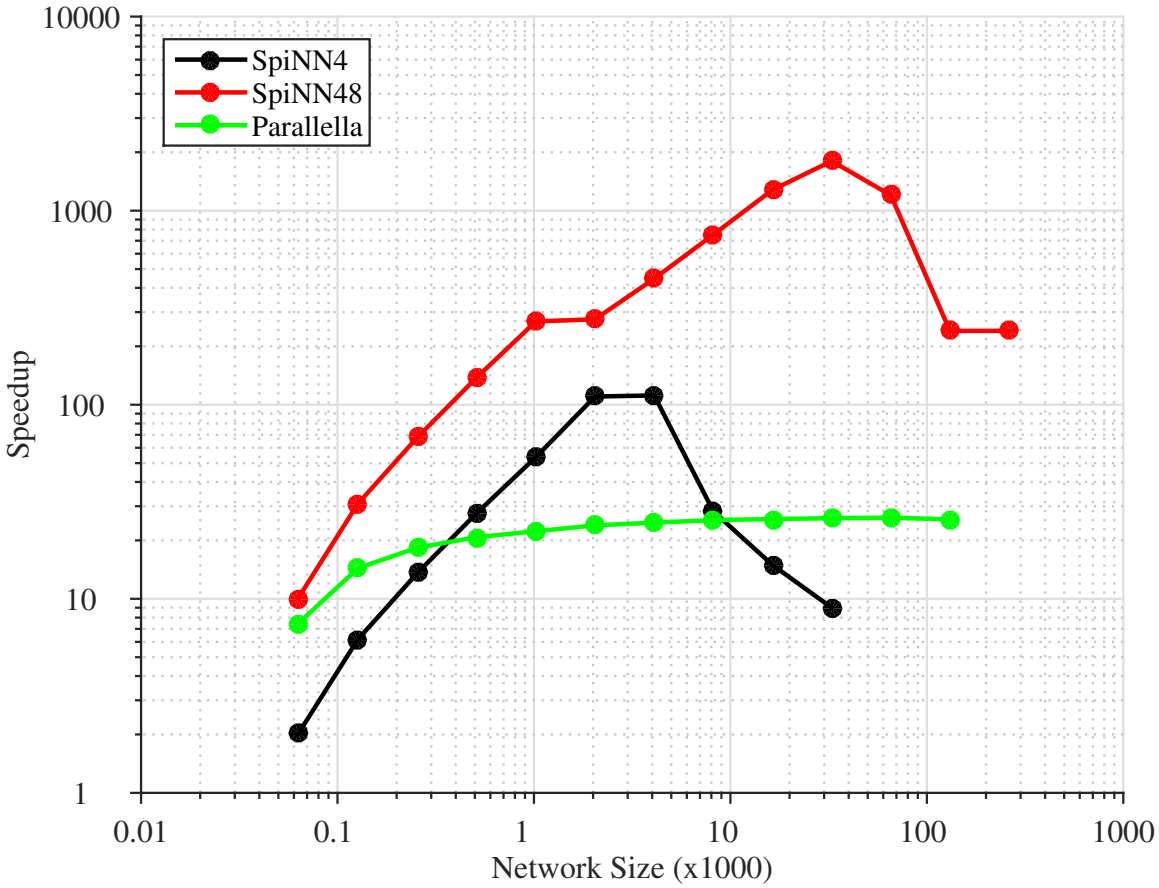


Figure 5.5: Speedups for 50,000 iterations of the Neural sampling algorithm on three platforms as compared to running it single-threaded using MATLAB on a Core i7 PC: 4-node SpiNNaker, 48-node SpiNNaker, and 16-node Parallella.

cores. These barriers were used to ensure that each core generates new samples at the same time and that each core does not read the stored incoming messages until all the other nodes have completed their sampling procedures and all the cores are ready to move onto the next sampling iteration.

The MATLAB code controlling the Parallella remotely from a desktop PC waits for the sampling process to complete and then sends the results back in a simple file, again using SCP. Then the conditional probability values can be examined as they

were when the data were returned from the SpiNNaker.

Binary tree-like networks of 6 through 17 layers were run on the board. At 18 layers there was not enough memory to store all the parameters, although better data packing techniques could alleviate that memory pressure at the expense of speed due to the operations required to pack bits tighter in memory. The runtime comparison for various Neural sampling implementations is shown in Figure 5.4. This runtime comparison includes all of the major binary Neural sampling algorithms described thus far in the thesis. The blue line corresponds to running the single-threaded MATLAB implementation on networks of 6 through 15 layers with a linear extrapolation of runtimes up to 18 layers. The black line corresponds to running Neural sampling in parallel on the 4-chip SpiNNaker for networks of 6 to 15 layers. The red line shows the runtimes for running parallel Neural sampling on the large 48-chip SpiNNaker on networks of 6 to 18 layers. Finally, the green line shows the results of running Neural sampling in parallel on the Parallella hardware using networks of 6 through 17 layers.

Speedups of the three parallel implementations on specialized neuromorphic hardware are shown in Figure 5.5 as compared to the reference single-threaded MATLAB implementation run on a Sandy Bridge era Core i7 PC. In the fastest scenario the 48-chip SpiNNaker achieves almost a 2000x speedup over the MATLAB implementation, the 4-chip SpiNNaker is over 100x faster, and the Parallella is well over 20x faster.

These speed increases are clearly not constant given a network size, and that

is due to many factors. One factor is the size of each color groups created when parallelizing the sampling procedure. For the binary tree-like networks described in this thesis there are 4 color groups in each network using the heuristics used here to create the groups. Therefore, for the smaller networks there are less nodes in each color group which means less nodes can be sampled in parallel. Sampling less nodes in parallel provides less speedup potential for the neuromorphic implementations. Another factor is the network used in each hardware architecture. For both SpiNNaker platforms as the networks grow larger the speedups decrease after a certain point. These speed reductions are likely due to the fact that many more messages must be sent through the network in order to update all the nodes that depend on the value of the current node for their sampling distributions. As more messages are sent more network traffic is present, and slowdowns occur as a result. On the other hand, the implementation on the Parallella has each node access memory to transmit messages, and while slowdowns can occur these are not deadlock-type issues that can occur on the SpiNNaker's mesh network where messages are sent in multiple directions simultaneously.

Another set of experiments was run on the 9-layer, 511-node network. Here the number of sampling iterations was varied from 1000 to 100,000, and the accuracy of the implementation was examined by looking at the mean absolute error between Gibbs sampling on the PC and Neural sampling on the Parallella and the SpiNNaker for the inferred probability of each node being 1 given the evidence. The Gibbs sam-

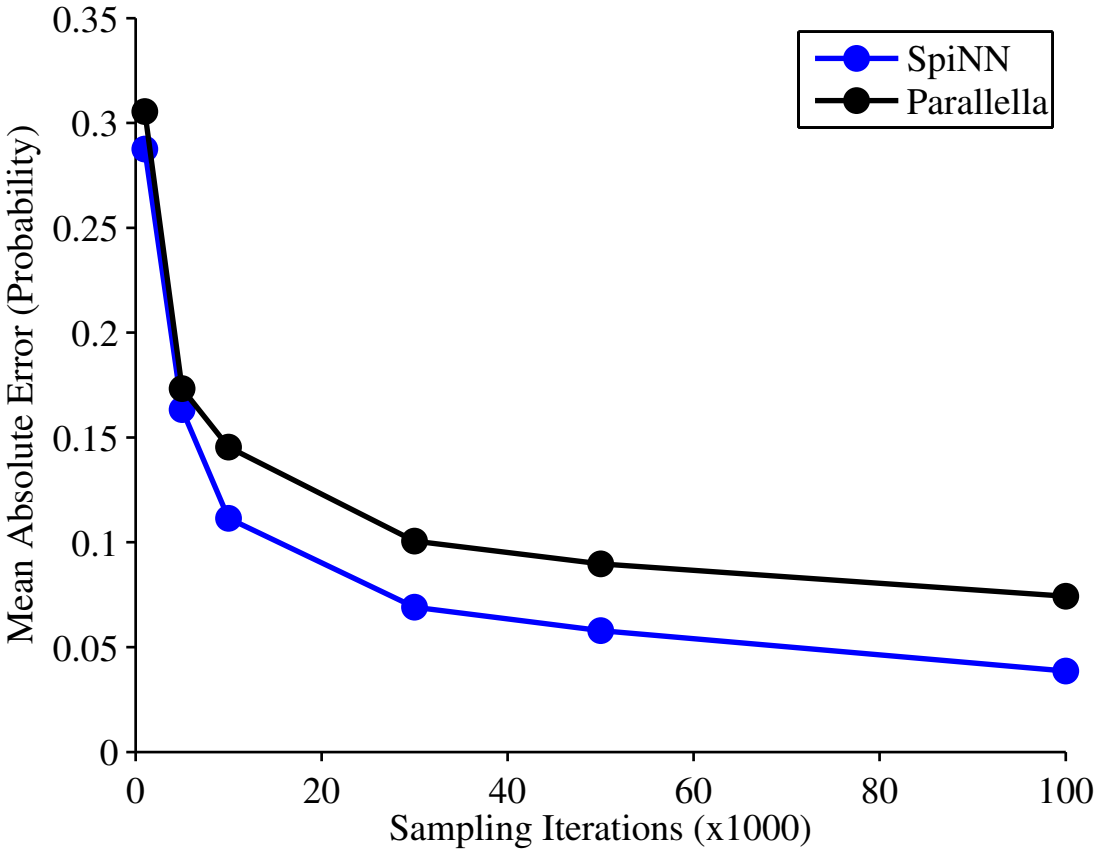


Figure 5.6: Mean absolute error values for inferred probability values of all the nodes in the 9-layer, 511-node network. Results from the SpiNNaker and Parallella were each compared to the Gibbs sampling implementation from Kevin Murphy's BayesNet Toolbox.

pling implementation used is from Kevin Murphy's BayesNet Toolbox. See Figure 5.6 for these results as well as a comparison to those of the SpiNNaker.

As the number of sampling iterations increases the differences between the implementations decrease which is reasonable. However, it looks like the SpiNNaker is closer to the results on the PC than the Parallella is. One implementation difference is that the C rand function on the SpiNNaker returns integers in a larger range than the Parallella does, so it is possible that this architecture difference accounts for at

least some of the difference since the rand function is used for generating samples.

5.3 Spatial Locality on the SpiNNaker

Thus far all of the node placement algorithms have not taken into account spatial locality. The nodes are placed around the SpiNNaker and Parallella hardware without regard for the other nodes they need to communicate with. However, nodes that need to communicate with one another should theoretically be located close to one another physically on the hardware. That way the messages can be passed to the appropriate nodes more quickly and the overall amount of traffic through the network is minimized.

In order to achieve these goals the directed Bayesian networks were converted to an undirected network, and the graph was moralized so the parents in the directed network were connected in the resulting undirected network. Connecting the parents preserves the Markov Blankets that were present in the original directed network.[42]

A search was done so that for each node all its neighbors would be added first to the SpiNNaker. One core was filled up at a time, and each core on a chip was filled before the next chip was populated. This procedure ensures that at least some nodes that frequently talk to each other (neighbors) are on the same core and therefore do not need to add any traffic to the mesh network connecting all the SpiNNaker chips.

On the large board, the smaller networks using spatial locality ran more quickly

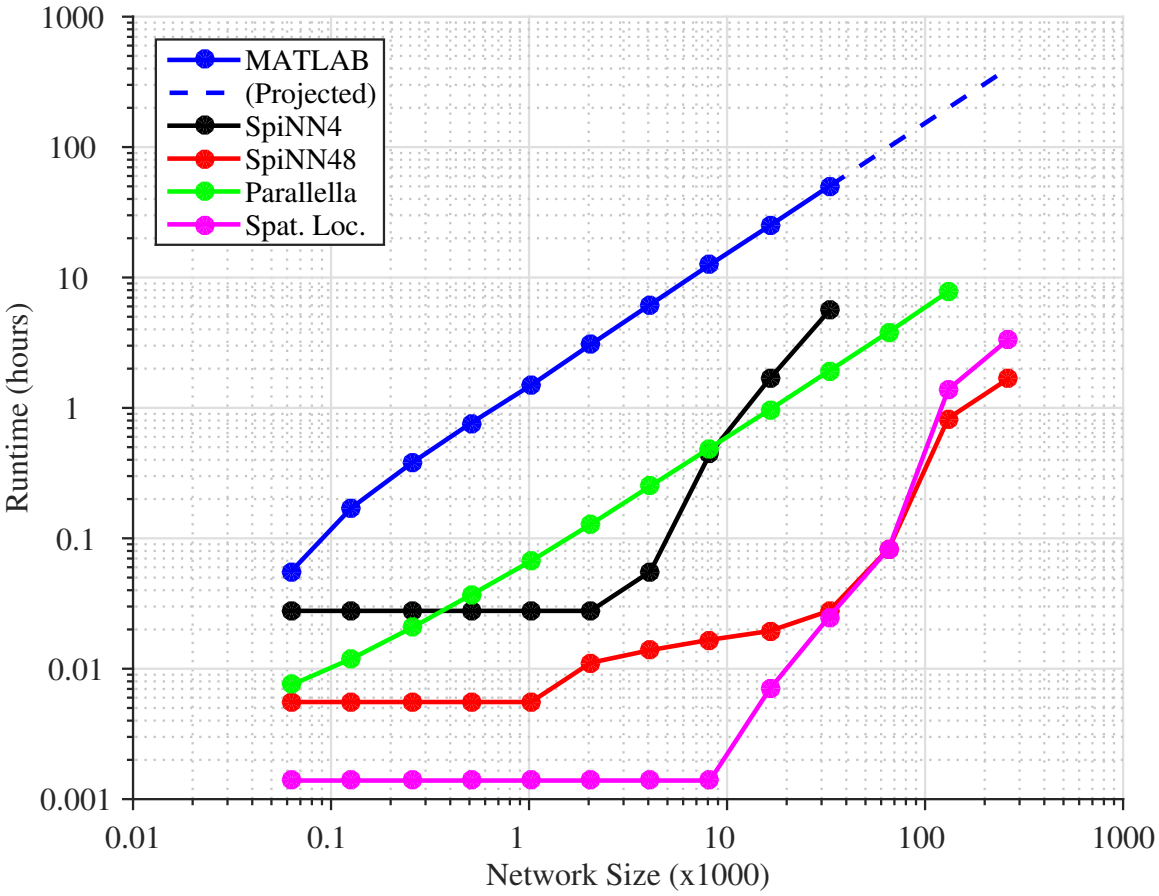


Figure 5.7: Runtimes for 50,000 iterations of the Neural sampling algorithm for five situations: single-threaded MATLAB on a Core i7 PC, 4-node SpiNNaker, 48-node SpiNNaker, 16-node Parallella, and 48-node SpiNNaker with spatial locality.

than the regular Neural sampling process, but when the networks became larger the performance dropped off. For example, in networks of up to 13 layers the spatial locality version was faster by up to a factor of 12 depending on the network size. But for the network of 16 layers both took the same amount of time, and the spatial locality runtimes became larger as the network size increased. See Figure 5.7 for a plot of the runtime results and Figure 5.8 for the speedups.

Other potential improvements to the heuristics used in taking advantage of spatial

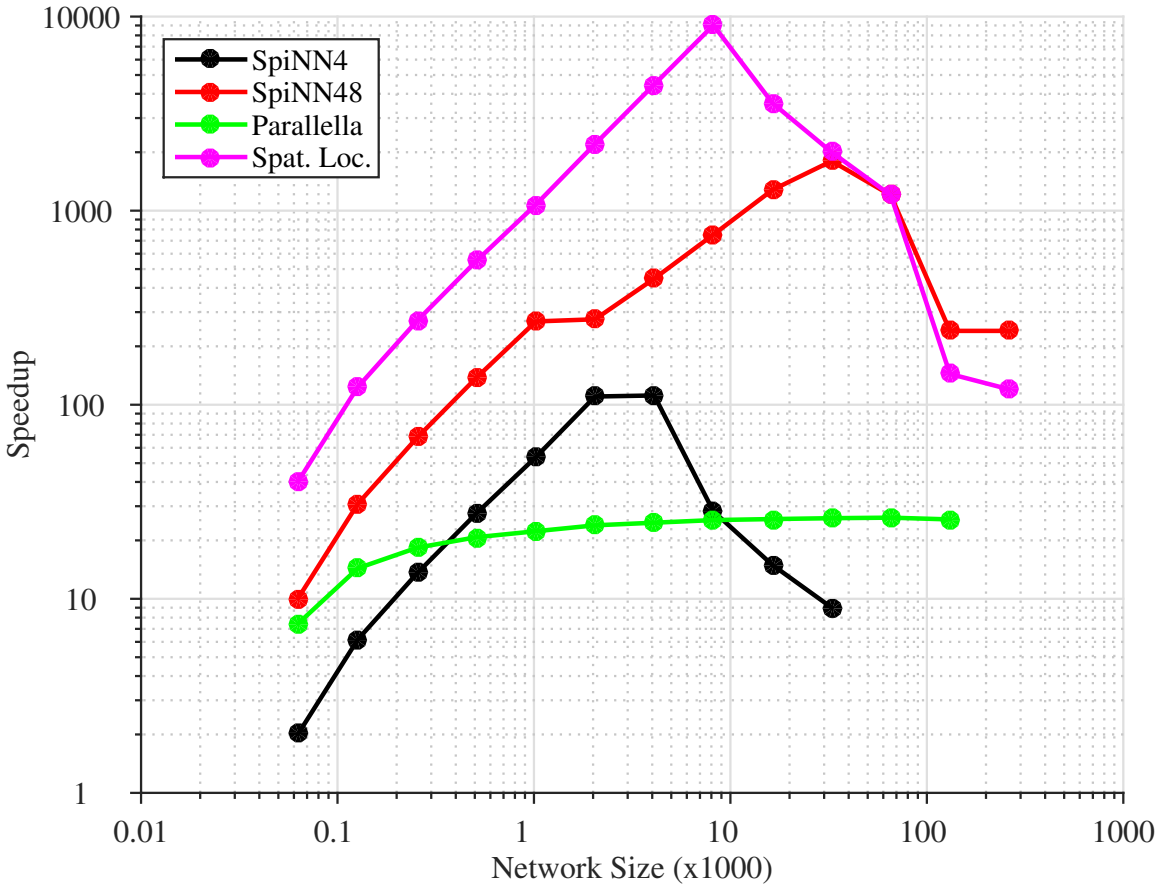


Figure 5.8: Speedups for 50,000 iterations of the Neural sampling algorithm on four platforms as compared to running it single-threaded using MATLAB on a Core i7 PC: 4-node SpiNNaker, 48-node SpiNNaker, 16-node Parallella, and 48-node SpiNNaker with spatial locality.

locality exist. While nearby nodes are roughly placed onto cores in the same chip, this technique is not perfect. Nearby neighbors are traversed and added to the core, but eventually another node must be chosen to be the seed for the other neighbors to add. Thus, some neighbors must be separated from the other neighbors by using this technique. And beyond keeping neighbor nodes on the same core, once one chip is filled up the numerically-next chip is populated. However, depending on where

the nodes on that chip are located in the original graph, those nodes may be better positioned on another chip that is physically closer to where the node's neighbors are located on the SpiNNaker.

5.4 SpiNNaker Complexity Analysis

This section describes the challenges and techniques involved in each of the main steps of performing Neural sampling on parallel hardware. Each subsection details a different step of the process. In all the sections below the number of nodes is denoted as N and the number of edges is denoted as E. These complexity numbers are kept in general terms here so it is important to keep in mind the actual time it takes to perform each action. For example, when values are read from files located on hard drives or network storage devices these operations may take longer than other parts of this process that might load values from RAM.

5.4.1 Load Network from File

The file describing the network structure first contains a list of all the nodes in the directed acyclical graph, one per line. Each edge is listed on the lines that follow the node list. Adding the nodes in the file to the list of all nodes takes $O(N)$ steps. Then the nodes are sorted once so the next step (adding edges) is sped up. That is another $O(N \log N)$ steps using MATLAB's Quicksort implementation.

Once the nodes are collected and sorted all the edges must be traversed to store them in the data structure. Each edge is added to both the parent and the child nodes so that the parent knows its children and the child knows its parents. In order to add an edge the parent is located in the list of nodes and the child is added. A similar process occurs for the child node. Locating each of these nodes takes $O(\log N)$ with a binary search, so adding the edges is $O(2E \log N)$.

The list of parents for each node is also sorted, but that is very quick ($O(P \log P)$ on average per node where P is the average number of parents per node) assuming that each node only has a small number of parents. This assumption must be true in order for the size of the probability tables to not become too large. Each new sample depends on the value of every node in the current Markov Blanket and that space becomes large fast because it grows exponentially with the number of nodes in the Markov Blanket. Since a sampling distribution is stored for each node given the state of the entire Markov Blanket, this table must remain small in order to work on the SpiNNaker.

The overall complexity is $O(N + N \log N + 2E \log N + NP \log P)$. Given that $P << N$ and $P << E$ the complexity is roughly $O((N + 2E) \log N)$.

5.4.2 Load CPD Tables from File

The first step is to create an empty CPD table at each node. This involves looping over the nodes and determining how many parents each node has. For the case of

Neural sampling each node is binary so there are 2^{P_n} rows in the table (where P_n is the number of parents for node $n \in \{1, 2, ..., N\}$) that cover all possible values of the nodes in the Markov Blanket. If P is the average value of P_n over all nodes then this process is $O(2^P \cdot N)$.

Next, each line of the file is iterated over. Each line corresponds to one combination of values of the Markov Blanket so there are 2^P lines per node on average. The node must be found in the list of nodes which is $O(\log N)$ since the list is sorted (binary search). If there are any parents then their values must be determined by parsing the line of the file. That parse is $O(P)$ on average. Using the values of the parents the row of the CPD table is determined and the probability values are inserted. Since the nodes are binary for Neural sampling, only one value is specified and the other is such that the two add to 1.

The overall complexity is $O(2^P \cdot N + N \cdot 2^P \log N \cdot P))$. Given that $P << N$ and that P is very small, the complexity is roughly $O(N \log N)$.

5.4.3 Determine Markov Blankets

Once the structure of the network and its CPD tables are established the next step is to determine where each node needs to send data. One important step toward defining these connections is to determine the Markov Blanket of each node. Since each sample is generated based on a distribution that depends on the values of all nodes in the Markov Blanket, each node in the Markov Blanket must know to send

its current value to the node under consideration. This process must be completed for each node in the network.

The Markov Blanket consists of a node's parents, children, and coparents. Using the procedure described above, a given node knows its parents and children directly so they can simply be listed, which on average is $O(P + C)$, where C is the average number of children per node in the network. Then the children must be iterated over and their parents are included in the Markov Blanket: $O(C(P - 1)) = O(CP)$ since for each child its parents must be iterated over.

Thus the overall complexity is $O(N(P + C + CP))$. Given that $P << N$ and $C << N$ the complexity is $O(N)$.

5.4.4 Determine Color Groups in the Graph

Nodes of the same color are nodes that can be sampled in parallel. This means that all the nodes of one color do not directly depend on each other's values when sampling in the current iteration. Therefore, nodes that are not in each other's Markov Blankets can belong to the same color group and can be sampled at the same time.

This work's implementation chooses these groups by iterating over each node in the graph and adding them to color groups in a greedy fashion. A given node's Markov Blanket is compared to all the nodes in the first color and the Markov Blanket for each node in that color group is compared to the current node. If there are no matches

then the node is added to that group. Otherwise the next group is checked and so on.

The worst-case scenario in terms of complexity of constructing the color groups is that all the nodes are unconnected in the graph and they can therefore all be sampled in parallel. This is, on the other hand, the most efficient way for sampling to be carried out later on since it provides the maximum amount of parallelism. In this scenario all the nodes are added to the same color group so every incoming node must be compared to every other node in the group. The first node is added with no required checks ($O(1)$). For the second node, the Markov Blanket of the first node is traversed to see whether any members are the new node. Then the new node's Markov Blanket is traversed to see whether any members are the first node. That total process is, on average, $O(2(P + C + CP))$ where P is the average number of parents for a node in the network and C is the average number of children. The third node needs to perform the same process but two times as often because there are two times as many nodes already in the color group to iterate over: $O(4(P + C + CP))$.

This process continues which results in the following worst case complexity:

$$O\left(1+\sum_{i=2}^{N}(i-1)2(P+C+CP)\right) \tag{5.1}$$

$$= O\left(1+2(P+C+CP)\sum_{i=1}^{N-1} i\right) \tag{5.2}$$

$$= O\left(1+\frac{2(P+C+CP)N(N-1)}{2}\right) \tag{5.3}$$

$$= O\left(1+(P+C+CP)N(N-1)\right). \tag{5.4}$$

Given that $P << N$ and $C << N$ the complexity is approximately $O(N^2)$.

5.4.5 Calculate Markov Blanket Probability Tables

When sampling is performed each node must generate new samples according to the distribution of that node given the current sampled values of all the other nodes in the network. This boils down to knowing the values of all the nodes in the Markov Blanket (see Figure 2.4 for an illustration of this fact and the discussion following that figure for an example of the mathematics involved in these computations). In the current implementation of Neural sampling, each node stores a table of distributions based on the current status of its Markov Blanket, and this section describes that process.

The two main loops are to first loop over all the nodes ($O(N)$) and then loop over

each configuration of the Markov Blanket ($O(2^{(P+C+CP)})$).

For each Markov Blanket configuration the parents are iterated over ($O(P)$ on average), and the CPD table of the current node is queried given its parents' values. In Neural sampling there are two different factors considered which are the numerator (current node is 1) and denominator (current node is 0) of the log ratio (see Equation 2.42), so for all these values two different cases are considered and stored.

A similar process is performed for the children. Each child is iterated over ($O(C)$ on average). The parents of each child are then iterated over ($O(P)$ on average) to retrieve the value for the factor associated with that child node given the current Markov Blanket configuration being examined.

Once these factors are stored two steps must occur. The first is to determine where to put the sampling distribution in the Markov Blanket table of probability distributions. Thus the column for each node in the Markov Blanket must be found in the table which is $O((P + C + CP)\log(P + C + CP))$ since the nodes are sorted. Once the columns are determined the row to fill in is easily calculated.

The second step is to multiply all the factors for the numerator and denominator together. Since there are C children on average there are roughly C factors for the numerator and the same for the denominator to multiply together. In the denominator this must be repeated once for the Neural sampling algorithm and potentially more times in general Gibbs sampling because the denominator is a normalization procedure and the process must be repeated for every value the current node can

have. Since this is a small constant number of repetitions it is omitted from this calculation.

Thus the overall complexity is $O(N \cdot 2^{(P+C+CP)} \cdot (P+CP+(P+C+CP)\log(P+C+CP)+C))$. Given that $P << N$ and $C << N$ the complexity is approximately $O(N)$.

5.4.6 Arrange Nodes on the Board

Once the parameters associated locally with each node are established the nodes must then be physically arranged on the board. Two different techniques have been used and they are each described in subsections that follow.

5.4.6.1 Simple Arrangement

In the simple case the nodes are arranged on the board by iterating over the nodes and simply placing each successive node in the next specified location following a fixed order list of locations on the board. However, before that is done the nodes are sorted by color so it is simpler to iterate over all the nodes in a given color during sampling. This sort is done by looping over the colors and then over the nodes. Each node is checked to see whether it matches the current color and if it does then it is added to the color. Therefore, the complexity for this sort is $O(N)$.

Once the nodes are sorted assigning them to a location is $O(N)$ because each successive node is assigned to a list of predetermined locations that are cycled through

as the nodes are iterated over.

The overall complexity in this case is $O(N+N)$, so the complexity is approximately $O(N)$.

5.4.6.2 Exploit Spatial Locality

Exploiting spatial locality in a perfectly optimal way is not a solved problem (see Section 5.3). However, the following is a complexity analysis for the steps that were conducted according to the explanation in Section 5.3, where greedy heuristics were used to obtain the results shown in this thesis.

Moralizing the graph begins by creating a new list of nodes of an undirected graph ($O(N)$). Then each node of the directed graph is iterated over ($O(N)$) and nodes from the directed graph that should be neighbors in the moralized undirected graph are added as neighbors to the new graph. These nodes include the parents and children of each node $O(P+C)$, but in addition for each parent all the other parents must be connected as neighbors which is $O(P(P-1))$ for a naive implementation. Thus the graph moralization process is $O(N+N(P+C+P(P-1)))$.

The nodes are then placed by group into each core in turn. Once a core is filled up the next one on that chip is entered and it is filled up. This process continues until the board is full of nodes. Each core stores approximately $N/768$ nodes since this thesis uses 768 cores on the large SpiNNaker board. This process is described in more detail in the next paragraph.

The nodes in the network are iterated over. The next unvisited node (unsorted search so $O(N)$) is chosen to be added to the current core. Its neighbors are iterated over ($O(P + C + CP)$ on average) and they are added to the group if they were not already visited. They are also set to be visited ($O(3)$ each time). When all the neighbors are exhausted the next unvisited node is chosen and the process continues until the core is full. Once the core is full the next one is chosen and the process continues again. This paragraph describes a process that is $O(N(P + C + CP))$ in the worst case.

More work needs to be done in order to get the nodes on each core ready. Each core is iterated over ($O(768)$) and the group of nodes on the core is sorted by color group ($O(N/768)$). Finally the official positions are recorded in another format so the rest of the toolchain can use the positions ($O(N)$) for directing messages that are sent from node to node on the board during the sampling process.

The overall complexity is then $O(N + N(P + C + P(P - 1)) + N(P + C + CP) + N + N)$. Given that $P << N$ and $C << N$, the complexity is approximately $O(N)$.

5.4.7 Generate Routes

After the nodes' locations are chosen on the board, the next step is to set up routes for information to flow across the board. Each chip contains a router that is used to redirect incoming packets based on 1024 possible programmable routes. Each route is associated with a key. In this work destination-based routing is used, so each

destination on the board corresponds to one key value, and each router around the board has a route specified that will move the packet toward its destination no matter where the packet is currently located on the SpiNNaker.

There are 48 chips, each containing 16 cores to be addressed in this work which means there are 768 possible destinations. Each chip contains a router, so there are 48 routers to be programmed for all those destinations. This means the complexity is $O(48 \cdot 768) = O(36,864)$. The calculation of the key and route parameters for each of these router entries is at most a constant value since the determination involves only a simple check of the current chip's location and what the destination for the route is.

Thus the overall complexity is $O(36,864) = O(1)$.

5.4.8 Perform Sampling

Sampling itself is performed for a chosen number of iterations I in this thesis. During each iteration N nodes are sampled, so if this is done without parallelizing the process the number of steps are $O(NI)$. However, in this implementation, during each iteration color groups are sampled in parallel one at a time until all groups are sampled. Despite the fact that one color is sampled after another, all the nodes in the network are sampled during each iteration. Assuming that there are at least 768 nodes in each color it can be assumed that the degree of parallelism is roughly 768 despite the fact that the nodes cannot typically be divided perfectly evenly into groups

of multiples of 768 nodes. Therefore, with that caveat in mind, the parallel complexity is approximately $O(NI/768)$ with regard to how many sampling operations are completed.

During each iteration the first step is to copy all the messages that came in during the last iteration to the current buffer for use. That copy has a constant cost, B, each iteration that depends on the connectivity of the graph. Then the current value of the node is checked ($O(1)$) to see whether it is an observed node that should not be sampled. If it is observed then its value does not change. Otherwise, if the neuron is refractory two parameters are updated to keep track of the refractory state and its current value ($O(2)$).

If the neuron is not refractory the sampling distribution is determined by looking at the node values in the Markov Blanket and determining which row of the probability table should be addressed. Checking the Markov Blanket node values is $O(P + C + CP)$ because all the nodes involved are sorted, including the storage of current values, so no search is required. That value is retrieved ($O(1)$), and a random number is generated to sample from that distribution ($O(1)$). Depending on the sampled value the neuron's refractory state and value are updated ($O(2)$).

Once all the nodes are sampled, messages are sent to other nodes on the board that need to know the updated values. Let N_o be the average number of outputs per node and then the complexity of sending messages overall per node is $O(N_o)$ since a separate message must be sent for each output using destination routing. When

the messages are received the values must be placed into the proper location which is $O(\log(P + C + CP))$ on average since the binary search is over the Markov Blanket.

Some details about timing are glossed over here though and the actual timing of these events is faster than described here. Node values are sent across the network right after all the samples on a given core are created, so many cores finish at different times. Thus some cores are interrupted to receive messages while they are still creating samples, and complicated timing interactions occur including the time it takes for each packet to traverse the network.

The overall complexity of sampling is then $O(NI/(768) \cdot (B + P + C + CP + 4 + N_o \log(P + C + CP)))$. Given that $P << N$, $C << N$, $N_o << N$, and B can be a significant fraction of N despite parallelism, the complexity is roughly $O(NIB)$ on average.

5.5 Heterogeneous Architecture

A heterogeneous architecture was created that combines both the Parallella and the SpiNNaker in order to see how the two systems can be combined to perform Neural sampling. The SpiNNaker and the Parallella have different characteristics that make them better-suited for different computations. For example, the SpiNNaker has a large number of fixed-point processors running at a relatively low clock speed while the Parallella has less processing units but they have floating-point capabilities and

run at a higher clock speed. Different computations can thus be run on each of the systems within the context of an algorithm to minimize the delay required for producing results and/or to reduce the power consumption of the architecture for a given task.

As a first step the two systems were connected together to share the burden of performing Neural sampling. In this paradigm the tasks done by each hardware system were identical to keep things simple and to see how computations can be sped up by combining the two architectures in a straightforward manner.

5.5.1 Heterogeneous via Ethernet

The two systems were connected to the same network via ethernet so that they could each send current samples to each other during each sampling iteration. Since the SpiNNaker cores are synchronized using a timer tick, the SpiNNaker controls the runtime for each sampling iteration. At the start of each iteration the SpiNNaker sends a UDP packet to the Parallella which tells the Parallella it should start executing its sampling iteration. TCP packets would ensure more reliability but the SpiNNaker requires UDP packets for communication with devices on the network.

The same general process occurs on both the Parallella and the SpiNNaker during each iteration. Each core generates new samples for the nodes it is responsible for, and once all the samples are generated the values are sent to the other nodes that need to know those values.

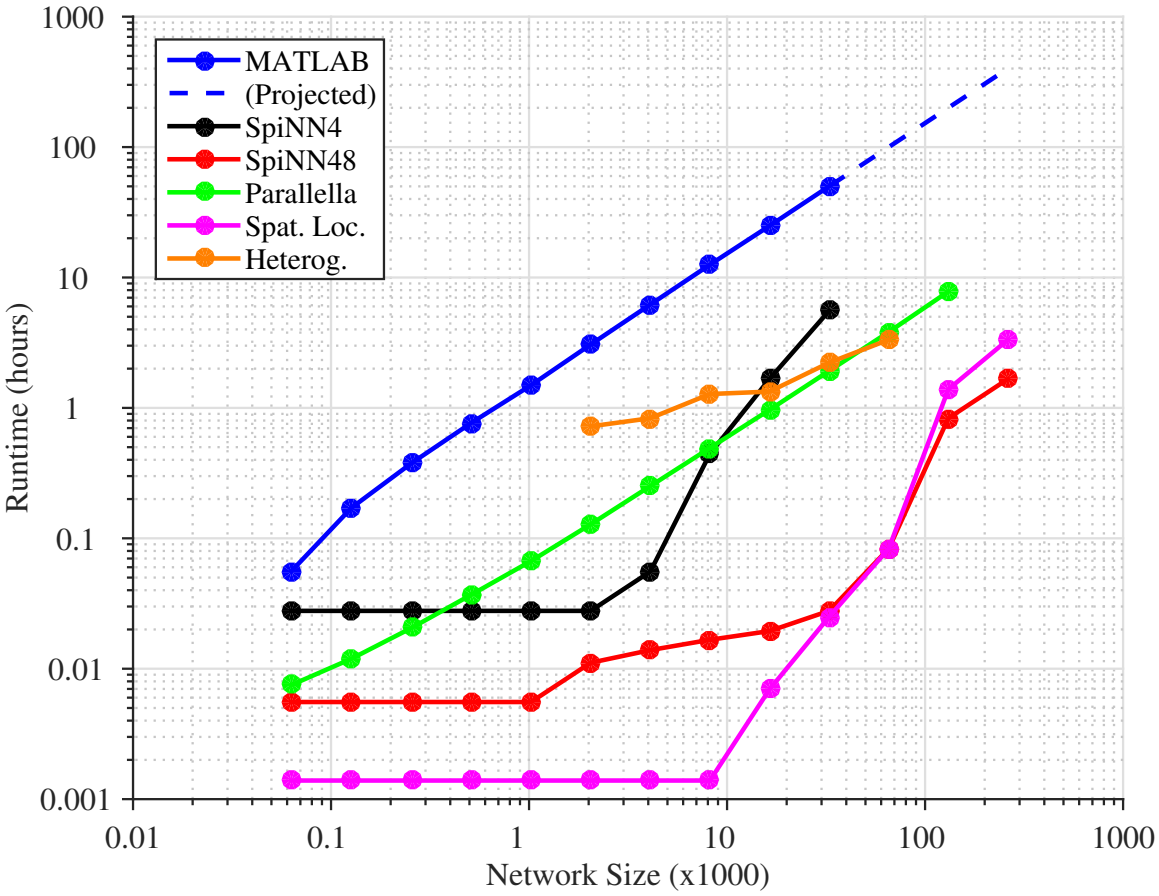


Figure 5.9: Runtimes for six implementations of Neural sampling: single-threaded MATLAB on a Core i7 PC, 4-node SpiNNaker, 48-node SpiNNaker, 16-node Parallella, 48-node SpiNNaker using spatial locality, and the heterogeneous architecture consisting of the 48-node SpiNNaker and the 16-node Parallella connected by Ethernet.

The mechanics of transmitting these sample values varies depending on which node is considered, its position in the graph, and the location of the node and other nodes that have that node in their Markov Blankets. For example, if the node is on the SpiNNaker and the new sample value needs to be sent to another location on the SpiNNaker, a multicast packet is sent in the same way that it is sent when Neural sampling is run only on the SpiNNaker. On the other hand, when the node is on the

Parallella and the information must be sent to another node on the Parallella, the proper memory location is populated with the new sample value just as it is when the algorithm is run only on the Parallella. When a message must travel from the SpiNNaker to the Parallella or vice versa, a UDP packet is sent to achieve that goal.

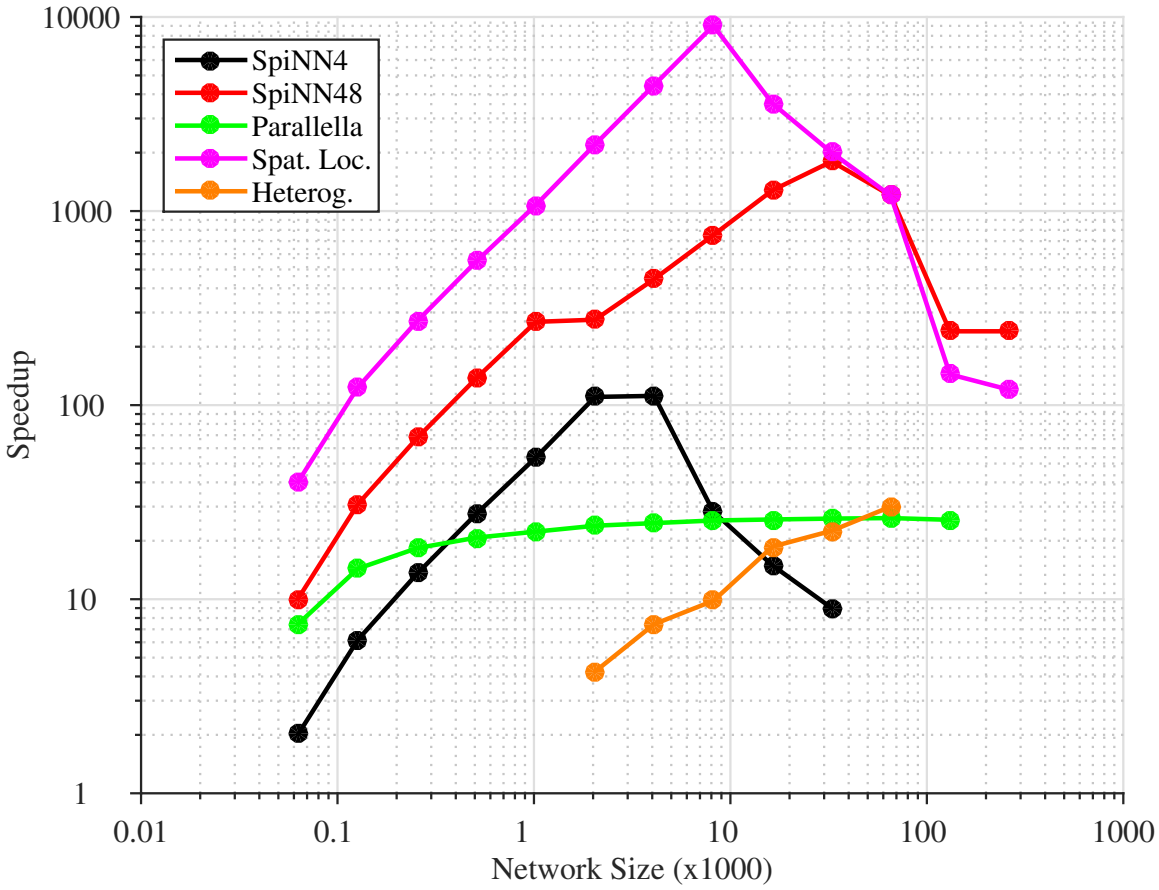


Figure 5.10: Speedups for the Neural sampling algorithm on five platforms as compared to running it single-threaded using MATLAB on a Core i7 PC: 4-node SpiNNaker, 48-node SpiNNaker, 16-node Parallella, 48-node SpiNNaker with spatial locality, and the heterogeneous architecture consisting of the 48-node SpiNNaker and the 16-node Parallella connected by Ethernet.

The SpiNNaker API takes care of the mechanics of using the hardware to send and receive UDP packets via ethernet. On the Parallella the communication code

must be run on the host ARM core using Ubuntu. Therefore, to trigger events such as sending or receiving information via ethernet to/from a Parallella Epiphany core, memory values are changed so that the Epiphany cores can communicate with the host ARM cores. The ARM cores poll certain memory addresses to see whether there is any information that should be sent via ethernet in addition to managing when the program is done being executed on the parallel cores of the Adapteva Epiphany. That way, when an Epiphany core wants to send a message it flags the proper location in memory and some bytes immediately following that flag area contain the message to be sent.

Binary tree-like networks of 11 through 16 layers were run using the heterogeneous architecture. The runtime comparison for various Neural sampling implementations is shown in Figure 5.9, and the speedups are shown in Figure 5.10. This ethernet-based heterogeneous architecture is slow because the latency of sending data via ethernet is relatively high compared to sending data across a single board. Since spikes need to be sent to other nodes this increased delay has a large effect on the speed of executing the sampling algorithm. However, note that as the networks grow larger the heterogeneous architecture's runtimes do not slow down drastically. Because the nodes are distributed across two different platforms and communication is limited by ethernet, the slowdowns that occur as the network becomes saturated on the SpiNNaker are not yet seen here.

An accuracy comparison was also done on a network having 11 layers (2047 nodes).

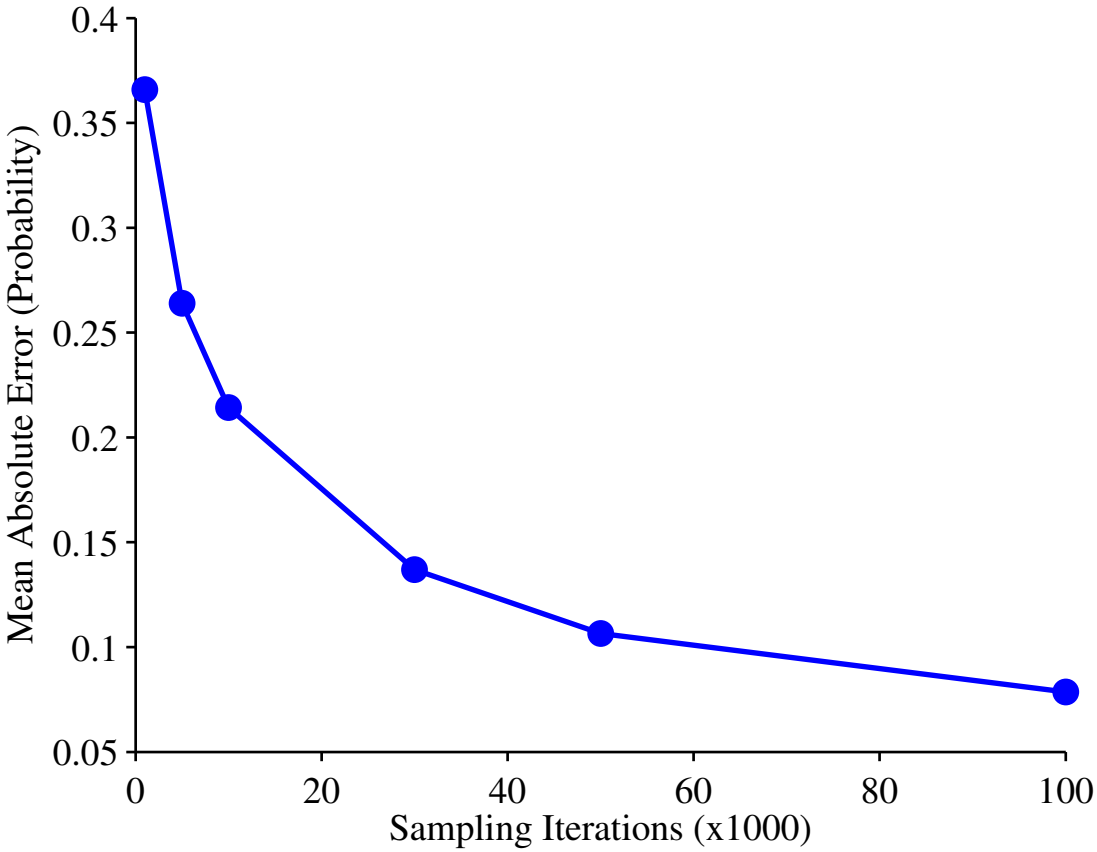


Figure 5.11: Mean Absolute Error (MAE) values for the inferred probability values of all the nodes in the 11-layer, 2047-node network using Neural sampling. These results compare the heterogeneous Parallella/SpiNNaker architecture to the results from Kevin Murphy's BayesNet Toolbox.

Sampling was performed on this network using Kevin Murphy's BayesNet Toolbox, and these results were compared to those of the heterogeneous architecture. The number of sampling iterations was varied from 1000 to 100,000 and the Mean Absolute Error (MAE) between the inferred probability of each node being 1 using each of the sampling frameworks was computed. These results can be seen in Figure 5.11. As the number of sampling iterations increases both techniques converge to the true value as expected.

5.5.2 Heterogeneous with Interconnect Board

In order to speed up heterogeneous computations the ethernet link must be replaced with something faster, so an interconnect board (See Figure 5.12 and Figure 5.13) was designed by Alejandro Pasciaroni to provide a platform for sending messages with less latency. The board contains a connector for the SpiNNaker link located at the top of the large SpiNNaker board as well as a connector for the north eLink of the Parallella so that the two devices can send packets to the board. There is a second eLink connector on the board but it is there only for structural support and is not connected to anything on the interconnect board.

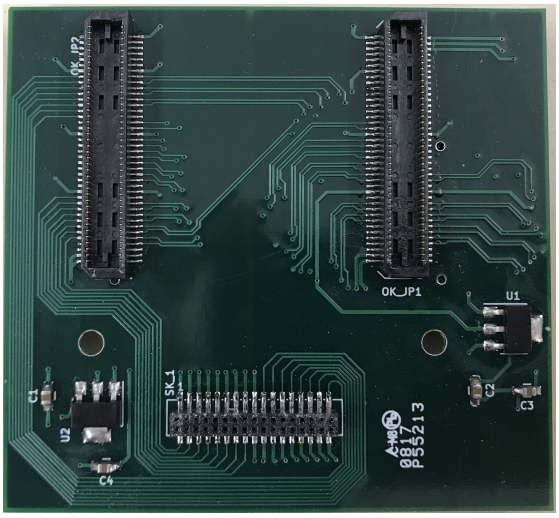


Figure 5.12: Heterogeneous interconnect board bottom view.

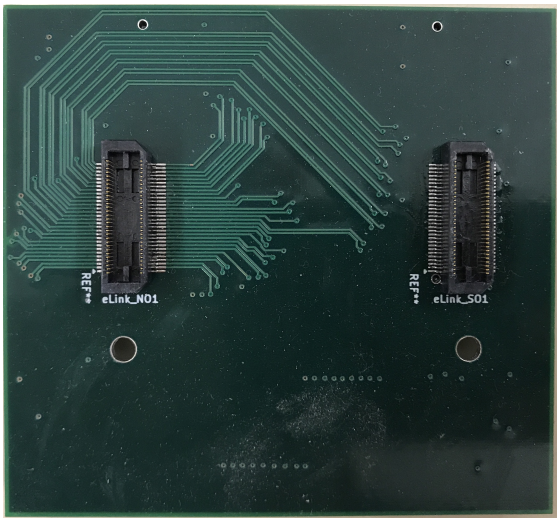


Figure 5.13: Heterogeneous interconnect board top view.

The data protocol the SpiNNaker uses is different than the data protocol the Epiphany uses, so the interconnect board has connectors that mate to the expansion connectors on an Opal Kelly XEM6310 and other compatible FPGA boards. The

inclusion of the FPGA provides the capability to communicate with both the SpiNNaker and the Parallella as well as convert data from one format to the other so that data can be passed from one device to the other and vice versa. Voltage regulators are placed on the board as well so that the signals from each device can be sent through the FPGA even though they have different maximum voltage levels.

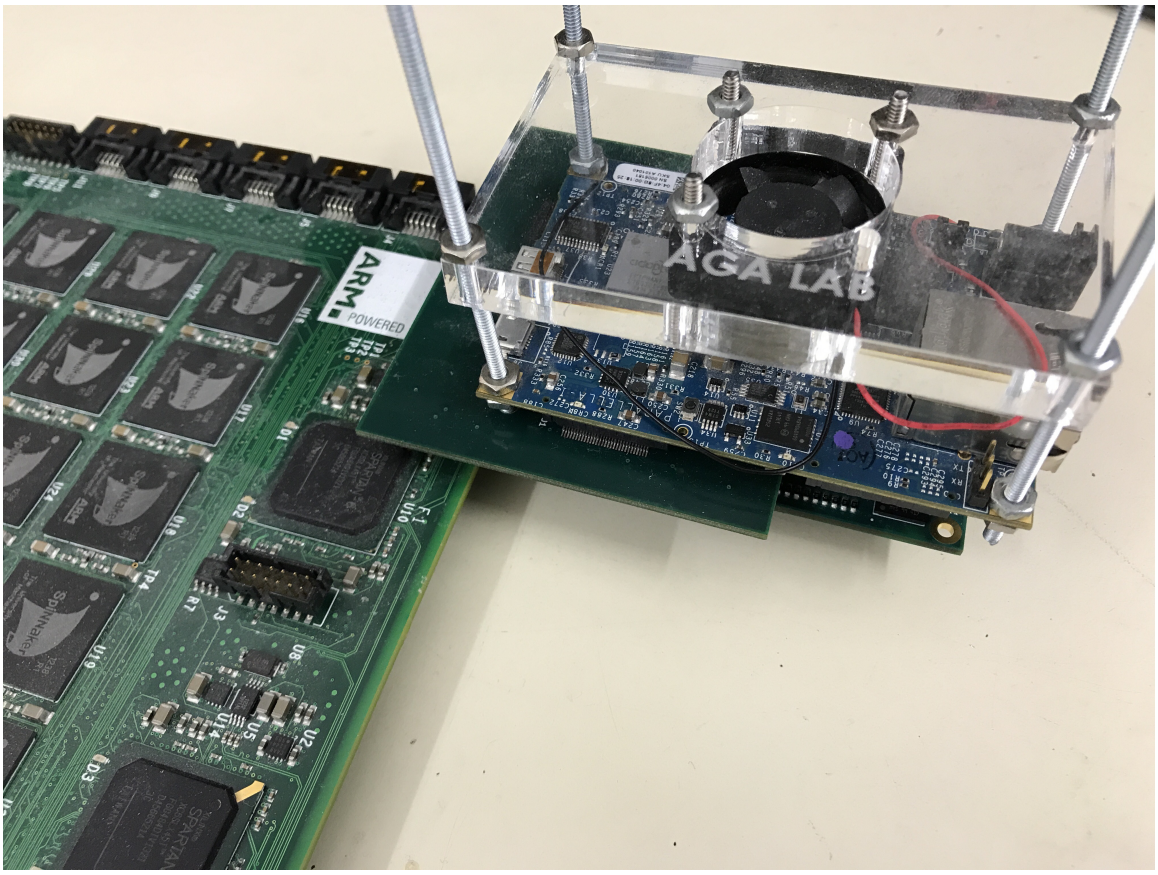


Figure 5.14: Close-up view of the heterogeneous architecture interconnect board connected to the SpiNNaker, Parallella, and Opal Kelly XEM6310 FPGA board.

Figure 5.14 shows a close-up view of the interconnect board connected to the SpiNNaker and the Parallella as well as an Opal Kelly XEM6310 FPGA board. The 48-node SpiNNaker is on the left and the Parallella is on top to the right. The FPGA

board is below the interconnect board, and the interconnect board sticks out on top of the SpiNNaker.

Both the SpiNNaker team and the Parallella team have made open source hardware description language (HDL) code available that creates designs that can communicate with their hardware, so those designs can be used with the FPGA on the interconnect board to transfer packets back and forth. These designs are currently being tested on the FPGA and the faster heterogeneous design is in progress.

Chapter 6

TrueNorth

IBM's TrueNorth Neurosynaptic System[29,30] is a brain-inspired, low-power parallel processing machine that is composed of neurons that are connected together (See Figure 6.1). The SpiNNaker contains general-purpose ARM CPUs and is therefore a more flexible architecture in terms of the classes of algorithms that can be practically implemented on it. However, the SpiNNaker consumes much more power than the TrueNorth because it must rely on simulating neuron behavior in software while the TrueNorth has built-in hardware neurons. Under typical utilization the TrueNorth chip consumes approximately 70 mW of power.

Despite being less flexible than the SpiNNaker there are a great deal of tasks the TrueNorth can be programmed to perform due to its programmable neurons and synaptic connections. The TrueNorth contains 4096 cores which each have 256 neurons and 256 axons that have a programmable crossbar array controlling their

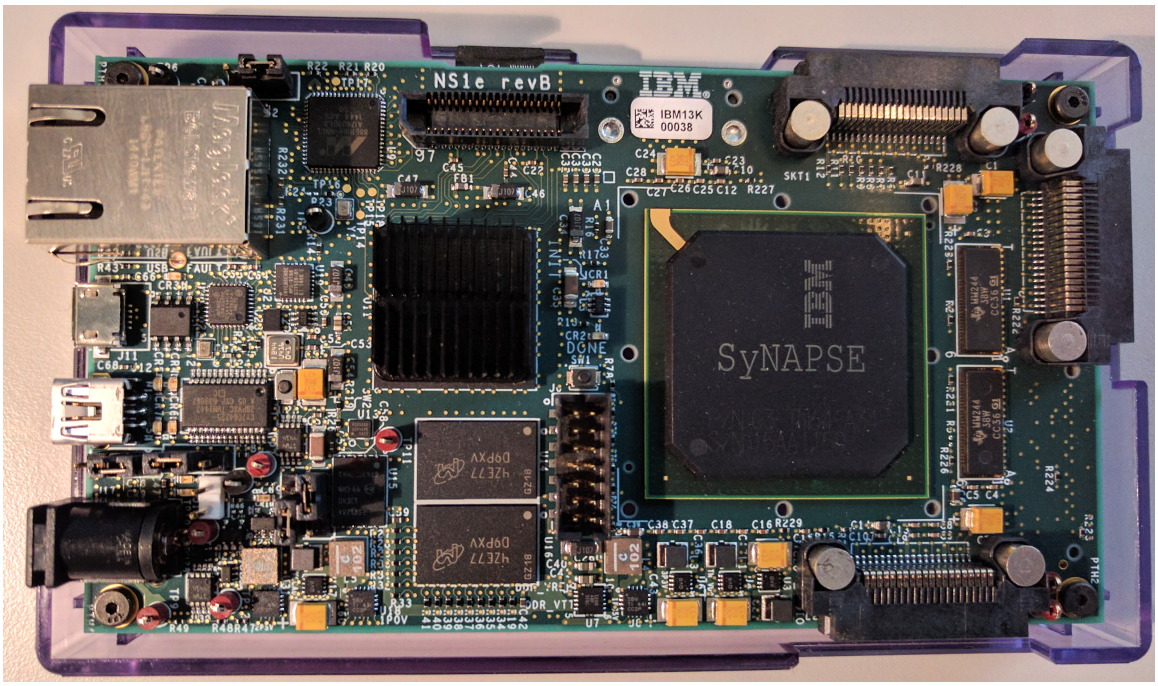


Figure 6.1: The IBM TrueNorth Neurosynaptic System.

synaptic connections. In total there are over 1 million individually-programmable neurons and over 268 million synaptic connections. These neurons have a flexible model, and each of their parameters can be custom-tuned including values such as their spike thresholds, leaks, and synaptic connection weights.

The TrueNorth chip itself is the large chip shown on the right of the board depicted in Figure 6.1. This TrueNorth board design is similar to that of the Parallella design described in Chapter 5 in that both designs utilize a Zynq-7000 series system on chip (SoC) to act as an intermediary between the host PC controlling the board and the TrueNorth/Epiphany chip contained within.

Code reuse is encouraged[65] by packaging a configuration of the TrueNorth as a "corelet," which allows other applications to use the same functionality and enable

hierarchical design by placing corelets inside of other corelets and being able to share them with the TrueNorth community. IBM has also made debugging convenient by creating the Compass simulator[66] so that a TrueNorth device does not have to be connected to the computer in order to work on a neural design.

IBM has also provided an energy-efficient deep neural networks (Eedn) software framework[67] that simplifies the process of programming the TrueNorth to run deep networks[68] without having to create a custom corelet from scratch each time. Instead, Eedn enables the user to specify the network layers and parameters, training data, development data, and later test data that are necessary for designing, training, and running the final version of the network. With these specifications the developer can tune the network on the training and development data, and the network can be ported over so that it runs directly on the TrueNorth hardware, providing a fast, low-power platform ready for mobile applications such as autonomous robots.[15] Training in Eedn takes into account the limitations of the TrueNorth architecture itself and learns trinary weights that can be ported right into the neurosynaptic system once training is complete.

This chapter focuses on some TrueNorth work that enables large-scale parallel vector-matrix multiplications (VMMs). An implementation with 4-bit precision as well as an 8-bit version are discussed as well as the tradeoffs between the two. An interesting natural language processing (NLP) application called Word2vec is also described along with some example results shown at the Telluride Neuromorphic

Engineering Workshop held during the summer of 2015. Finally, a way to perform stochastic multiplications on the TrueNorth platform is shown, and that implementation is used to perform nonuniformity correction (NUC) on images to show that this technique could be used in an image processing pipeline where the image originates straight from an image sensor. MATLAB simulations were also created to verify the design in addition to running directly on the TrueNorth itself.

Some of the work presented in this chapter has already been published[69] by the author of this thesis, particularly the Word2vec application described in Section 6.2.

6.1 4-bit Vector Matrix Multiplications

The TrueNorth hardware architecture contains 4096 cores which can all run computations in parallel, so one of the main goals was to perform the computations in a massively-parallel manner. Luckily, this problem is trivially decomposable into a parallel architecture by splitting the matrix, or dictionary, into smaller pieces that all compute the VMMs at the same time.

Each core in the TrueNorth hardware contains 256 neurons. However, as will be explained in Section 6.1.2, 4 neurons are required to represent each value in the matrix, so only 64 words from the matrix can be represented by each core.

The number of elements in the input vector (and the matrix elements) is 64. The math is all done as signed arithmetic, so there are actually 128 input channels to the

architecture. These 128 channels are split and sent to all the corelets.

The overall computation is the following:

$$y = x \cdot A, \tag{6.1}$$

where $y \in \mathbb{Z}^{1\text{x}N}$ is the result vector, $x \in \mathbb{Z}^{1\text{x}64}$ is the input vector, and $A \in \mathbb{Z}^{64\text{x}N}$ is the dictionary itself with N entries (each entry N is a column of the matrix). Because the computations are done with 4-bit accuracy the input values (x and A) must be within the range $[-8, 7]$. The output y can be in a larger range but that requires waiting longer for the resulting spikes to come out of the network because they are encoded as a simple rate code. In addition, this implementation only outputs positive VMM results because the neurons on the TrueNorth can only spike when their internal state hits a positive threshold. Clever tricks can be done to get output spikes for negative values, but since the application in mind is Word2vec negative values are not of interest. As explained in Section 6.2, finding the maximum values are the main objective, and Section 6.1.4 further explains these negative summations.

The number of corelets, M, required for implementing the math for the 4-bit VMM on TrueNorth is therefore:

$$M = \lceil N/64 \rceil . \tag{6.2}$$

Every corelet except for the last one is size 128x64, and the last one is size

128x[what is left]. The dictionary is sliced so that there are 64 columns in each chunk, and each of those chunks is processed in parallel in each of the main cores on the chip.

6.1.1 Main Corelet Architecture

Each corelet consists of two actual hardware cores. The first core consists of a dual-rail binary decomposition of the dictionary matrix assigned to the corelet, and the second core is used solely to add up the spikes coming in from the first corelet with the correct weights so the math works properly. These details are explained in Section 6.1.2 and Section 6.1.3.

The input vector, x, consists of 64 4-bit signed elements. In order to represent these numbers there are 128 input channels to the chip (and therefore 128 input channels to each corelet once the input is split and duplicated across all corelets in this architecture). Since there are 64 elements in the input vector, each element in the input vector goes to one of the two associated rails. The input goes to the positive rail if it is a positive number, and it goes to the negative rail if it is a negative number. The input is encoded as a regular rate code which means that the number of spikes corresponds to the value that is encoded. However, this could also be done as a stochastic rate if desired. Stochastic rate codes are discussed further in Section 6.3.

6.1.2 First Core

Let the portion of the dictionary represented by each corelet be denoted D_i where i is the index of the corelet ranging from 1 to M as described in Equation 6.2. The first physical core in each of these corelets represents D_i in a dual-rail binary decomposition manner.

Each set of two rows of the core's crossbar represents one row of D_i. The first of each of these two rows is the positive rail while the second is the negative rail, and that pattern continues the whole way down to the bottom of the matrix.

Each set of four columns of the crossbar represents one column of actual values in the original matrix D_i. Each of these four columns in the crossbar represents a different weight: $\{-8, 4, 2, 1\}$. The axon weights are all set to be 1 in the neuron model rather than the aforementioned weights, but each column still nevertheless corresponds to its respective binary weight. The correct weights will be taken into account and added together in the second core of the corelet.

So in total, each number in D_i takes up two rows and four columns of the crossbar. Let the current entry in D_i be denoted as k. k is a signed 4-bit integer ranging from -8 to 7 inclusive. The number must be encoded both on the positive and negative rails of the crossbar across the current four columns. For the positive rail, the connections are determined by performing a binary decomposition of the number. For the negative rail, the sign of the number is reversed and then the binary decomposition again determines the crossbar connections.

This dual-rail encoding works because if the input is positive, that positive rate encoding will come in on the positive rail and the binary decomposition is correct because it is multiplied by a positive number. On the other hand, the sign of the number on the negative rail must be reversed because it is multiplied by a negative number (if the entry in x is negative then the spikes come in on the negative rail). The dual-rail encoding takes into account the sign of the input spikes because the input spikes are simply a rate and cannot express whether they are positive or negative values. Instead the position of that rate (positive or negative rail) is used to denote the sign of the input value.

Every axon (input) in this core is of the same type (type 0 in this case), and all the neurons are also identical. Each neuron is a linear neuron with a threshold of 1 so that every time its state is positive it spikes once and reduces its state by 1. These neurons have no leak. Therefore, each column constantly sums up how many times a spike comes in for each of the values in that column of the dictionary, and the neuron outputs one spike corresponding to the weight of that column of the crossbar ($\{-8, 4, 2, 1\}$). Section 6.1.3 describes the straightforward manner in which the second physical core in each corelet converts these spikes into their proper number of output spikes to produce the results.

6.1.3 Second Core

The second core is used to sum the results of the multiplication from the first core. While the weights in the first core are all one, each column is designated to be a certain weight. Therefore, these columns are summed in the second core with their corresponding weights, and the second core outputs the final results.

The neurons in the first core are grouped into sets of four columns, and these sets of four neurons are connected to the axons in the second core in the same order. Therefore, in the second core, each set of four axons is a group. Within each set of four axons there are four types: $\{0, 1, 2, 3\}$, and these types tell the neurons which weights should be added to the neuron states when spikes come in.

The neurons in this second core are all set up so that axons of type 0 have weight -8, type 1 have 4, type 2 have 2, and type 3 have 1.

The crossbar is set up so that neurons 1 to 64 in the second core each correspond to the column sum for each of the 64 words in the dictionary (of course in the last corelet there may be less than 64). In order to do this, each set of four axons must be connected to a separate neuron. The first neuron has connections with the first four axons, the second neuron has connections with axons 5-8, and so on.

The result of all these connections is that each neuron outputs the proper summation from the dot product of the input vector with its corresponding column in the matrix. Since the input is a rate code and the connections all add up to one value of the dictionary VMM multiplication, the outputs are all rate-coded as well.

6.1.4 Negative Summations

Each neuron can only be configured to spike when its state hits a positive threshold value, so results for when the summation in a column of the calculation is positive work correctly. On the other hand, when the result for an addition in a column is negative, no spikes are output.

A different architecture could be constructed so that negative results are also output, but it would be less space-efficient. This example alternative architecture could reverse the sign of the math in each column so that there would be two output neurons for each word in the dictionary. One would spike when the result is positive and the other would spike when the result is negative. However, this type of architecture was not implemented because in the Word2vec application the maximum output values are sought and negative values are not very useful (See Section 6.2).

6.1.5 4-bit Unsigned VMM

In addition to the signed architecture described thus far is an implementation of an unsigned version. This version is very similar except that it of course only works with positive values. The unsigned design additionally allows for vectors of length 256 rather than 128. Since the values are all positive the second input row is not useful for the input values. The matrix is encoded using only the positive binary decomposition as well because nothing needs to be done to take care of negative

input. The number of outputs per corelet remains the same because there are still 4 bits in the computations and therefore 4 bits that must be accounted for via the axon types. These four axon types are still reflected in the four outputs per column going into the second core of the corelet and therefore allow for only 64 output columns per corelet.

6.2 Word2vec

Word2vec was developed at Google by Tomas Mikolov and other researchers,[70,71] and it provides ways to take a large amount of text data and create a vocabulary of words represented in a high dimensional vector space. Further analysis[72] was performed to show that Word2vec works well when compared to other methods that accomplish the same task. Although this general concept has been around for a while[73] and has been improved upon since,[74] in this thesis the implementation on Google Code was used.[1]

These so-called "word embeddings" are trained so that, generally, words that are similar semantically and/or syntactically are closer together in vector space and words that are not similar are farther apart in the vector space. Since one similarity metric for vectors is the dot product, the aforementioned VMM framework on the TrueNorth can be used to detect word similarities given a dictionary of word vectors trained

[1] https://code.google.com/p/word2vec/, accessed in 2015. However, a more current version can be found at https://github.com/dav/word2vec as the old version is now unavailable.

using the Word2vec framework. The VMM essentially can be used to perform many thousands of dot products simultaneously using spiking neurons while consuming very small amounts of power. Therefore, if the input vector to the VMM is the vector corresponding to the word of interest and the matrix is the dictionary of word vectors corresponding to the vocabulary, the maximum results in the VMM indicate the words that are most similar to the input word in the trained model.

6.2.1 Background

Word2vec essentially works by trying to predict which words are near other words in any given text. Given a corpus of length T containing words $w_1, w_2, ..., w_T$, the "Skip-gram" model maximizes[71]

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c<j\leq c, j\neq 0}\log \mathrm{P}(w_{t+j}|w_t), \tag{6.3}$$

where c determines the size of the window, or context, of words surrounding the current word. The window size trades off training complexity versus accuracy.

The probability in Expression 6.3 is nominally[71] the softmax function:

$$\mathrm{P}(w_O|w_I) = \frac{\exp(v'^{T}_{w_O} v_{w_I})}{\sum_{w=1}^{W}\exp(v'^{T}_{w} v_{w_I})}, \tag{6.4}$$

where the v terms correspond to the vector representations of the w words in Expres-

sion 6.3 and v_w is the input representation while v'_w is the output representation. W is the number of words in the dictionary. Expression 6.4 clearly favors maximizing the dot product of the vector representations of nearby words in text which should make words that are used in similar situations be closer in vector space. However, when maximizing Expression 6.3, the gradient of Expression 6.4 is taken, and calculating that is usually computationally expensive due to the large size of W.

The model is physically expressed as a neural network with an input layer, hidden layer, and output layer.[70] The input consists of T nodes, where as expressed earlier, there are T words in the dictionary. The hidden layer is a linear layer, and the number of nodes in this layer is the length of the vector representation of the words. Finally, the hidden layer is fully connected to the output layer which computes the softmax function.

The input for the Skip-gram model is a one-hot encoding of the word, which means that all the input values are 0 except for the input corresponding to the current training example which is 1. During training the output layer becomes a similar one-hot encoding to train the network to represent the input word. The context window is taken into account by keeping the input fixed and changing the output one-hot encoding to represent the other words in the context window[70] before moving on to another input training word. This choice of words within the context window is done by performing sampling so that the words farther away from the current word are chosen less likely. One way this is done[70] is by randomly selecting a value from 1 to

the size of the context and using that value as the context for that training iteration.

Since the input is a one-hot vector and the hidden layer is a simple linear layer, the weights for the connections going from the input layer to the hidden layer actually encode the vector representation of the word. This vector representation is then fed into the softmax layer for the output.

On the other hand, an alternative to the Skip-gram model is the Bag-of-Words model. The Bag-of-Words model essentially reverses the Skip-gram model by making the input be various words in the context and the output be the current word.[70] Each technique is slightly different but can be better suited to particular tasks.

Improvements have been made to this basic formulation to speed up training[71] by approximating the softmax calculations. Another useful change is to either ignore common words or to subsample them by discarding each word with a probability value related to the frequency with which the word shows up in the corpus, both of which accomplish the goal of ignoring spurious similarities with common words such as "the."

Once the model is trained it is trivial to find similar words. A chosen word is converted to its vector representation by finding it in the dictionary. Then similar words can be determined by performing the dot product of every word's vector representation in the dictionary with the query word's vector representation. The words with the largest dot product values are the words most similar in the Word2vec framework.

6.2.2 Word2vec Word Similarities on TrueNorth

These dot products take time to compute, and a way to perform them in parallel is to perform the equivalent VMM. Taking that concept one step further, the 4-bit VMM architecture on TrueNorth (Section 6.1) can be used to calculate word similarities in a low-power, neuromorphic manner. The bulk of the work done to establish the main TrueNorth results was accomplished at the 2015 Telluride Neuromorphic Engineering Workshop.

The first step was to use the Google Code implementation of Word2vec to create a large dictionary. All the text from Wikipedia was downloaded and fed into Word2vec. The number of hidden layers (and thus the length of the word vector representations) in the Word2vec model created was 64. In addition, once the model was trained, the computations done with the 4-bit VMM architecture on TrueNorth are limited to 4 signed bits, so the dictionary was quantized to fit into that 4-bit range.

MATLAB code was written to automatically convert a given dictionary into a corelet required for programming the TrueNorth and to also communicate with the TrueNorth to send the spiking input into the programmed design. Code was also written to receive the results back from the TrueNorth and interpret them, including determining which words in the dictionary correspond to the largest VMM entries and thus the most similar words in the dictionary. Each of these steps was made possible by the use of IBM's APIs which provide the interface for programming the hardware.

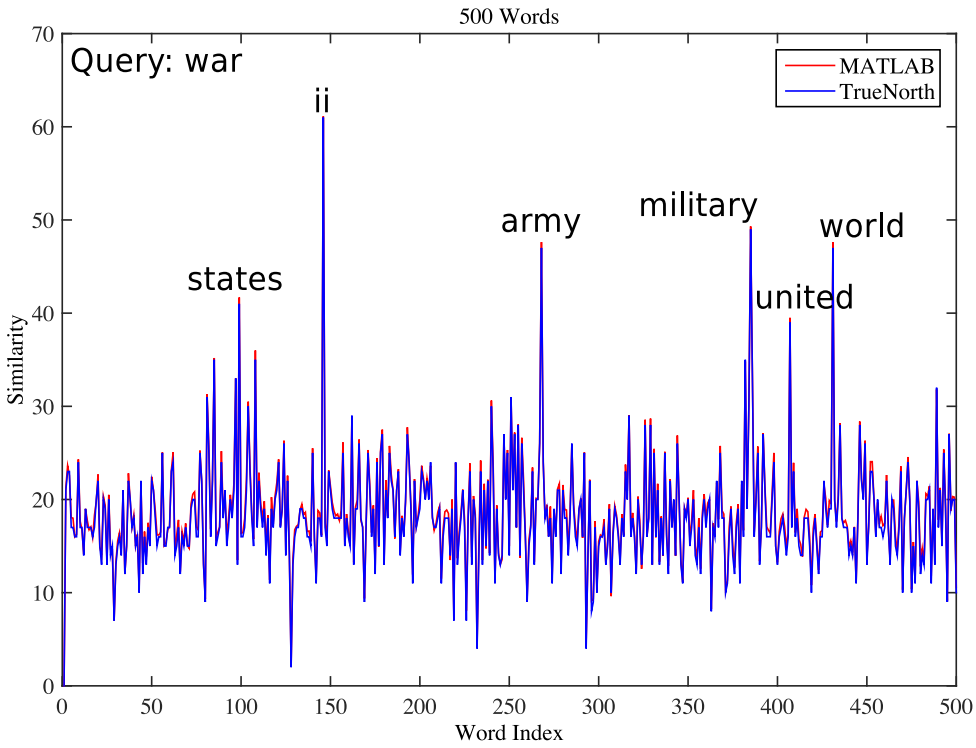


Figure 6.2: Similarity values for a 500-word dictionary trained on Wikipedia.

A simple experiment was conducted using the 500 most-commonly found words in the Wikipedia dictionary which can be seen in Figure 6.2. Here the query was the word "war," and both the query and some of the most similar words are labeled. VMM results from both the TrueNorth and MATLAB are shown. The reason why the results are not identical is because of the magnitude of the output results. Despite the input vector and the dictionary vectors all being quantized to be signed 4-bit numbers the dot product results were very large. Therefore, in order to speed up the simulation on the TrueNorth so that less output spikes had to be gathered, the results were effectively divided by 16 by setting the neuron threshold in the first core of each corelet to be 2 instead of 1 and setting the threshold in the second core to 8 instead of 1. These thresholds clearly do not perform actual perfect division, so in actuality there is a bit of quantization error as compared to dividing by 16 with floating-point numbers in MATLAB which is shown in the plot. This error does not drastically influence determining the words of maximum similarity, though, which is the main goal of this architecture. In addition, these thresholds can be chosen for future applications by trading accuracy vs. speed depending on the goal.

The final demo in Telluride used a much larger dictionary of the 95,000 most-commonly used words in Wikipedia. All told this design utilized 3,991 TrueNorth cores including splitter cores used to route the input spikes to all the smaller VMM blocks being calculated simultaneously. These cores account for approximately 97.4 percent of the total possible TrueNorth chip utilization. The computations were all

telluride	**basketball**	**mountains**	**neuron**
carbon	volleyball	landforms	electron
colorado	handball	ranges	protein
copper	soccer	glacier	tissue
springs	ncaa	plateau	cells

Table 6.1: Similarity results for four queries running on the TrueNorth hardware with a dictionary of 95,000 words trained on Wikipedia.

completed within 153 ms due to the parallel nature of the combined software/hardware architecture, although receiving spikes off the board took more time. In addition to altering the neuron thresholds, this large dictionary example incorporated a neuron leak in order to further reduce the number of spikes coming out of the system. The leak introduces more error than the threshold changes alone but does not significantly affect which dictionary vectors end up as the most similar to the input vector.

Table 6.1 shows the top results for a few queries run on the TrueNorth hardware with a dictionary of 95,000 words trained on Wikipedia data. These top similarity results match the results from a Python VMM implementation used at the Telluride workshop.

Another interesting task that can be accomplished using these word vectors is completing analogies.[70–72] For instance, one example[72] is that the vector between the representation of "man" and "woman" is similar to that between "king" and "queen." Therefore, to solve the analogy "man is to woman as king is to ..." the input to the VMM can be the vector math corresponding to woman - man + king. The results of that analogy and a few others are listed below:

- man is to woman as king is to: **queen**
- man is to woman as uncle is to: **aunt**
- good is to better as bad is to: **worse**
- france is to wine as germany is to: **beer**
- apple is to iphone as google is to: **search**

Many analogies do not work well with the Word2vec framework, but it is interesting nonetheless that relationships like these can be found by simply looking at which words occur together in text using a neural network.

6.3 Stochastic Multiplications with Column Select

The binary decomposition technique described above is useful because it allows for exact computations to be performed. However, only 64 columns of the matrix can be represented at once, and an entire second core must be used to perform the summations to complete the process, so the method is not very space-efficient.

On the other hand, a different technique called stochastic multiplication with column select is more space-efficient if the task is to multiply a number times a

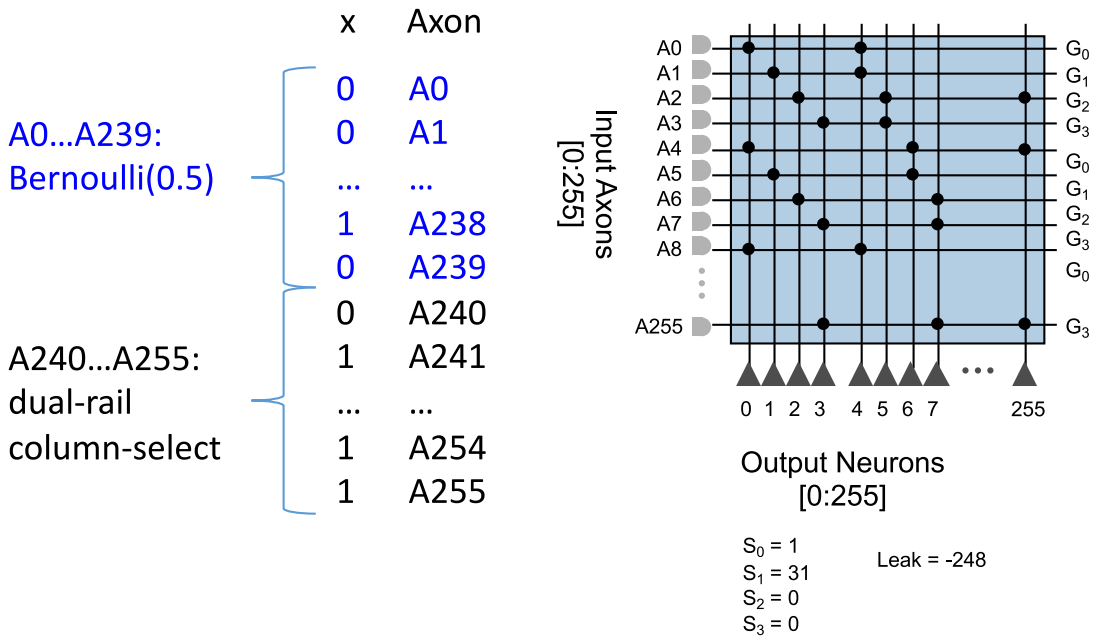


Figure 6.3: Stochastic multiplications with column select.

vector. Only one core in total is required, and the input number can be multiplied by a vector of 256 different elements.

Figure 6.3 shows an overview of how the core works. The input is a scalar value represented by a 256-element vector connected to the axons of the core. The first 240 elements represent the input number as a stochastic rate,[75,76] and the last 16 elements are a dual-rail encoded column address that must match the column address of the number in the crossbar the input should be multiplied by.

The crossbar is organized so that each column represents a separate number. The first 240 connections in the crossbar for that column are encoded as a stochastic rate, and the last 16 elements are the dual-rail encoded column address.

The dual-rail column address is divided into 8 positive rails and 8 negative rails. The positive rails consist of the binary decomposition of the column address, and the negative rails have a connection wherever the positive rail does not have a connection. Therefore, in each column there are 8 total connections in the crossbar for the column address.

The axon types for the first 240 axons are all identical. These axons are used to represent a stochastic rate, so they are all just summed when input spikes come in. The axon types for the column addresses are all identical as well, but they are a different type. The weight for this type is set to be 31, and the leak on the neurons is set to be 31*8 = 248 (it is negative).

The weights work because all 8 crossbar connections in the column address must match in order to cancel out the neuron leak. Since the neurons only spike when their states reach a positive value, the columns that do not match the input column do not spike. However, this of course only works if the density of the crossbar connections representing the elements of the vector is low enough. For example, if one column is only off by one address bit, then there can only be a maximum of 31 crossbar connections above the column address. Otherwise that other column can also spike. Note that these addresses do not limit the computations to only target one column at a time. Instead the addresses are flexible and can be reused, so multiple columns may be selected at once. This technique provides a high degree of spatial flexibility when designing cores because it is possible to choose sections of each core to use for

computations while ignoring other portions of the crossbar.

The architecture described thus far can only work once and must somehow be reset in order to perform another computation. The neurons do not have an innate ability for their membrane potentials to be reset, so once a subset of columns is selected using its address those columns are not going to immediately be reset back down to the value of the neuron leak. In order to improve the design and to correct this issue, Kaitlin Fair from the Air Force Research Laboratory suggested reserving a row of the crossbar for resetting all the neurons. This row has a third axon type in addition to the stochastic bit type and the address type so that the neuron reacts properly to the input and performs a reset. The weight for this axon type at each neuron must be more negative than the negative neuron threshold. Each neuron must also be configured to have a negative saturation at its threshold. This setup provides the opportunity to send a reset signal using the reset axon which makes all the neurons go back to their negative threshold states and prepare the neurons for the next computation.

Since the stochastic column-select multiplications can only select one column to multiply the input by each time, there is a space-time tradeoff at play. In order to multiply the input value by each element of the vector represented by the crossbar, each multiplication must be done one at a time. So the input comes in on the axons once, the output must propagate through the network of spiking neurons, and then the column address on the input must be changed to the next value before the process

can repeat. However, this type of architecture uses less space than the 4-bit VMM architecture. Of course the 4-bit VMM achieves a different goal as well since it multiplies a vector by a matrix instead of a value by a vector as is done here.

In addition, the limitation of having 31 or less crossbar connections above the column address in each column limits the resolution of the numbers to be represented in the core. However, tricks can be done to increase the resolution by, for example, making two columns have the same address and treating them both as separate sections of the same stochastic rate code. Then the outputs of the two neurons attached to each column can be added together in order to create more resolution (more potential time slots for spikes to occur). This technique can be extended to more columns for more resolution, but adding more columns requires more space and/or time depending on whether the addition of the columns is handled in the same core in another column or in another core. Adding an extra column for addition in the core would likely also require using less than 31 rows for the stochastic stream to allow for some of the rows to be used exclusively by the addition column.

Another change to this architecture is that instead of multiplying one number by a vector, a VMM computation can be done instead. Here the 240 stochastic bits can be split up into K different sections, each one representing a different number. This split of the 240 bits is done both to the input and to each column of the crossbar. That way there can be multiple input values and multiple values per column, and they are all multiplied and added automatically. Of course, the limit of a maximum

of 31 crossbar connections per column must still be enforced, though. If the number of bits used to store the neuron leak were increased then the leak could be larger than 255 (it is currently 248), and then the weight on each column address bit could be larger. This larger weight would give more room to allow more stochastic rate spikes to be in each column of the crossbar.

There are many potential tweaks and avenues of exploration with this base architecture for performing stochastic multiplications using the TrueNorth crossbar arrays. It is also an interesting way to select a given column without wasting space that could otherwise be used by other neurons with that input axon. The trivial way of selecting a column would be to only have one crossbar connection in a given row so that the input from that axon only goes to a given neuron. However, with this technique other columns can have connections in that row but still will not output anything when they are not selected as the column of interest.

Simple experiments were conducted to test the architecture using stochastic multiplications, and once they were completed further work was done to show that non-uniformity correction (NUC) math can be done to correct for noise in image sensors using this design. See Section 6.4.2 for more details on those results.

6.4 MATLAB Simulations

In addition to running the corelets on the official IBM TrueNorth simulator and directly on the hardware itself, separate MATLAB simulations were created to simplify the process for particular problems. One simulation covered the Word2Vec functionality, and the second simulation covered the stochastic multiplications with column-select. Both simulations involve the creation of classes designed to emulate the behavior of various parts of the TrueNorth architecture including all the spikes and internal neuron states, etc. This type of access also simplifies the debugging process when creating a corelet because this simulator provides access to everything whereas the IBM simulator does not. However, the way this simulator is programmed is different than IBM's Compass simulator, so it requires describing the corelet in a different way than is done using the official API.

6.4.1 Word2vec

A simulation was designed to emulate the Word2vec implementation on the TrueNorth hardware. The simulation loads the dictionary, chooses the query, creates the input spikes, creates the simulation, and loops through all the input spike steps to run the simulation and create the output spikes. These output spikes are collected and analyzed by comparing the accuracy of the distributed VMM computation with doing the math directly in MATLAB, and exact results were found with dictionary sizes up

to 95,000 words. Note that the 95,000 word dictionary took a long time to run (it was run overnight).

Another point to note is that the number of simulation ticks matters. Each neuron's threshold is set to be 1 in the MATLAB simulations, so for every increment in the output value for a word similarity that corresponding neuron must spike once. If the simulation does not run long enough some of those spikes will never occur, so the simulation must be run long enough to gather all the output. 5000 ticks was adequate for the 95,000 word dictionary with a query on the word "war."

Figure 6.4 and Figure 6.5 show the results for a small 500-word dictionary when comparing exact math in MATLAB with the TrueNorth simulations written in MATLAB. Figure 6.4 shows the results when the thresholds on all the neurons are 1 and no quantization error occurs. On the other hand, Figure 6.5 shows the results when the threshold for core 1 is 2 and the threshold for core 2 is 8. In that situation the same quantization error is introduced as was created when running on the TrueNorth hardware itself. Both of these figures can be compared to Figure 6.2 which shows the results of running the Word2vec similarity query on the TrueNorth hardware itself.

6.4.2 Nonuniformity Correction

Image sensor arrays consist of pixels that do not all react exactly the same way to the same light intensity that arrives at the sensor. These nonuniformities, often called fixed pattern noise, consist of two main concepts.[77,78] The first is that there

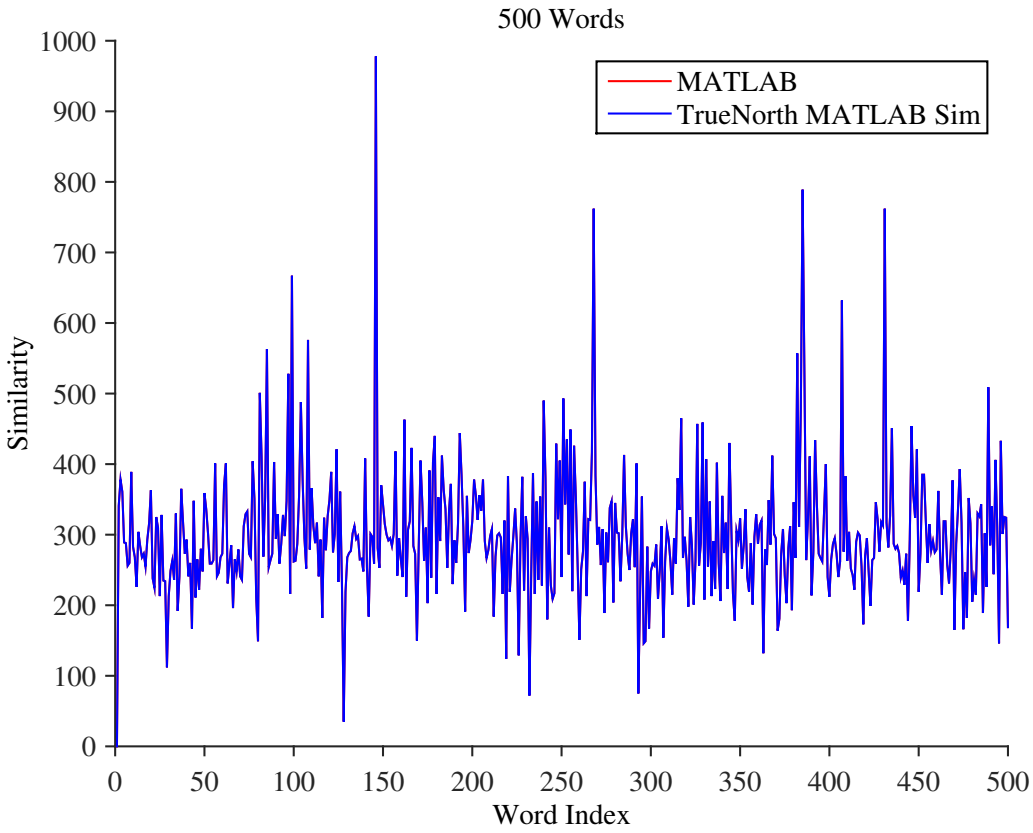


Figure 6.4: Exact similarities versus a MATLAB simulation of the TrueNorth neuron model with a 500 word dictionary trained on Wikipedia. This version has neuron thresholds of 1 for each of the two cores in the corelets, and the math matches exactly.

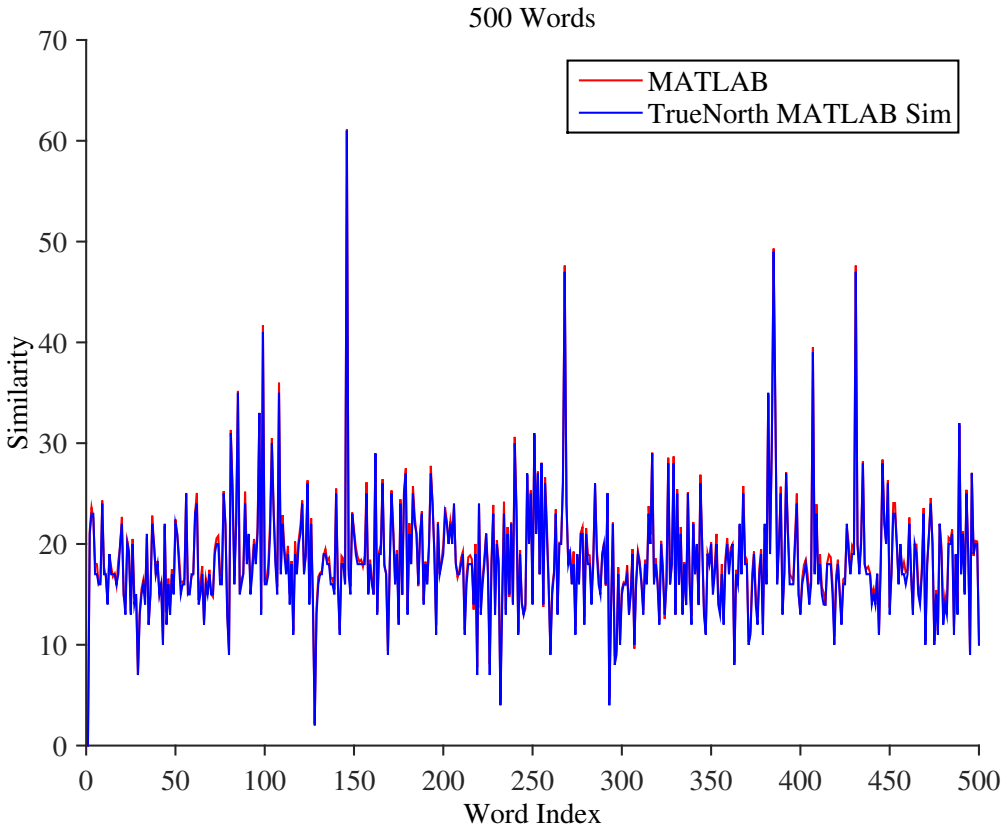


Figure 6.5: Exact similarity values (divided by 16) versus a MATLAB simulation of the TrueNorth neuron model with a 500 word dictionary trained on Wikipedia. This version has neuron thresholds of 2 and 8 for cores 1 and 2 in each corelet, respectively, so that the number of output spikes is reduced by a factor of 16. As a result, there is some quantization noise in the number of output spikes, and this noise matches the results from the real TrueNorth implementation.

is a fixed offset response for each pixel i when no light hits the sensor. The second is that each pixel has a different gain which controls how sensitive the pixel is to changes in light intensity. Expressed another way,

$$x_i = \frac{1}{g_i} \cdot y_i + o_i, \tag{6.5}$$

where x_i is the pixel output, $1/g_i$ is the gain of the pixel, o_i is the offset value, and y_i is the actual light intensity. Therefore, the following operation can be done to correct the output x_i and arrive at the value of interest, y_i:

$$y_i = (x_i - o_i) \cdot g_i. \tag{6.6}$$

The process of fixing the nonuniformities in the array is called nonuniformity correction (NUC). Since an image consists of an array of pixels the pixels can be arranged to form a vector of pixels. Then the stochastic multiplication with column select architecture can be used to perform the multiplication portion of the nonuniformity correction. Therefore, MATLAB code was developed to create random offsets and gains for each pixel of an image. All of the pixels were quantized to the number of bits used, and the gains and offsets were also quantized. Then, the NUC operation was done four different ways. In each of these ways the subtraction was done in MATLAB to simplify the process and just focus on the multiplication for now, so the stochastic NUC operation is only applying a gain multiplication.

The first way, called "NUC MATLAB," is done using floating-point math on the quantized values described above. Then the result is also quantized. The second way is "NUC Stochastic MATLAB," which means that the numbers are encoded stochastically using a random number generator in MATLAB and multiplied stochastically (bitwise) using MATLAB. Using the random number generator means that with an infinite stream of bits the number is encoded perfectly, but with a limited stream of bits the encoding depends on the random sampling that occurs. Then the results are decoded and quantized back to the specified number of bits.

The third way, called "NUC Stochastic Correct MATLAB," is done the same way as the third technique except that the number of 1's and 0's is fixed to be as correct as possible (down to quantization error corresponding to the length of the bit stream). So instead of randomly sampling each bit from the correct probability distribution, the ratio of 1's and 0's is fixed to approximate the number as well as possible given the limited amount of bits in the stream. Then the position of these bits is permuted and the same stochastic math is performed. Because the values are guaranteed to be as accurate as possible using this third method it is expected to yield better results than the second method.

Finally, the technique called "NUC TrueNorth Stochastic" emulates the straightforward stochastic technique (NUC Stochastic MATLAB, which is the second technique) but runs on the MATLAB TrueNorth simulator designed in this thesis by using the stochastic core.

There are various parameters for the simulation. The simulation loads in a large image of size 1440x2560, but the user can choose the number of rows and columns to process. In addition, the number of bits with which to quantize all the results and intermediate values can be chosen. Finally, the NUC operation can be averaged over a chosen number of trials to improve the estimates of the real values utilizing the law of large numbers.

All of the simulations shown in Figure 6.6 through Figure 6.13 were done with only a single column of 50 values, so the number of rows is 50 and the number of columns is 1. The number of bits for quantization in each of these simulations is 6. However, the number of times the operations were done so they can be averaged varies from 1 to 150.

The layout of the images in each figure is the same. The first row contains the original image in MATLAB which again consists of one 1x50 column of pixels from the upper-left hand part of the image. The right image is the "raw" quantized image created by fabricating gain and offset values for each pixel and applying them so there are nonuniformities present.

The second row contains the results of performing NUC using floating-point math on the quantized values. The result is quantized, and that is what leads to the absolute differences seen on the right. At most the differences are 1 because exact math is done here here and there is only quantization error to deal with.

The third row contains the NUC done stochastically in MATLAB along with the

absolute differences to the original image. The fourth shows the "correct" stochastic computations in MATLAB as described earlier along with the absolute differences. Again, note that the "correct" version is expected to be more accurate than the regular stochastic technique due to sampling a finite number of bits in the streams representing the numbers. Finally, the last row shows the results of performing the stochastic NUC using the TrueNorth simulation framework created here.

Figure 6.6 through Figure 6.13 show that it takes a good number of iterations before the performance of the stochastic NUC approaches that of the floating-point MATLAB version. In addition, all of the stochastic techniques show similar results (MATLAB, "correct" MATLAB, and the TrueNorth simulation version).

The MATLAB TrueNorth simulation code is slow to run, so that is why these characterization experiments were done, and they provide an idea about roughly how many trials must be run each time and averaged in order to obtain various levels of performance. Then the algorithms can be run later on a larger image patch without having to try out so many different parameter values.

Figure 6.14 through Figure 6.18 show results for 100 averaging trials and 6 bit quantization on image patches rather than just a single column. The size of each image patch ranges from 64x64 up to 256x256 for the last figure. These results show that the TrueNorth simulation works approximately as well as the stochastic MATLAB simulations and the "correct" MATLAB stochastic simulations.

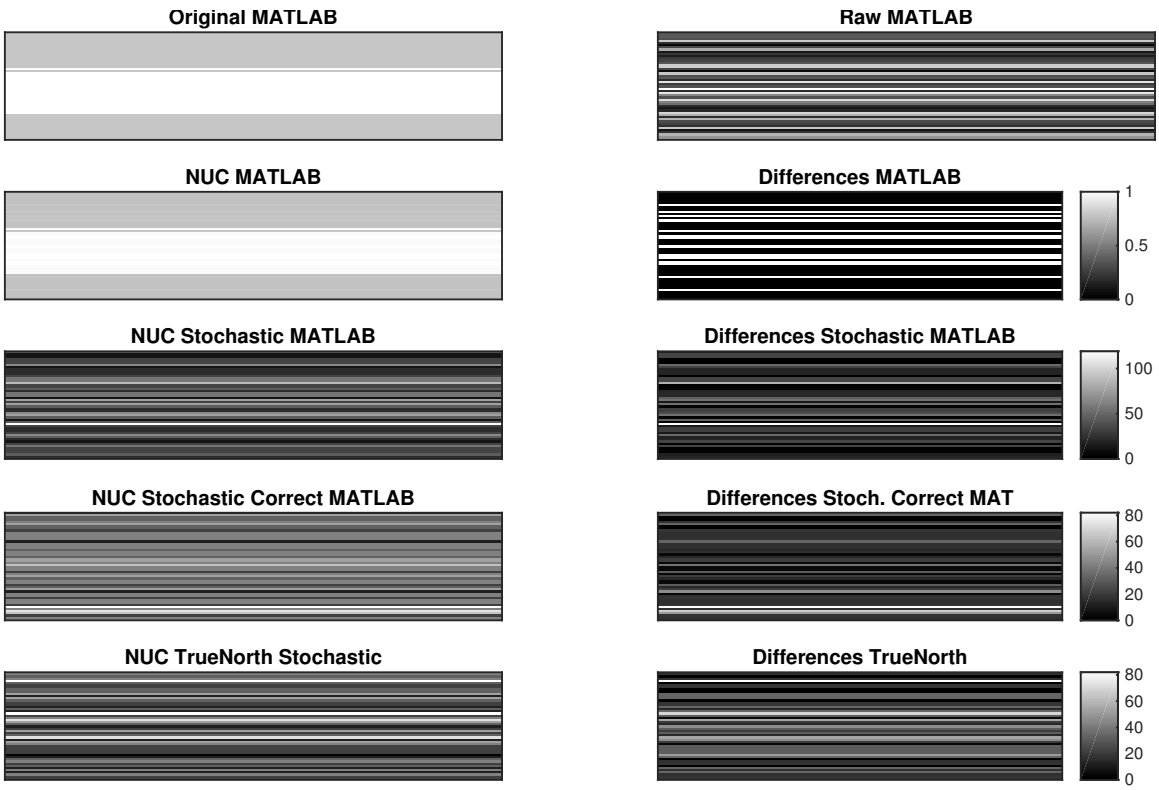


Figure 6.6: 1 averaging trial done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 50 rows, and 1 column.

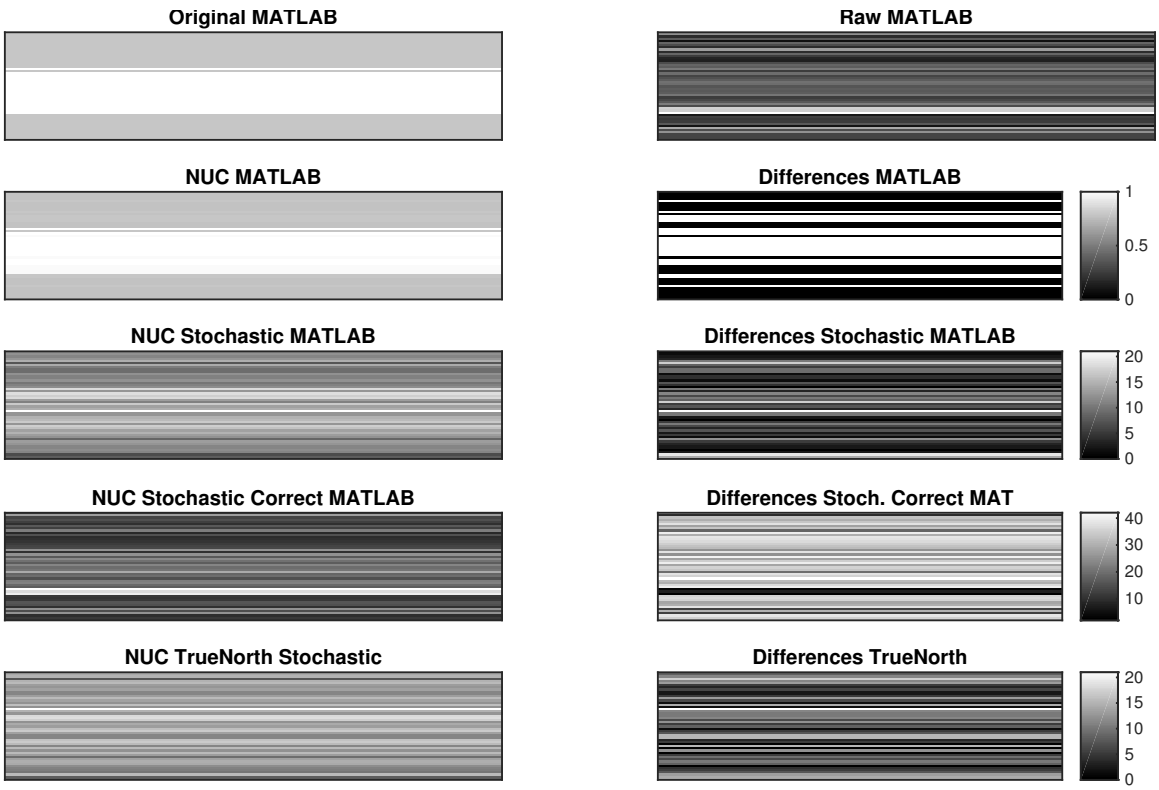


Figure 6.7: 10 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 50 rows, and 1 column.

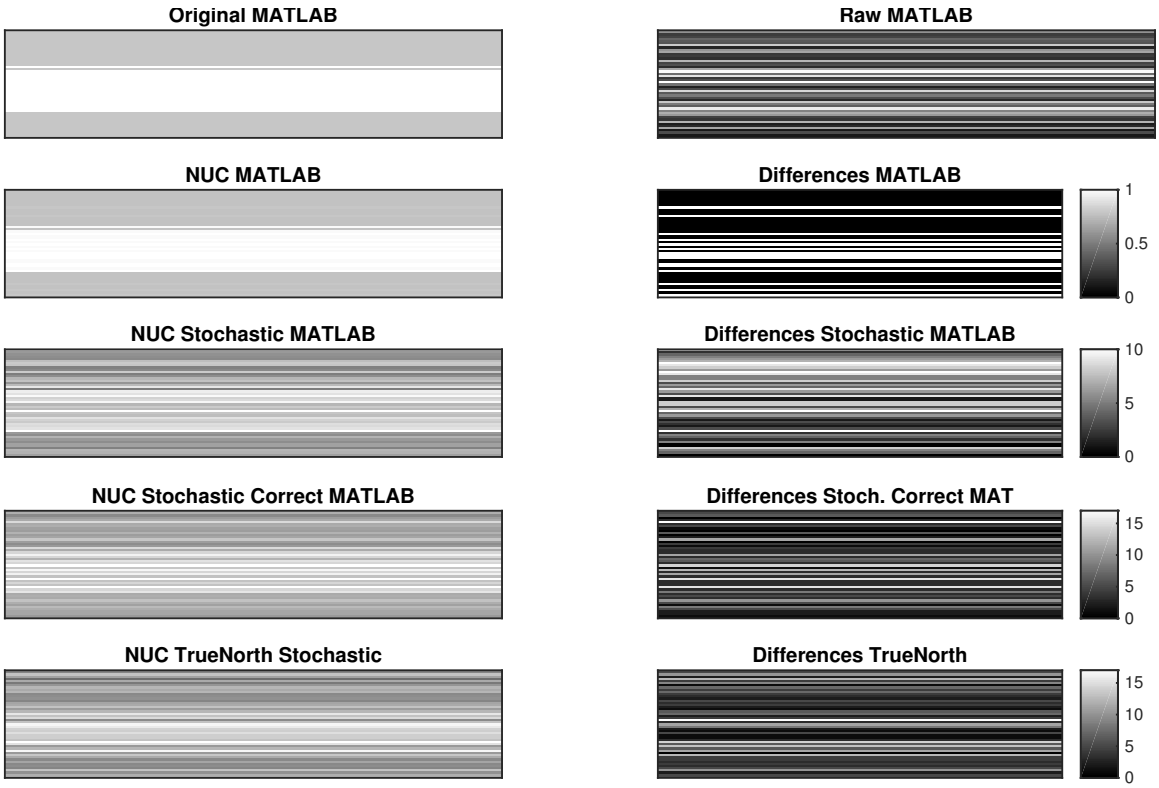


Figure 6.8: 20 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 50 rows, and 1 column.

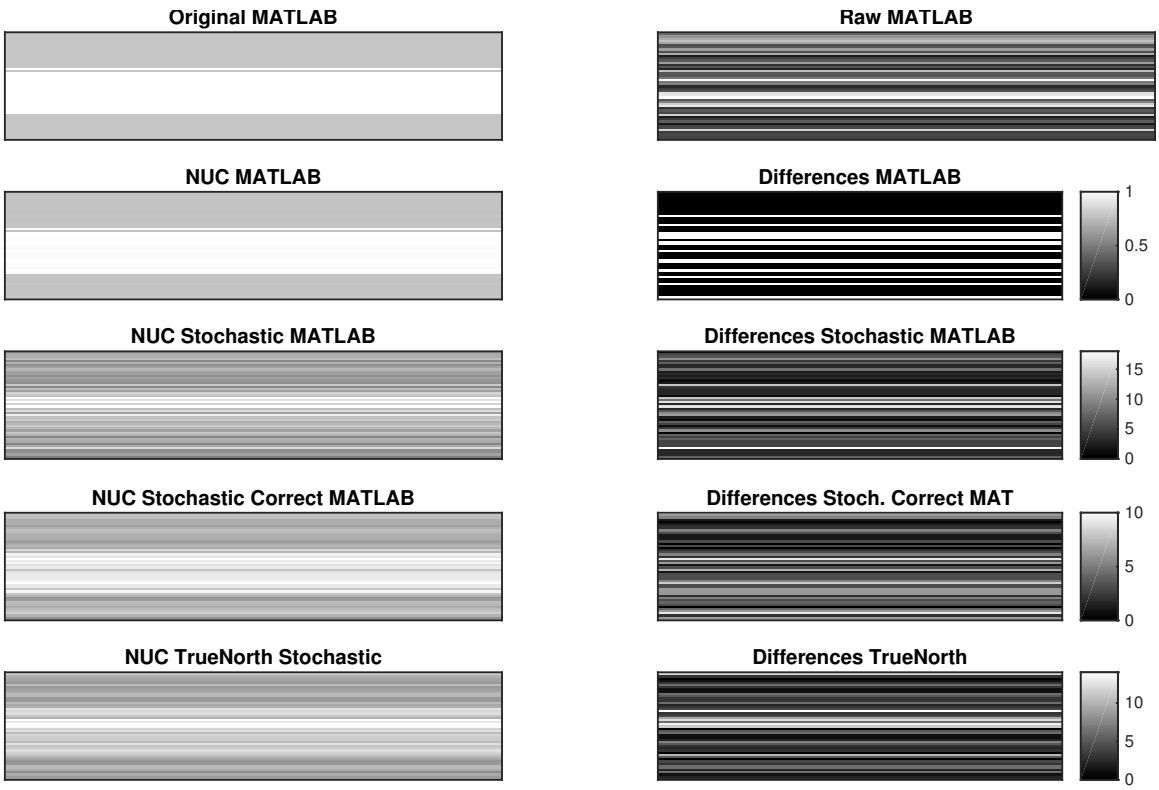


Figure 6.9: 30 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 50 rows, and 1 column.

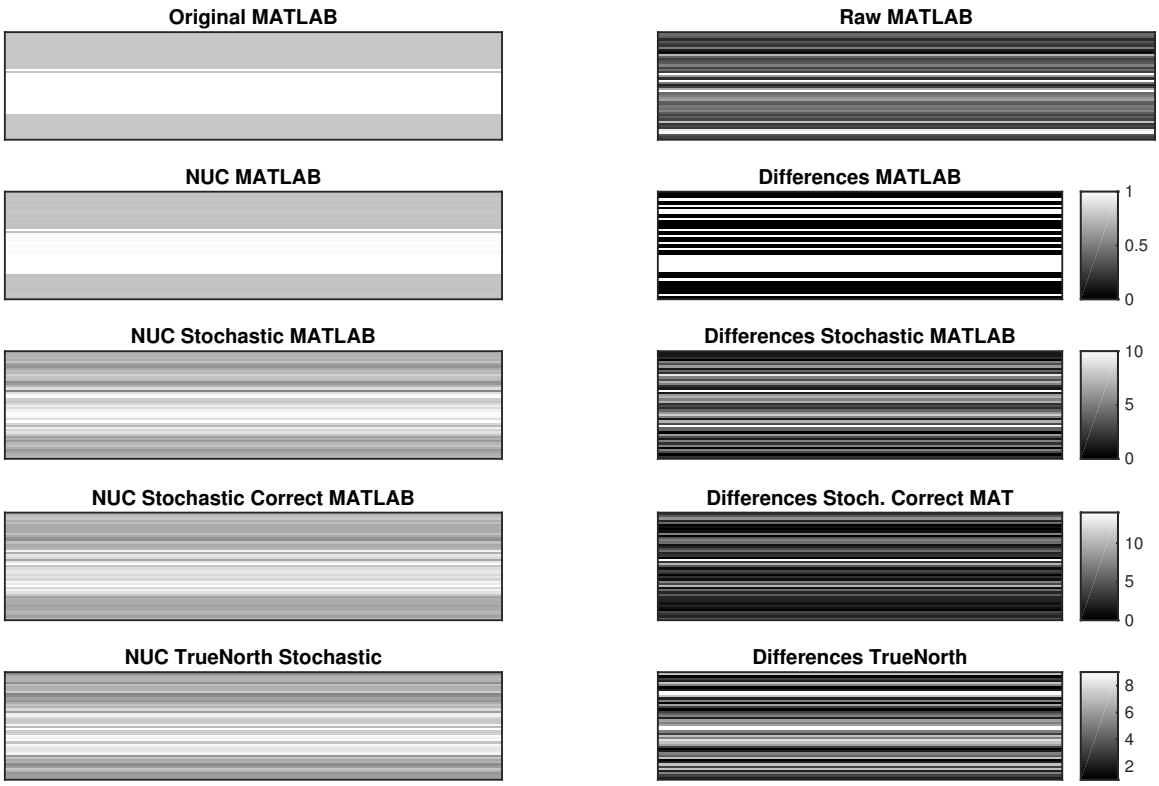


Figure 6.10: 50 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 50 rows, and 1 column.

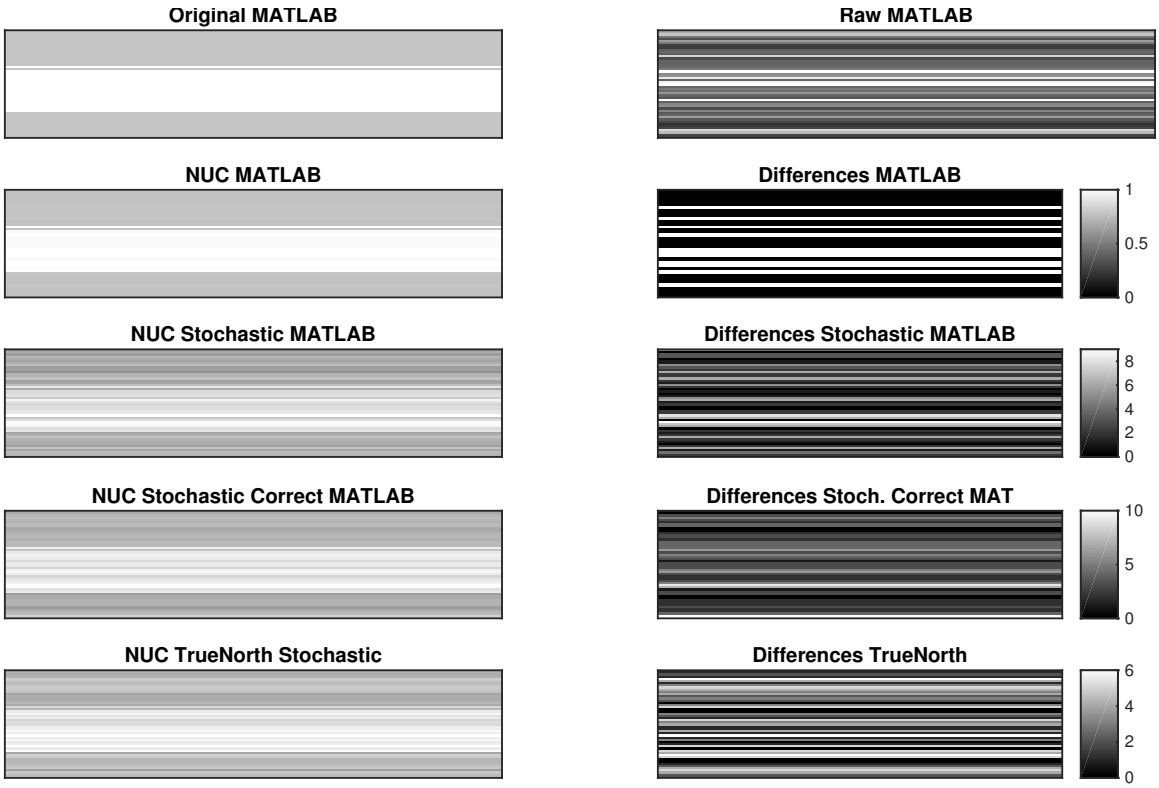


Figure 6.11: 75 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 50 rows, and 1 column.

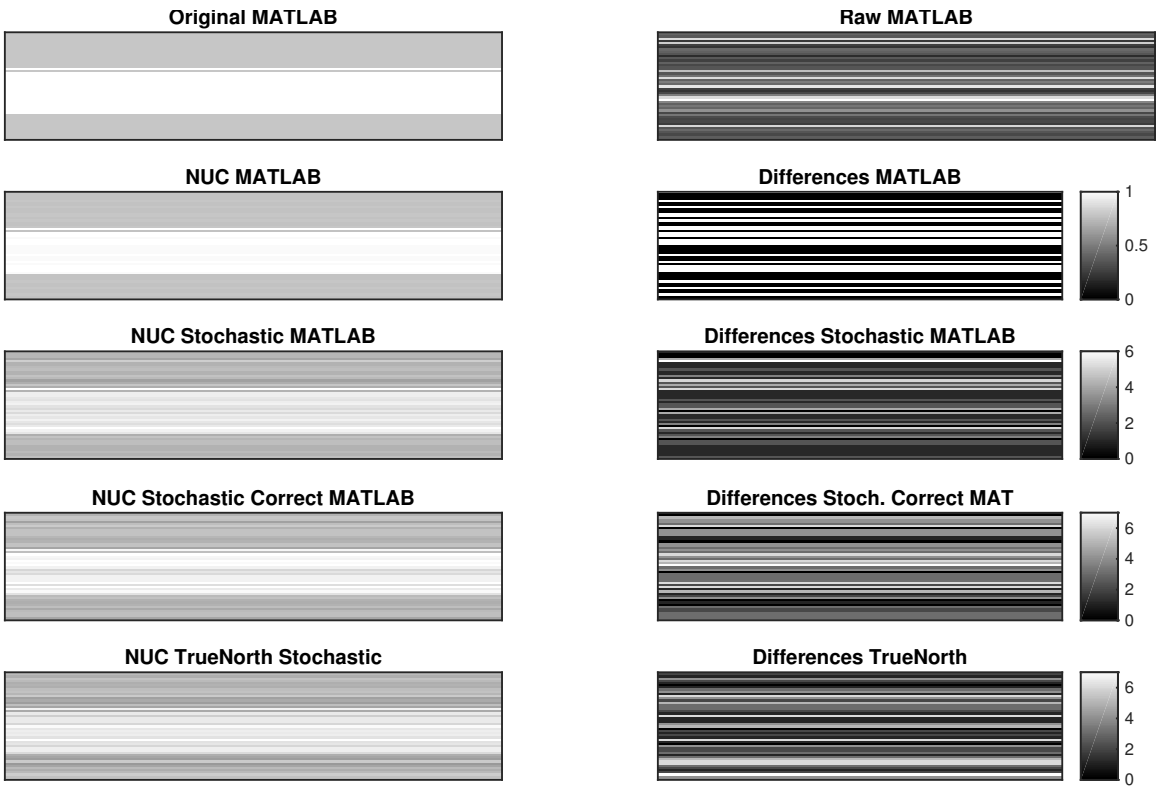


Figure 6.12: 100 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 50 rows, and 1 column.

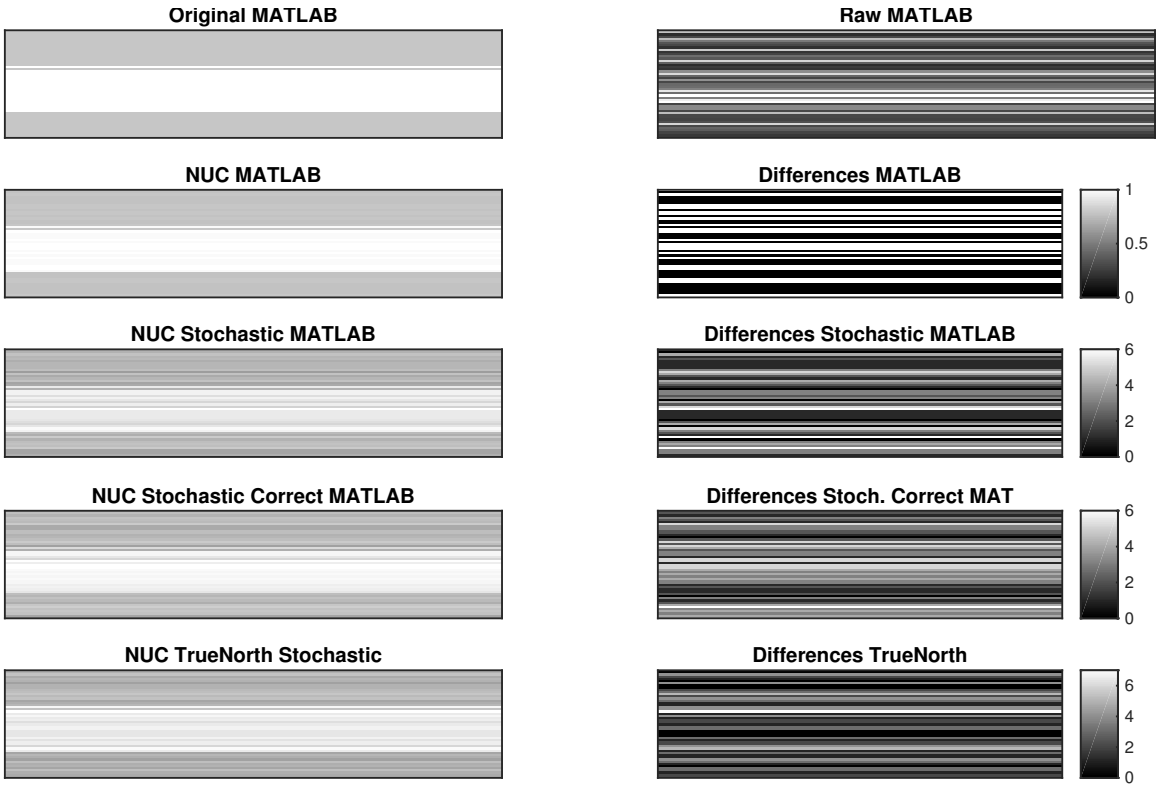


Figure 6.13: 150 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 50 rows, and 1 column.

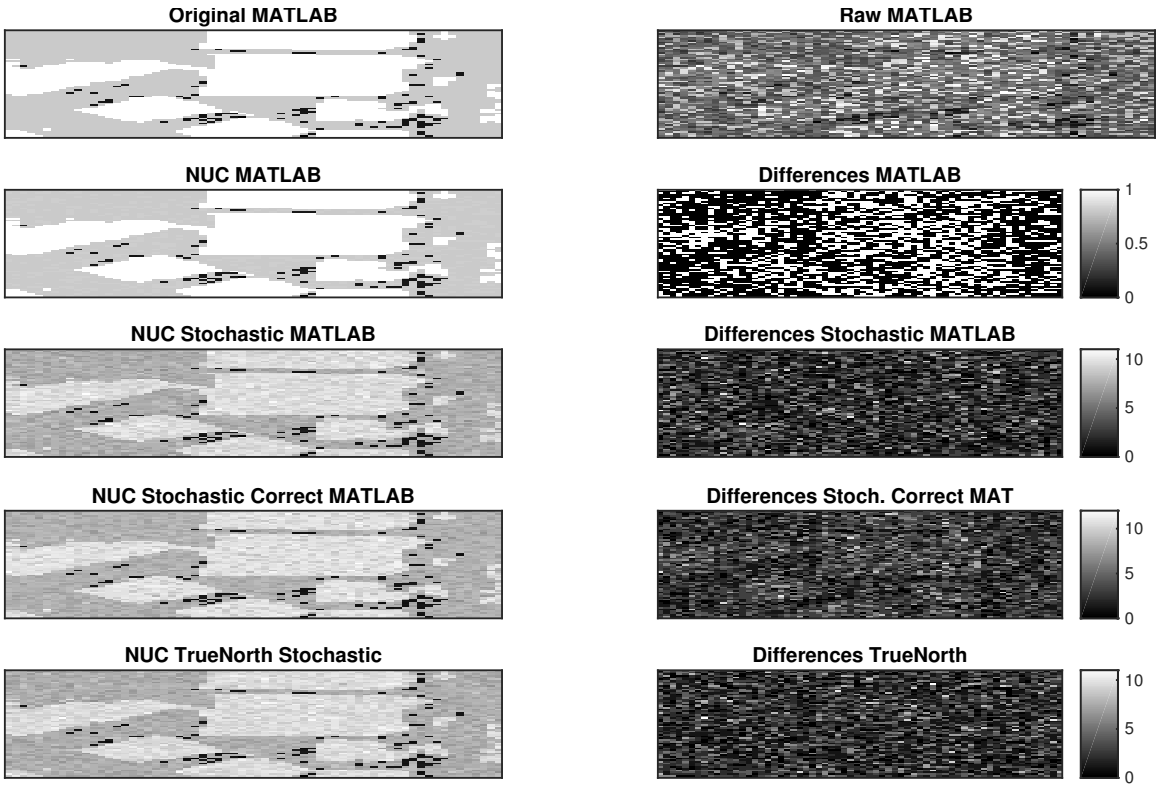


Figure 6.14: 100 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 64 rows, and 64 columns.

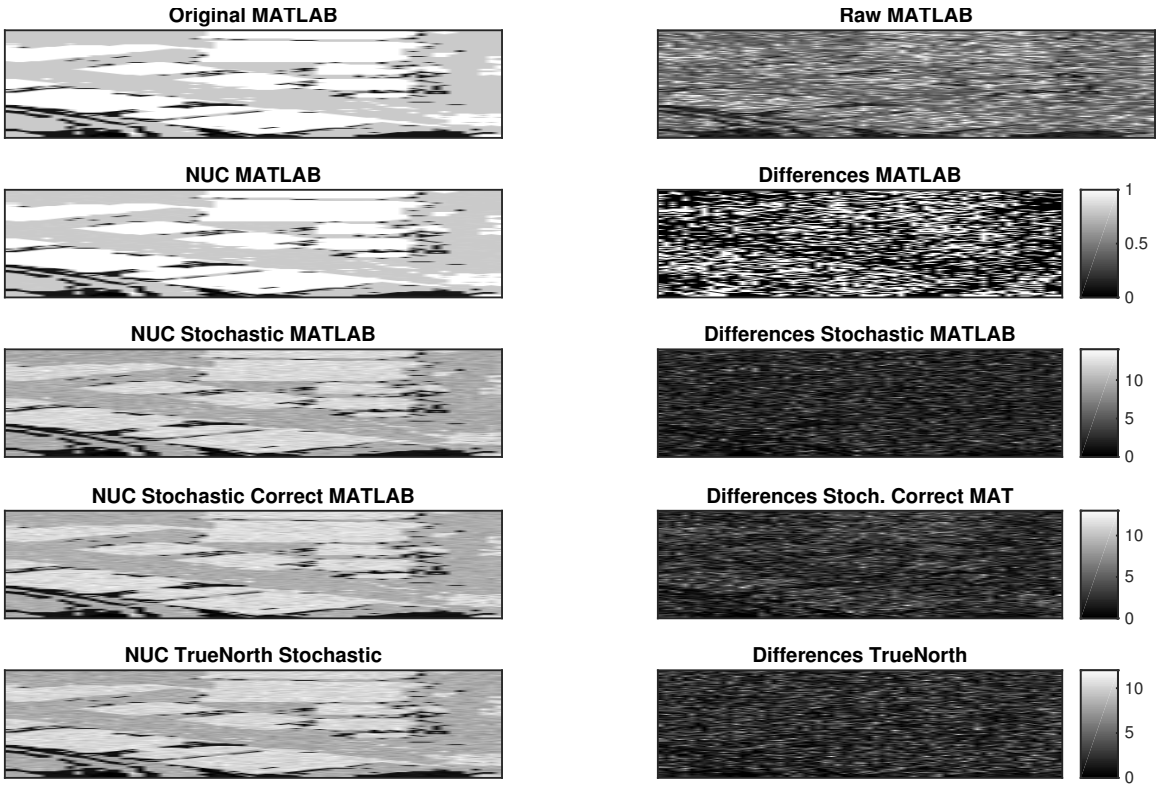


Figure 6.15: 100 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 128 rows, and 64 columns.

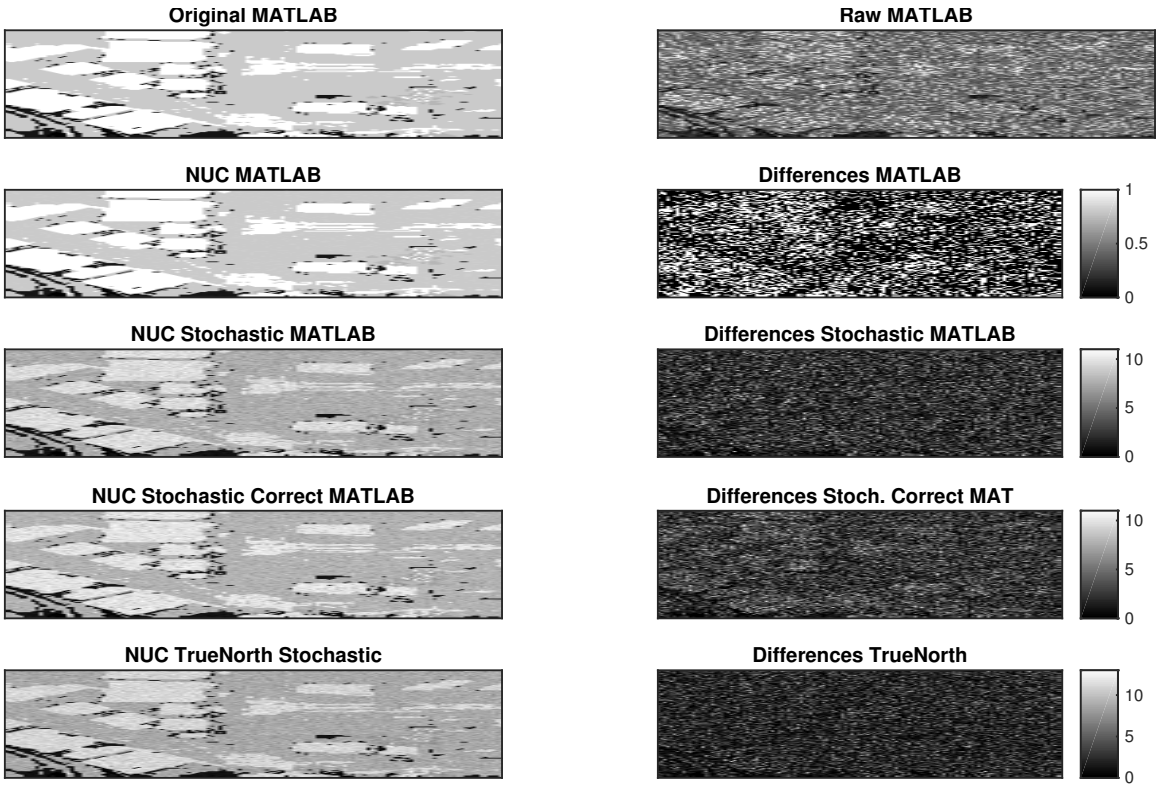


Figure 6.16: 100 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 128 rows, and 128 columns.

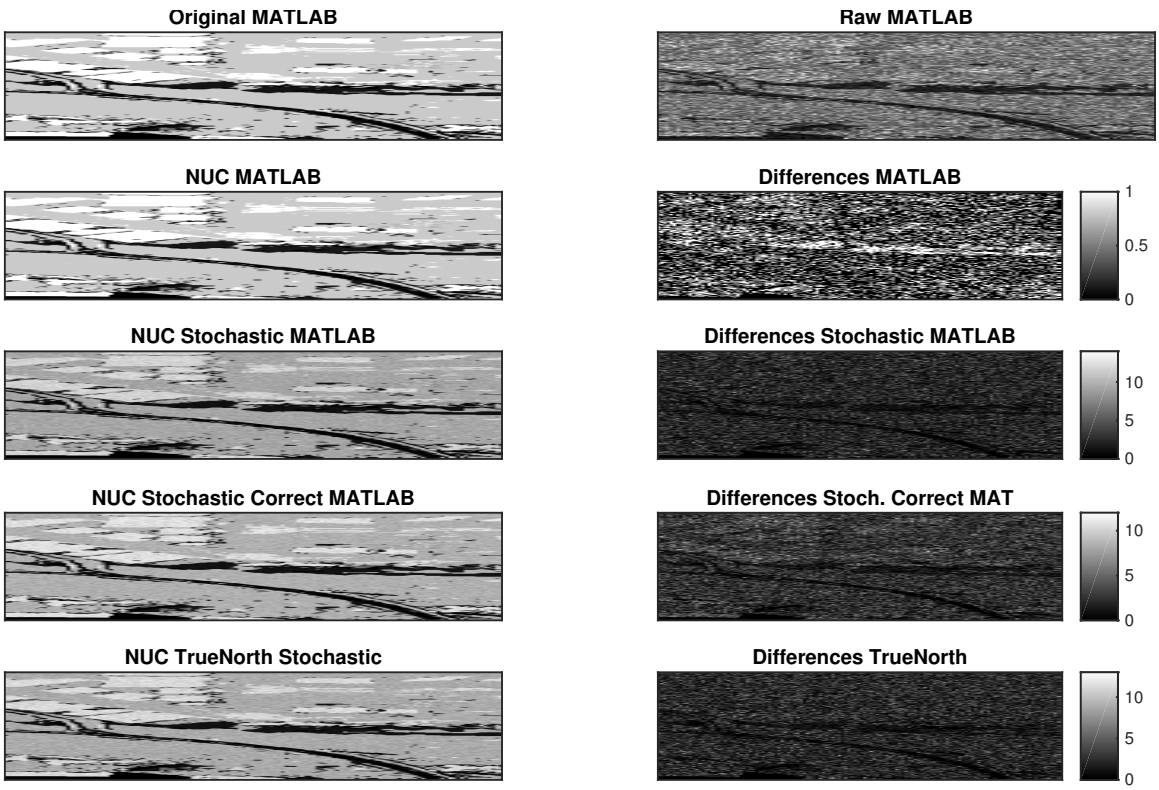


Figure 6.17: 100 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 256 rows, and 128 columns.

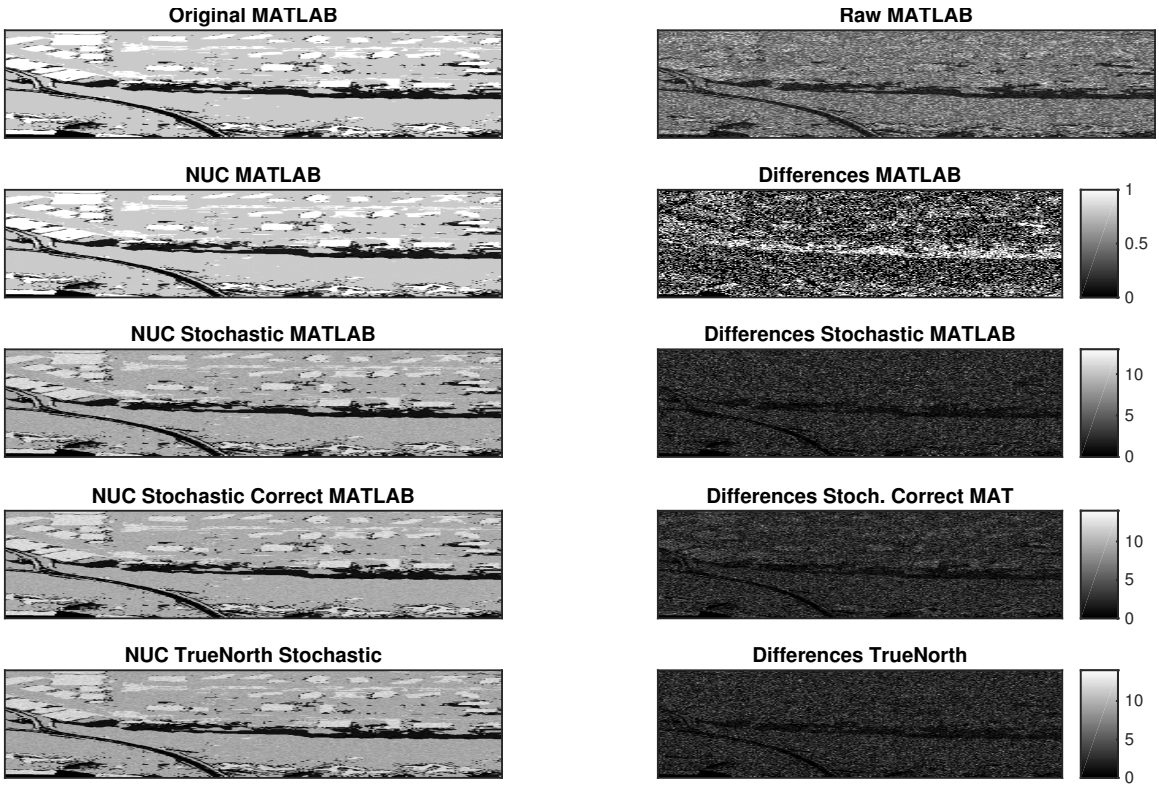


Figure 6.18: 100 averaging trials done here. MATLAB NUC, stochastic NUC in MATLAB, "correct" stochastic NUC in MATLAB, and stochastic NUC done on TrueNorth. The differences are shown in the right-hand column. There were 6 bits, 256 rows, and 256 columns.

6.5 8-bit Unsigned Vector Matrix Multiplications

The 4-bit (or less) VMM computations are best suited to the TrueNorth architecture in terms of space considerations because there are only 4 axon types that can be used to differentiate incoming spikes in the crossbar array. These types make it possible to create 4-bit numbers because each spike can indicate a different binary weight. However, it is also possible to trade space for precision by using more bits (and space on the chip) to represent the values in the VMM. Another tradeoff involves the amount of output spikes that are necessary to represent the final values. Since more bits means larger values, it also takes longer for the output spikes to flow through the system and aggregate at the end to get the results. Even if thresholds are changed similar to what was done in the Word2vec example, since the outputs are larger in magnitude, this operation creates larger errors if the amount of spikes is to remain manageable. Nevertheless, this technique can still be used in situations where more bit precision is required or larger values are needed.

This particular setup extends the 4-bit unsigned vector matrix multiplication code to allow for an 8-bit dictionary matrix. This provides a tile-able unit for matrix multiplication which can be used as a unit in a larger processing task on the core, or as a standalone unit for matrix multiplication. The 8-bit VMM corelet tile works with input vectors up to length 256 as well as matrices up to size 256x128. This

corelet requires the use of 15 TrueNorth cores (9 processing cores and 6 splitter cores) for each 256x128 block of the dictionary matrix. Andrew Dykman assisted with the implementation of the 8-bit VMM.

6.5.1 Design

The input is an unsigned vector up to 256 elements represented as a rate code. This means that each number is encoded as a number of spikes. For example, to input the vector [5, 10] one would input 5 spikes into the first axon, 10 to the second, and none to the rest. Just as with the 4-bit architecture the timing of these spikes is unimportant (these could be represented as a stochastic rate code for example), although inserting them right away means the result will arrive the quickest.

Inside each tiled corelet are 6 core splitters to allow the input vector to reach 4 layer one neurons inside the core. The 256 inputs are set up as spikes and fed to either the upper or lower first level splitter. Each half goes through the first level splitter to two second level splitters, and each layer one core is fed by an upper and a lower first level splitter. The full chart is shown in Figure 6.19.

Layer 1 acts as the dictionary matrix, with each input row corresponding to a row of the dictionary matrix, and every eight columns in the crossbar correspond to a column in the matrix (see Table 6.2). So each 1x8 block corresponds to a single element in the matrix. In each 1x8 block there is a binary big-endian representation of the number where connections correspond to 1s and lack of connections correspond

Figure 6.19: Representation of a single tiled corelet. Ellipses represent data while rectangles represent individual TrueNorth cores. Arrows show the flow of information through the corelet as a rate code.

to 0s. All synaptic connections have a weight of 1.

The number of corelets can be increased as long as there are available cores on the

1	0	1	1	0	0	0	1	0	1	0	0	1	1	1	1
1	0	1	0	1	1	0	1	1	1	0	1	1	0	1	0
0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1

Table 6.2: A Layer 1 crossbar for a 3x2 matrix consisting of the numbers $\{177, 79, 173, 218, 44, 1\}$ (left to right and then top to bottom). Each '1' corresponds to a connection in the crossbar and each '0' corresponds to a lack of connection.

chip. The number of columns in the whole matrix per tile corelet is 128, but there are 32 columns of the actual matrix in each of the four Layer 1 cores because each number in the matrix requires 8 columns in the TrueNorth core. The number of rows for the dictionary is limited to 256 regardless of number of tile corelets because all inputs are fed to all Layer 1 cores, which are limited to 256 input axons. Inputs are fed simultaneously to the Layer 1 cores across all tiles at the corresponding axons. Layer 1 neurons are grouped in fours inside of the tile corelet, and their outputs are fed directly to Layer 2 neurons.

Layer 2 acts as the first layer summer, summing the top four bits and bottom four bits of each output from the corresponding Layer 1 core. 256 outputs from the Layer 1 neurons are directly connected to 256 input axons on the Layer 2 core. The upper four bits are summed in one column of the crossbar, the next four are summed on the next (as each neuron can only have 4 synaptic weights), and so on (see Table 6.3). This means that each pair of neuron outputs corresponds to one element of the matrix multiplication, with the first neuron's output having sixteen times the weight of the second neuron's output. This weighting is achieved in Layer 3 by performing another summation (weights in Layer 2 are identical for both the 16x and 1x neurons).

1 (8)	0
1 (4)	0
1 (2)	0
1 (1)	0
0	1 (8)
0	1 (4)
0	1 (2)
0	1 (1)

Table 6.3: The Layer 2 crossbar pattern. This pattern is shared by all Layer 2 cores. Connections are illustrated as '1' and weights are in parentheses. This pattern repeats along the diagonal until the core is filled up, so there are 64 outputs from this core.

There are an equal number of Layer 2 axon inputs as Layer 1 neuron outputs, as each Layer 2 core takes every neuron output as an axon input. However, due to the 4-to-1 column to row layout, only 64 neuron outputs actually result. This is why there are four Layer 1 and four Layer 2 cores inside each tile corelet, as four 64-output Layer 2 cores can feed into the 256 inputs of a single Layer 3 core.

The Layer 3 core is the final summer, combining the upper and lower halves of the outputs of the Layer 2 cores into a single rate based output. Layer 3 is unique in the corelet, taking axonal inputs from four Layer 2 cores. As each Layer 2 core only has 64 outputs, four of them can be combined on a single Layer 3 core. Structurally, every Layer 3 core is the same across all tiled corelets, with each row having one synaptic connection and each column having two. All the neurons are identical, and the synaptic weight for the first axon in each pair is weighted at sixteen times the second axon to account for the fact that the spikes arriving at the first axon represent the more significant bits of the multiplication results. The outputs from these 256

neurons are the outputs from the entire tile corelet and are encoded with a rate code representing the results of the vector matrix multiplication.

6.5.2 Results and Discussion

Two different parameters were examined. One was the number of tiles in the VMM which controls the number of columns in the matrix. The other, which is here called the "count factor," controls the amount of quantization error in the results by determining how often the linear neurons spike. One stage in the TrueNorth pipeline is set up so that the neurons only spike once per count factor. This reduction in the number of output spikes makes the simulation take less time in two ways: less spikes coming out makes the simulation finish more quickly, and less spikes need to be interpreted after they come off the chip. The count factor is the same concept as the two neuron thresholds discussed in the Word2vec section (Section 6.2).

Experiments were done where the vector and the matrix were created randomly in MATLAB and run on the TrueNorth using the custom-made tiled corelet. Figure 6.20 shows the output with a count factor of 256 compared to the MATLAB output divided by 256. Figure 6.21 shows the output with a count factor of 512, this time with the MATLAB output divided by 512. Each of these experiments was run with the same pseudorandom number generator seed so the MATLAB results are identical up to the scaling factor.

If the input vectors are limited to having 1 bit precision the output can contain

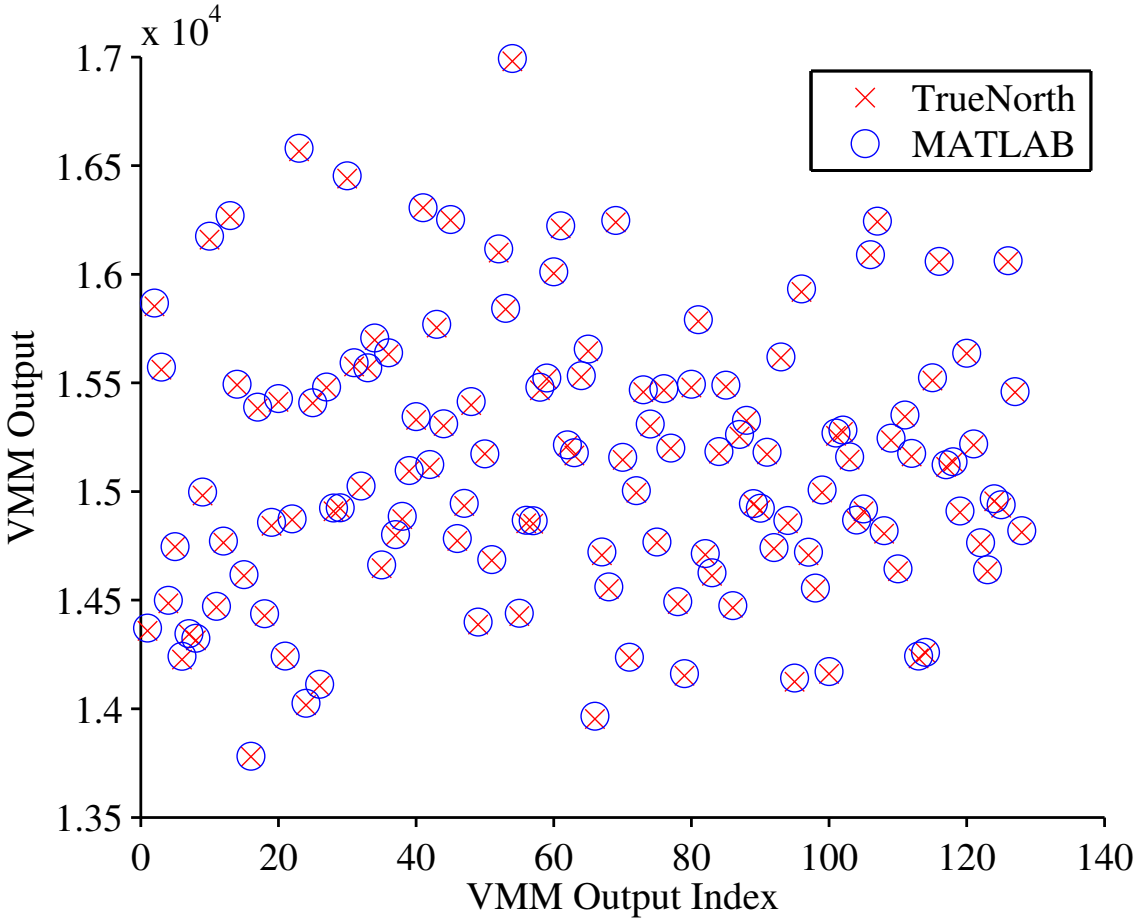


Figure 6.20: 8-bit VMM accuracy with a count factor of 256.

a maximum of 255 spikes for each term in the summation of each column when performing an 8-bit VMM. If the vectors are of length 256 then the maximum number of output spikes per column is $255 * 256 = 65,280$ spikes. These spikes come off the chip one spike at a time per output pin, so theoretically it is possible to have to wait that many time ticks for the output to complete. Using a millisecond time tick means that 65,280 spikes takes over 65 seconds to arrive. For the unsigned 4-bit case the maximum amount of time to wait with 256-element vectors is $15 * 256 = 3,840$ ticks.

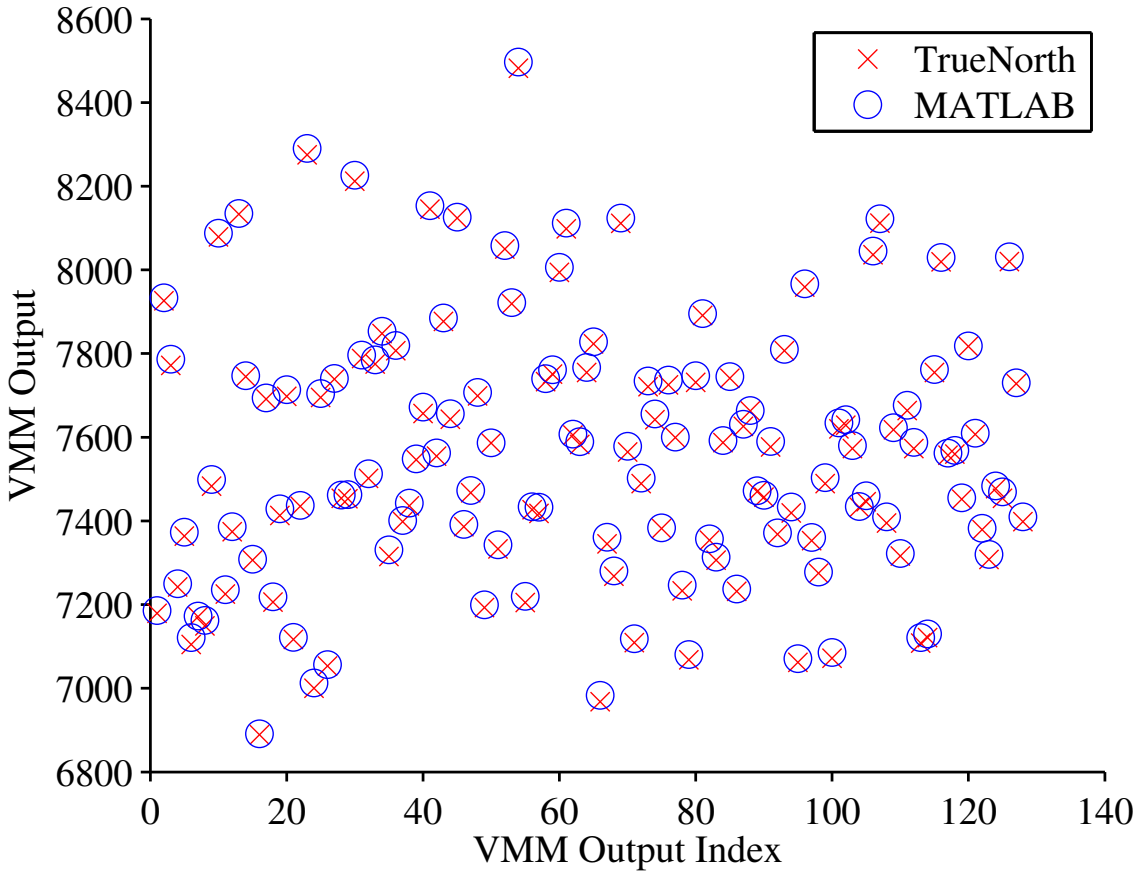


Figure 6.21: 8-bit VMM accuracy with a count factor of 512.

The signed 4-bit case has half the number of rows in the input and the matrix as well as less possible input spikes so the maximum number of ticks is $7 * 128 = 896$ ticks.

If the inputs have more levels than one bit those ticks must be multiplied by the maximum number of levels in the input, so the 8-bit version clearly requires a long wait if exact math is desired. However, if the application does not require exact computations then accuracy can be sacrificed by introducing quantization noise to the output on the chip. By changing the linear neuron threshold at some point in the processing chain the number of output spikes can be divided by that threshold at the

expense of potentially losing some spikes (threshold - 1 spikes).

The magnitude of the VMM outputs is so large that both a 256 scaling factor and a 512 scaling factor are relatively small. Therefore, even though quantization noise is again introduced by changing the neuron thresholds, the amount of error is small compared to the actual output value. For many applications this error is acceptable given the amount of speedup it creates (256 and 512 times speedups, respectively), and larger scaling factors can also be applied for various applications.

As was shown in the signed 4-bit case for the Word2vec application, if the goal is to find the largest values coming from the VMM calculation then introducing this quantization noise to improve speed at the expense of accuracy is a worthwhile trade because the largest values are still likely to be found.

Ignoring splitter cores this design uses 9 processing cores to process 128 dictionary vectors, each containing 256 elements. For comparison, the unsigned 4-bit design uses 2 processing cores to process 64 dictionary words of size 256. This means that the unsigned 4-bit design (Section 6.1.5) takes approximately 44% of the number of cores to represent the same data (albeit limited to 4-bit instead of 8-bit data). In addition, the 8-bit version creates many more challenges in the time-accuracy tradeoff due to all the extra spikes produced and therefore is likely only used when precision is truly necessary and the time for computation is unimportant.

Each 8-bit corelet uses 15 cores, allows for 128 dictionary words of size 256, takes 256-element inputs, and creates 128 outputs. With 4096 TrueNorth cores this provides

space for 273 tiled corelets and 34,944 8-bit words in the dictionary.

Chapter 7

Cognitive Audio-Visual Beamforming

Previous chapters in this thesis have thus far focused on neuromorphic hardware/software architectures that include performing computations using spiking neurons. However, brain-inspired parallel computing is not solely limited to computations that are currently done using actual neurons. Brain-inspired computing can take concepts from the human brain and use them to advance the state of the art in other ways, paving the way for possible spiking neuron computations in the future. Much of the work in this chapter has been previously published[79] by the author of this thesis.

This chapter discusses beamforming and then localizing sounds in a 3D environment. However, rather than simply locate sounds occurring in a scene, "cognitive" signal processing is done to specify which sound characteristics are important to lo-

calize and then hone in on exclusively. Just as the human brain and audio system can focus on one particular sound and determine where it is coming from while ignoring other sounds, this work shows how similar localization tasks can be accomplished in a parallel architecture. High-level objectives can also be used to direct the cognitive system to focus on specific types of sounds.

In addition to the sound capabilities of this system, 360 degree camera images are incorporated into the scene's analysis to gain a better understanding of what is occurring and provide an opportunity to provide further analysis. The sound localization results are overlaid on the 360 degree video frames to provide a full audio-visual experience. The combination of audio results with video frames also means that visual scene analysis algorithms such as visual salience map techniques[80–83] can be run on the videos and also included in this system's analysis. These visual algorithms have been implemented on FPGAs[84] which allows for rapid analysis in combination with the audio algorithms discussed in this chapter. The combination of separate auditory and visual information used for a localization task is a first step towards cognitive audio-visual scene analysis.[85,86]

Beamforming is a technique that can virtually "steer" a microphone array in a given direction so that sounds in this direction are emphasized while those from other directions are attenuated. Localization is the process of determining from which direction a sound emanates and can sometimes include using beamforming algorithms to determine which direction contains the most amount of acoustic energy.

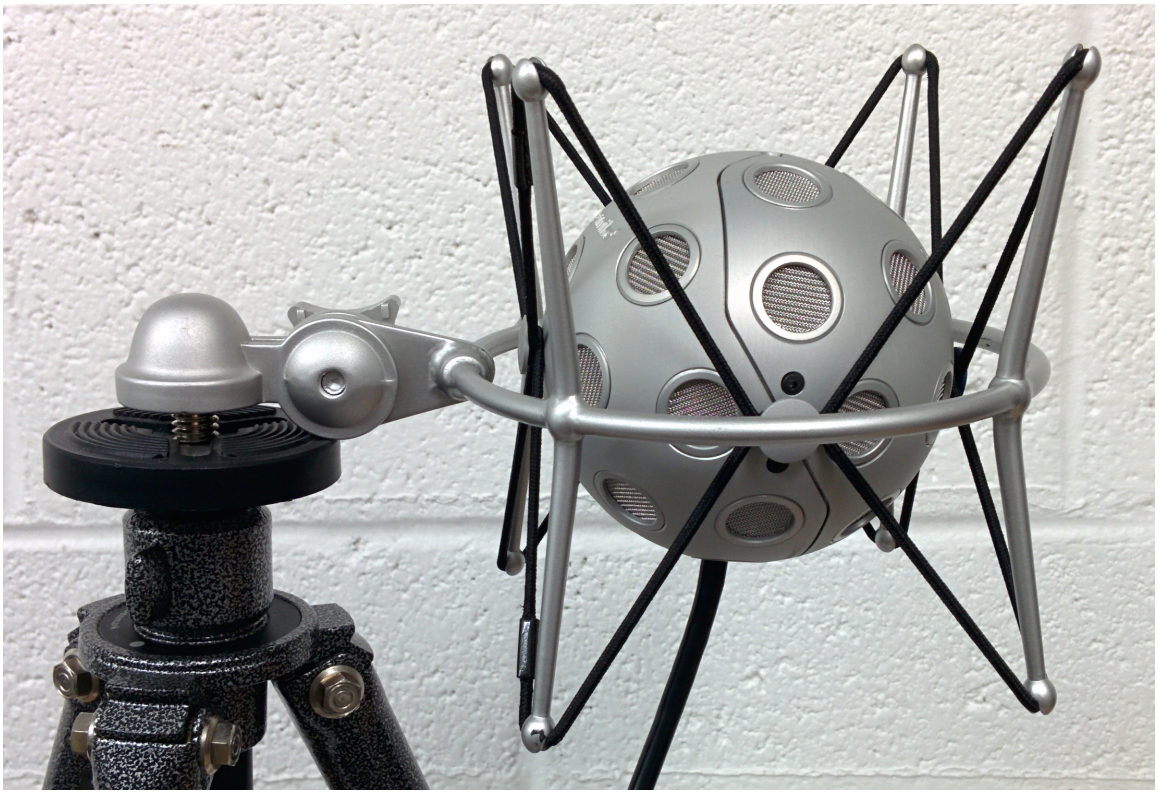


Figure 7.1: MH Acoustics Eigenmike spherical microphone array.

Figure 7.2: Sony Bloggie MHS-FS1K.

Two hardware systems were used to perform the work in this chapter. The first system consists of two instruments combined together. The MH Acoustics Eigenmike spherical microphone array[87,88] (see Figure 7.1) consists of 32 microphones. The radius of the sphere is 4.2 cm, and the sampling rate is 44.1 kHz. Visual recordings were done using a Sony Bloggie MHS-FS1K (see Figure 7.2), which is an economical video camera that captures video in high definition (HD) resolution, 1920x1080p. The camera comes with a small 360-degree lens attachment along with software to unwarp the video and turn it into a wide panorama of the scene at a usable resolution of about 1280x182p.

The second system is the much more expensive VisiSonics audiovisual array.[89] This system contains 64 microphones (again setup for equal microphone weights) that record audio at 44.1 kHz along with 15 cameras that each record video at a resolution of 1328x1048p. Software provided with the array stitches the videos together and creates a panoramic view. The software can also perform localization and other features, but this thesis only used the stitched panoramic video and raw microphone data so that algorithms and software implementation could be fully explored.

The geometry of each microphone array is important. The Eigenmike array's 32 microphones are arranged at the centers of the faces of a truncated icosahedron, which allows for equal microphone weights when performing beamforming without letting the errors become too large[90] (see Equation 7.8 in Section 7.1). In addition, the VisiSonics array's 64 microphones are positioned to minimize errors when performing

beamforming.

Both arrays enable the use of Spherical Harmonic Beamforming because a spherical configuration is required, and a standard spherical coordinate system is used in this work. The azimuth angle, $\phi \in [0, 360]$ degrees, describes the horizontal angle in the coordinate system, and the elevation angle, $\theta \in [0, 180]$ degrees, describes the vertical angle. An elevation of $\theta = 0$ degrees points directly up, 90 degrees is completely horizontal, and 180 degrees points directly down.

7.1 Spherical Harmonic Beamforming

The sound source is assumed to be a plane wave with wavenumber $k = 2\pi f/c$ coming from the direction $\{\theta_s, \phi_s\}$ where θ_s is the elevation angle of the sound source and ϕ_s is the azimuth angle of the sound source. In that case the following is a solution to the wave equation representing that sound source at a distance a from the origin of the coordinate system which is located at the center of the microphone array:[87,91]

$$G = 4\pi \sum_{n=0}^{\infty} i^n b_n(ka) \sum_{m=-n}^{n} Y_n^m(\theta, \phi) Y_n^{m*}(\theta_s, \phi_s). \tag{7.1}$$

Here, $i = \sqrt{-1}$, Y_n^m are the spherical harmonics of order n and degree m, and $*$ denotes the complex conjugate. b_n is a coefficient that applies when the observation

point is located on the surface of a rigid sphere of radius a:[91]

$$b_n(ka) = j_n(ka) - \frac{j_n^{'}(ka)}{h_n^{'}(ka)} h_n(ka) \tag{7.2}$$

because the sound reflects off the sphere. Other coefficient values result when the microphone array is located on a mesh, for example. j_n is the spherical Bessel function of order n and h_n is the spherical Hankel function of the first kind. The spherical harmonics are defined in the following manner:[91]

$$Y_n^m(\theta, \phi) = \sqrt{\frac{(2n+1)}{4\pi} \frac{(n-|m|)!}{(n+|m|)!}} P_n^m(\cos\theta) e^{im\phi}, \tag{7.3}$$

where P_n^m are the associated Legendre functions. The spherical harmonics are orthonormal which is important to note later on as the beamformer is constructed.

In order to steer the array in a given direction (beamforming), an ideal beamform function can be constructed so that every direction other than the one chosen is attenuated:[92,93]

$$F(\theta, \phi, \theta_s, \phi_s) = \delta(\theta - \theta_s)\delta(\phi - \phi_s). \tag{7.4}$$

This ideal beamformer is not very useful, however, because it does not show how to actually perform this steering of the array. It simply states the objective for how a perfect beamformer should behave.

On the other hand, any square-integrable function $g(\theta, \phi)$ on the unit sphere can

be expressed as an infinite sum of spherical harmonics:[91]

$$g(\theta,\phi) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} A_n^m Y_n^m(\theta,\phi), \tag{7.5}$$

where A_{nm} are weights that depend on the function $g(\theta,\phi)$. To approximate the ideal beamformer response F the weights are chosen as:[92,93]

$$A_n^m = Y_n^{m*}(\theta_s,\phi_s). \tag{7.6}$$

The weights shown in Equation 7.6 intuitively work because of the aforementioned orthonormality of the spherical harmonics. Therefore, the only terms that contribute to the function are the ones pointing in the same direction as the sound source.

The number of microphones in a spherical array that are used to spatially sample the incoming sound is finite. It is possible[92] to make the weights A_n^m depend on the incoming acoustic velocity potentials over the surface of the sphere, $\Phi_a(\theta,\phi,\theta_s,\phi_s)$, and still approximate the ideal beamformer. In that case,

$$g(\theta,\phi) = \sum_{n=0}^{N} \sum_{m=-n}^{n} \frac{A_n^m}{i^n b_n(ka)} Y_n^m(\theta,\phi), \tag{7.7}$$

and

$$A_n^m = \int_0^{2\pi} \int_0^{\pi} \Phi_a(\theta,\phi,\theta_s,\phi_s) Y_n^{m*}(\theta,\phi) \sin\theta d\theta d\phi, \tag{7.8}$$

where N is the maximum order of the spherical harmonics. The integral in Equa-

tion 7.8 is approximated by utilizing the cubature rule,[92,94] which requires that the microphone array has at least $(N+1)^2$ microphones. Thus the approximate beamformer $g_N(\theta,\phi)$ steered to look in the direction (θ,ϕ) is[92]

$$g_N(\theta,\phi) = \sum_{q=1}^{Q} W_q \Phi_a(\theta_s,\phi_s,\theta_q,\phi_q), \tag{7.9}$$

where Q is the number of microphones in the array and Φ_a are the velocity potentials at the microphones located at $\{\theta_q,\phi_q\}$.

W_q is defined in the following manner:

$$W_q = \sum_{n=0}^{N} \frac{1}{i^n b_n(ka)} \sum_{m=-n}^{n} w_q Y_n^{m*}(\theta_q,\phi_q) Y_n^m(\theta,\phi), \tag{7.10}$$

where w_q are the weights for the cubature formula.[90] As mentioned earlier the weights w_q can all be set to 1 here for both the Eigenmike microphone array and the VisiSonics microphone array due to the geometry of each system. They are each designed to minimize errors in the cubature approximation and keep the weights as equal as possible.

The microphones in the array do not directly record the velocity potentials Φ_a required for beamforming, but the phase information can be recovered by performing a Fast Fourier Transform (FFT) on the recorded sound data from each microphone[92] which provides a scaled version of the velocity potentials. Since the FFT result is in the frequency domain, Equation 7.9 can be modified to sum over the desired

frequencies:

$$g_N(\theta, \phi) = \sum_f \sum_{q=1}^{Q} W_q \mathcal{F}_f(\theta_s, \phi_s, \theta_q, \phi_q), \tag{7.11}$$

where $\mathcal{F}_f(\theta_s, \phi_s, \theta_q, \phi_q)$ is the FFT response for a given frequency indexed by f from the microphone signal recorded at position (θ_q, ϕ_q). This also means that frequencies of interest can be specified by only including those FFT values in the beamformer response and ignoring all others. In order to perform localization and determine where sounds are coming from a grid of look directions is specified by creating an array of azimuth and elevation angles for which beamforming is performed in those directions. Then the magnitude of the response in each direction is calculated to form a map indicating the amount of acoustic energy coming from each direction. See Section 7.2 for some examples using real data. The computation done for each direction in the localization map is separate from the others and can therefore be parallelized to drastically speed up these computations.

Equation 7.11 is the basic building block for creating a cognitive beamformer. The ability to choose time/frequency windows of interest is built into the beamforming framework, so intelligent choices can be made here when deciding on the characteristics of the sound of interest. Intelligence can be built into the system to differentiate between different types of sounds by including only time/frequency windows that suit the particular sounds of interest. These techniques are explored in the next section.

7.2 Experiments

Since the Eigenmike/Bloggie system consists of two separate pieces of hardware, the data recorded using those devices must be synchronized. A MATLAB Graphical User Interface (GUI) was developed for this purpose which loads the sound data from the Bloggie video and one channel of the Eigenmike so that they can be shifted until they best align. It helps to have an obvious audio cue such as hands clapping occur at the beginning of the recording to assist with alignment. The software allows the user to manually align the two time-series waveforms for analysis with buttons for shifting the signals.

The software also resamples the Bloggie audio because it is sampled at 48 kHz as compared to the Eigenmike's 44.1 kHz sampling rate. The localization library itself was implemented in C++ using functions from the Boost C++ library,[95] and a MATLAB MEX function was created to access this functionality directly from MATLAB. The FFT is computed in MATLAB but the rest of the main beamforming and localization code is implemented in the C++ code.

7.2.1 Human Voices

An outdoor experiment at Johns Hopkins University in Baltimore, Maryland was conducted in order to test the functionality of the Eigenmike system coupled with the localization software developed in this thesis. Three participants were located

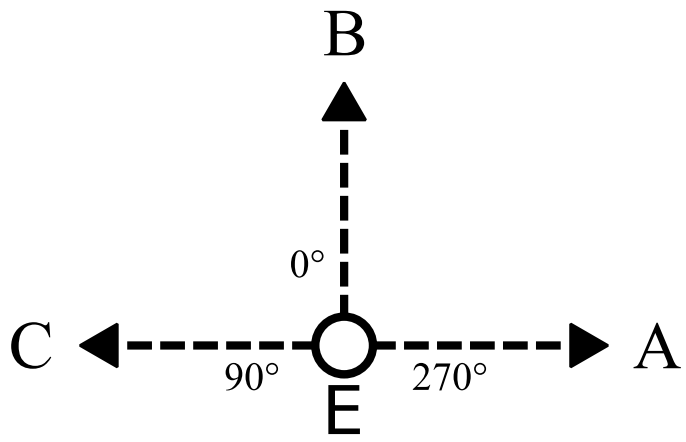


Figure 7.3: Experimental setup for three participants saying the letters "A," "B," and "C," respectively. The Eigenmike, labeled "E," was located in the center. The approximate azimuth angles for each person are shown.

in three separate positions, each approximately 4.9 meters away from the Eigenmike array. With respect to the Eigenmike's microphone coordinate system, the azimuth angle for each person's location was 270, 0, and 90 degrees. Each person was assigned a letter, "A," "B," and "C," respectively (see Figure 7.3), and each letter was yelled from the designated location one at a time in order.

Figure 7.4 shows a spectrogram created from the first channel of the Eigenmike array for this experiment. The blue boxes highlight the three time/frequency ranges sent to the localization algorithm in order to estimate the position of each person. The time-frequency intervals in the spectrogram were chosen for two main reasons. The first reason is that by ignoring the low-level background noise other sounds in the environment are better ignored including cars that were driving by. The second reason is that choosing a high-energy portion of the spectrogram corresponding to each person saying the letter maximizes the ability of the beamforming algorithm to

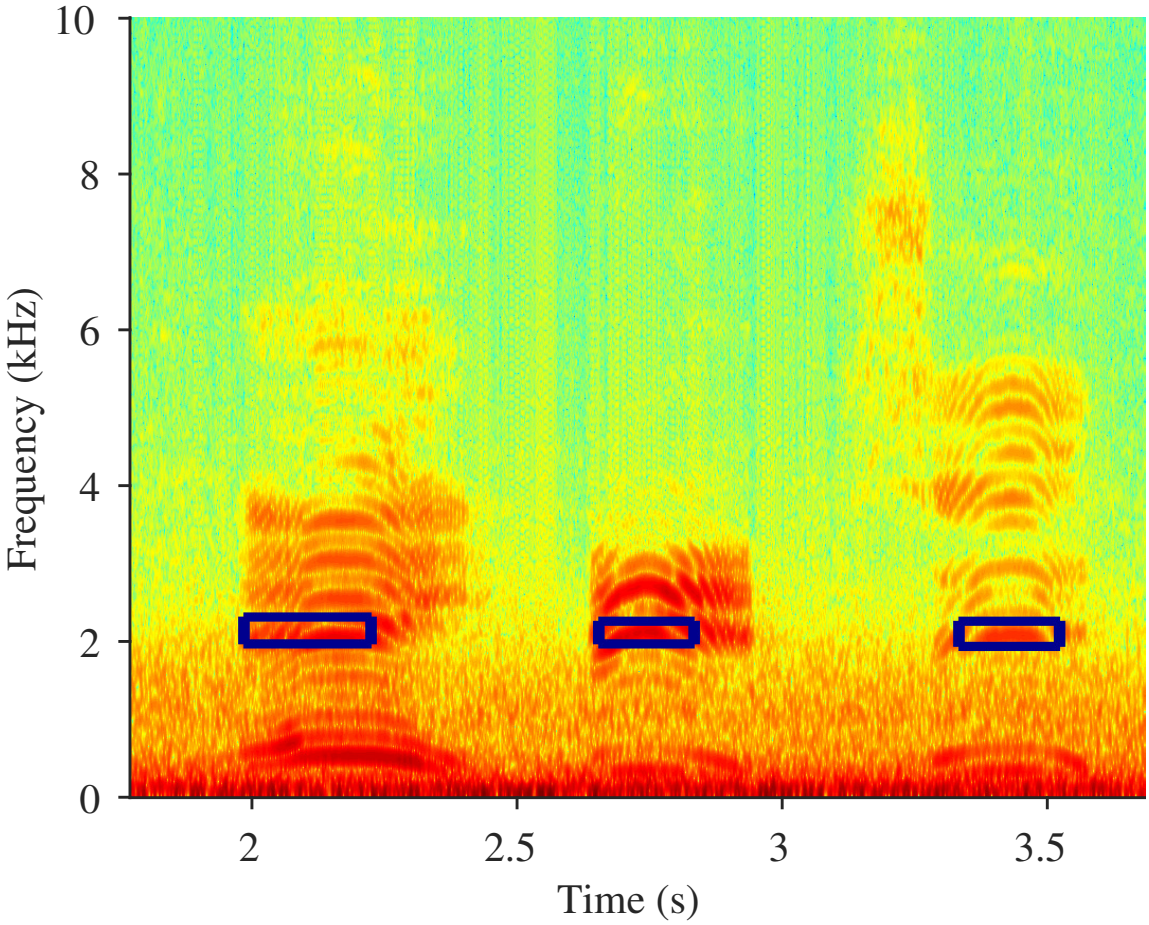


Figure 7.4: Spectrogram for three participants saying the letters "A," "B," and "C," respectively. The audio was taken from microphone 1 of the Eigenmike array, and the blue boxes indicate the time/frequency ranges selected to localize each sound (one box per localization in this example).

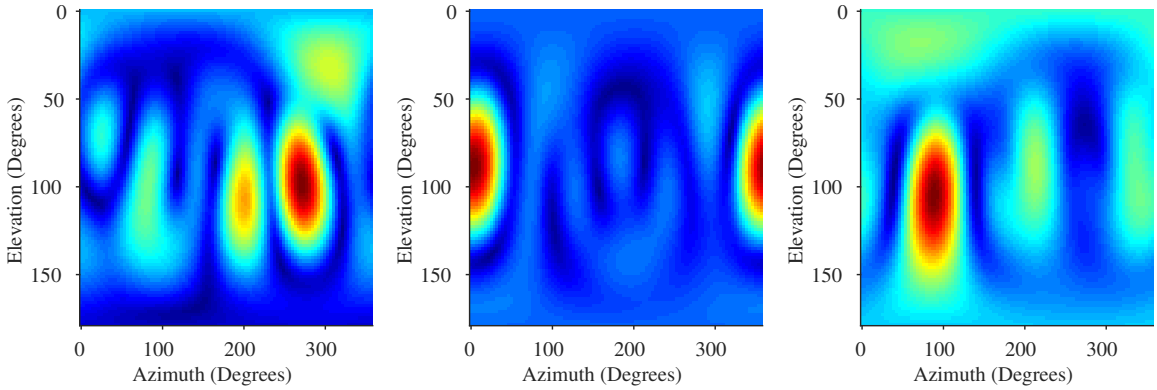


Figure 7.5: Localization maps for three participants saying the letters "A," "B," and "C" at azimuth angles of approximately 270, 0, and 90 degrees, respectively. The ground was sloped so that the person saying the letter "B" was at a somewhat higher elevation than the other two participants.

pick up the noise of the signal of interest.

The resulting localization maps can be seen in Figure 7.5. These maps were formed by first creating a grid of azimuth and elevation angles before steering the microphone array toward each of these angle combinations. The highest order of spherical harmonics was chosen as $N = 3$ for these experiments. The resulting energy from each of these directions was then plotted in a scaled image and displayed in the figure.

The azimuth angles of maximum intensity were right around 270, 0, and 90 degrees, as expected. In addition the experiment was performed on sloping ground, with the 0-degree position higher in elevation than the 270- and 90-degree positions, which were at approximately the same, lower elevation. It can be seen that the maximum of the "B" response (at zero degrees) in Figure 7.5 is at a higher elevation than the other two maxima.

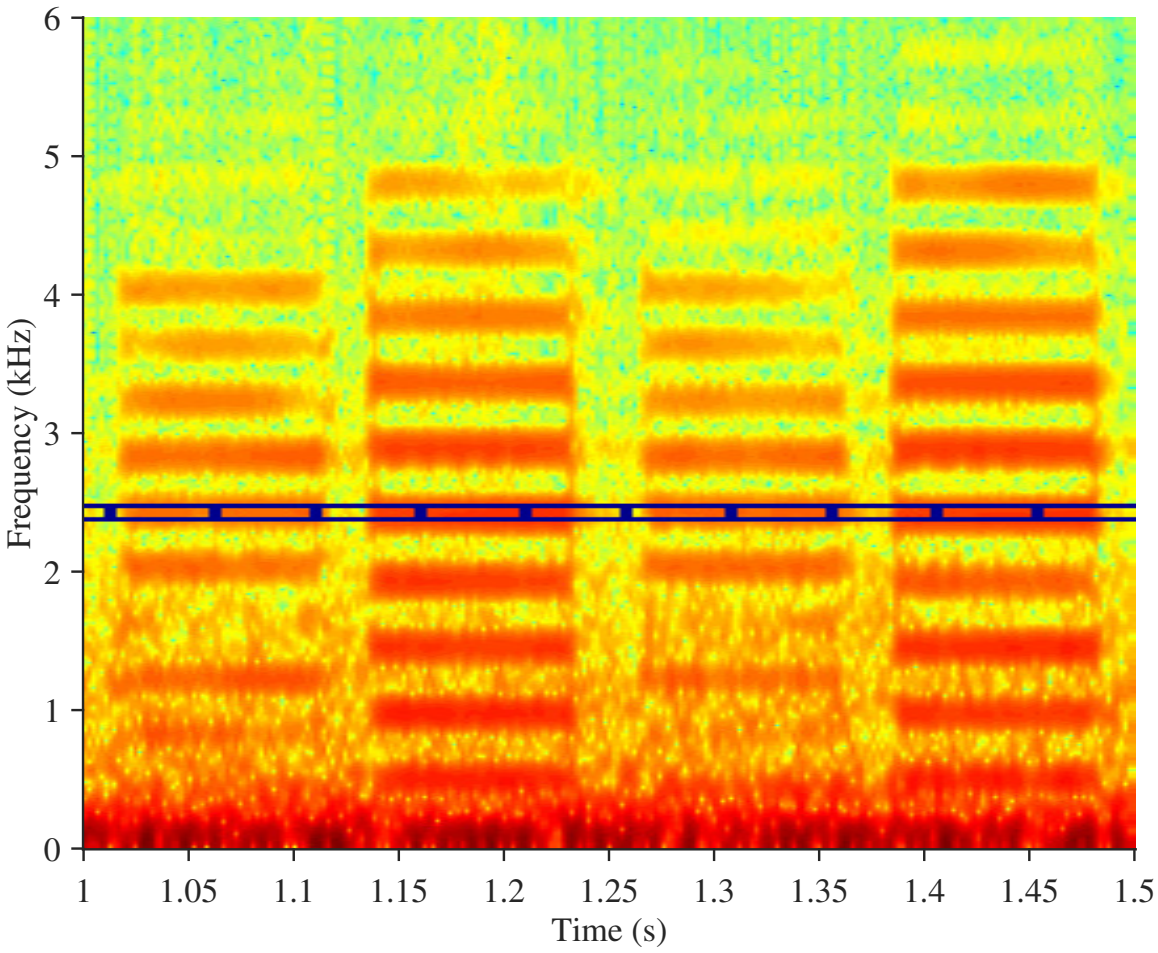


Figure 7.6: Spectrogram from a short segment of the AB Tones experiment using the Eigenmike array. The blue boxes represent the time/frequency windows used to automatically localize the tones emitted from the speakers.

7.2.2 AB Tones

Another experiment involved playing sounds through two portable speakers (JAM Plus wireless Bluetooth speakers) and localizing those sounds. Two participants each held a speaker over his head and walked around at various distances from the hardware systems. The same layout shown in Figure 7.3 was used for this experiment, except that both acoustic arrays recorded the scene and that the sound sources were moving

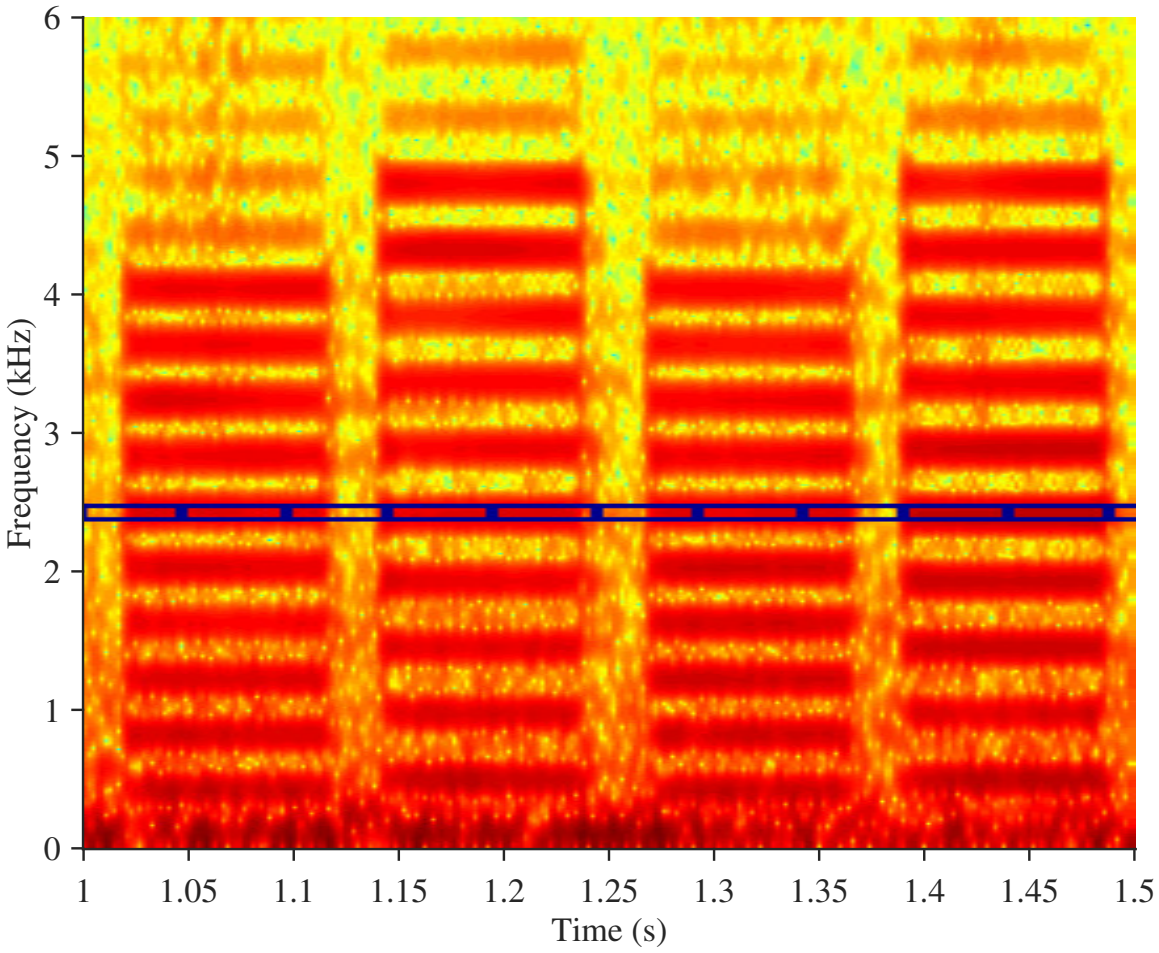


Figure 7.7: Spectrogram from a short segment of the AB Tones experiment using the VisiSonics array. The blue boxes represent the time/frequency windows used to automatically localize the tones emitted from the speakers.

rather than stationary.

Each speaker played a separate sound channel, and two different tones were played, one from each speaker. These tones were harmonic tones with fundamental frequencies at 400 Hz and 475.7 Hz and included 10 total frequency components up to 4000 Hz and 4757 Hz, respectively.[85,86] The tone durations were 100 ms including 10 ms onset and offset ramps, and a 25 ms gap was left between successive tones.

The two spherical acoustic arrays (Eigenmike and VisiSonics array) both recorded the scene roughly 3-4.5 meters away from each other, and the participants holding the speakers generally walked in a circular motion at various distances around the arrays. Participants paused and turned around at various points in the recording. The distances from the recording devices to the speakers changed throughout the experiment. In addition, a car was parked between the arrays and a building, so some of the localizations were done when participants were visually obstructed by the car.

Figure 7.6 shows the spectrogram for a small portion of the recording from the first channel of the Eigenmike array, and Figure 7.7 shows the spectrogram for the same portion from the first channel of the VisiSonics array. The signals from each array were manually synchronized so that the first channel of each array matched. The blue boxes again represent the windows used to perform localization of the tones. These windows were placed in a small frequency range from about 2362 Hz to 2458 Hz which intersects with one of the harmonics of each tone's fundamental frequency

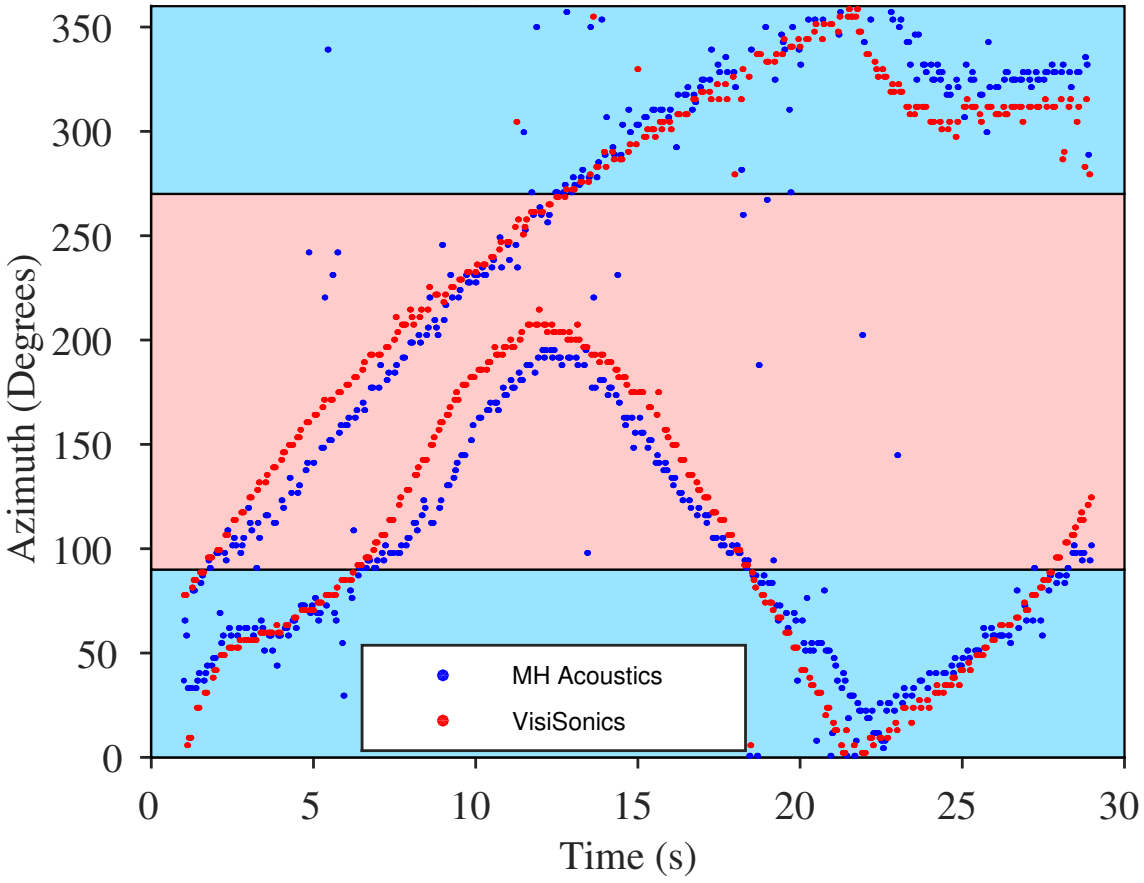


Figure 7.8: Localized azimuth angles for two participants walking around holding the speakers playing the "A" and "B" tones using the raw microphone recordings from both the MH Acoustics Eigenmike and VisiSonics audiovisual arrays. The angles wrap around at 0/360 degrees. Also note that the two arrays were located about 3-4.5 meters from each other which creates discrepancies in localized angles for certain speaker locations (close to the arrays and angles closer to 0/360 and 180 degrees).

in the spectrogram. The duration of each window was approximately 0.049 s long, and there were no gaps between successive windows.

These windows were chosen again to avoid the low-frequency noise present in the noisy Baltimore, Maryland environment. In addition, this frequency range corresponds to an overlapping region of the harmonics present in both tones so it can be used to localize both tones without having to change the windows. However, note that the windows were simply regular repeating time windows rather than being centered on each tone. Some windows even include energy from successive tones rather than only one, so it is expected that some tones will not be localized properly given this time window methodology.

Figure 7.8 shows the results of performing localization in each of these windows over a total combined time period of 28 seconds. The elevation angle is ignored and the azimuth is plotted, and the localization was done separately using both the MH Acoustics Eigenmike and VisiSonics audiovisual arrays using the raw sound data. The angles wrap around when they reach 0/360 degrees.

Since the two devices were physically separated in space, the localized angles cannot be identical. The microphone arrays were arranged side by side where the label "E" is located in Figure 7.3 (but separated by 3-4.5 meters). Therefore, when an object is located at 90 degrees or 270 degrees, the angles reported by both arrays should be equal because they are both in line with each other and the object. On the other hand, when the angle is between 90 and 270 degrees, the Eigenmike's

localized angle should be less than the angle from the VisiSonics hardware. When the object is in the semicircle between 270 and 90 degrees (the semicircle containing "B" in Figure 7.3), the Eigenmike should report an angle greater than the VisiSonics hardware (omitting, of course, the discontinuity at 0/360 degrees).

The background of Figure 7.8 indicates which localized angle values should be higher in each region. When the background color is red the red data points should have a higher value, and when the background color is blue the blue data points should have a higher value. The majority of the data points agree with these criteria. Keep in mind again that some time/frequency windows should have spurious results because they contain sounds from two successive tones and a lot of background energy in between them so looking at a single azimuth angle for those windows is not constructive.

7.3 Audio-Visual Integration

The VisiSonics audiovisual array already couples video and audio data together. However, the other system used in this work consists of two separate devices, the MH Acoustics Eigenmike microphone array and the Sony Bloggie camera.

The synchronization GUI extracts the audio data from the Bloggie camera along with the audio from the first channel of the array, and it provides manual controls to shift the two audio signals until they are synchronized. This synchronization is then

Figure 7.9: Localization results for both the VisiSonics system and the combined Bloggie camera/Eigenmike system. The panel on top is from the combined system, and the panel on the bottom is from the VisiSonics system. The blue lines indicate the localized angles calculated using only the raw microphone data from each system. This particular frame of the video illustrates the differing perspectives of the two cameras and shows why the angles in Figure 7.8 do not exactly match.

saved to a file for convenient access later on. Of course, listening to the two signals in MATLAB and looking at the spectrogram of each signal can also help.

After synchronization is complete, software provided by VisiSonics was used to stitch the videos from all 15 cameras together. Another MATLAB GUI was created to view the VisiSonics video and the Bloggie camera video for the AB tones experiment (see Section 7.2.2) simultaneously. This GUI added the ability to visualize the localized angle for each system directly in the video (see Figure 7.9).

The top panel in the Figure 7.9 comes from the combined Bloggie camera/Eigenmike system, and the bottom panel corresponds to the stitched panorama coming from the VisiSonics system. The localized angles were calculated using the spherical harmonics implementation. The azimuth angles were calibrated to the two videos simply by choosing the corresponding location for the 0/360 degree discontinuity in each video and linearly extrapolating all the other angles to the rest of the pixels in the video. In addition, the videos were calibrated with each other by circularly shifting the VisiSonics video to roughly match the Bloggie video.

In some respects the video from the cheap digicam works better than the more expensive VisiSonics system. In the VisiSonics system there are 15 separate cameras, all of which are facing different directions. The multitude of angles means that in outdoor scenes, particularly ones where the sun is out and there are also dark shadows present, it can be difficult to get the exposures of all the cameras to form a pleasing image. On the other hand, having cameras covering all directions allows for

a visualization of all elevation angles. The Bloggie camera provides a more pleasing panoramic view of the scene with a consistent exposure, but it lacks a view of the more extreme elevation angles covering areas up high and down low.

7.4 Discussion

The experiments described here use simple criteria to choose which sound signals to locate. However, further work can be done to automate the process in more complicated scenes. For example, in the AB Tones experiment (Section 7.2.2), the time/frequency selections for performing localization were not optimized for the actual timing of the tones. Many of the time windows included two successive tones that often came from very different locations. One improvement for this situation is to manually fix the timing of the windows for the sounds used in the experiment, but the real goal of this project is to make this process automatic. The spectrogram can be examined on the fly, and only high-energy areas can be chosen as windows. Other improvements include looking at the characteristics of the sounds themselves. Unique sounds can be located within the spectrogram based on their spectral characteristics which allows those specific sounds to be found in 3D space and localized.

Spherical harmonic beamforming proved to be effective using the raw audio data from both the VisiSonics audiovisual array and the MH Acoustics Eigenmike microphone array. Coupling the Eigenmike with an inexpensive digital camera provided a

reasonable audio/visual sound localization system costing much less than the VisiSonics system. However, depending on the required visual quality of the visual portion of the A/V system it may be worth coupling the Eigenmike with a nicer camera or set of cameras to better approximate the full coverage of the VisiSonics system.

Chapter 8

ARM Cortex M0 Architecture for UPSIDE Project

The Defense Advanced Research Projects Agency (DARPA) created a program called Unconventional Processing of Signals for Intelligent Data Exploitation (UPSIDE)[96] which launched in 2013 due to a "need to dramatically expand the real-time processing of wide-area, high-resolution video imagery, especially for target recognition and tracking a large number of objects." Existing systems use a considerable amount of power and one of the main goals for this project was to drastically reduce the amount of power required for these computations. As part of that project the lab at Johns Hopkins University has been working toward developing large low-power, brain-inspired, and massively parallel chip multiprocessor architectures to achieve those goals, particularly for aerial images.

The image processing pipeline includes tasks such as non-uniformity correction, debayering, dewarping, object detection, and object tracking. Non-uniformity correction[77,78] is done to compensate for the incoming raw pixel data from the sensor due to transistor mismatch (also described in Section 6.4.2 of this thesis). Debayering is required for converting the conventional Bayer filter pattern into standard red, green, and blue (RGB) image pixels. Dewarping is necessary for rotating the images coming from an aerial device so that the background is in a fixed position rather than rotating with the device in order to aid the detection of which pixels are background and which pixels are not. Once those main steps are completed, object detection and object tracking can be accomplished using various algorithms. Some of this pipeline has already been implemented and simulated on an array of FPGAs,[97] and now custom application-specific integrated circuits (ASICs) have been designed to perform these tasks.

The chips, called the nano-Abacus chiplets, contain a variety of processing units connected by two network-on-chips (NoCs), one dedicated to data transmission to and from the main memory and the other for controlling processing units (see Figure 8.1). These units, while developed for the purpose of completing the image processing pipeline, can also be used for a variety of tasks due to their general-purpose nature. They are programmable via the NoCs so their parameters can be configured for multiple datasets and situations. In addition, the order of computations is not fixed and can be changed to suit other applications.

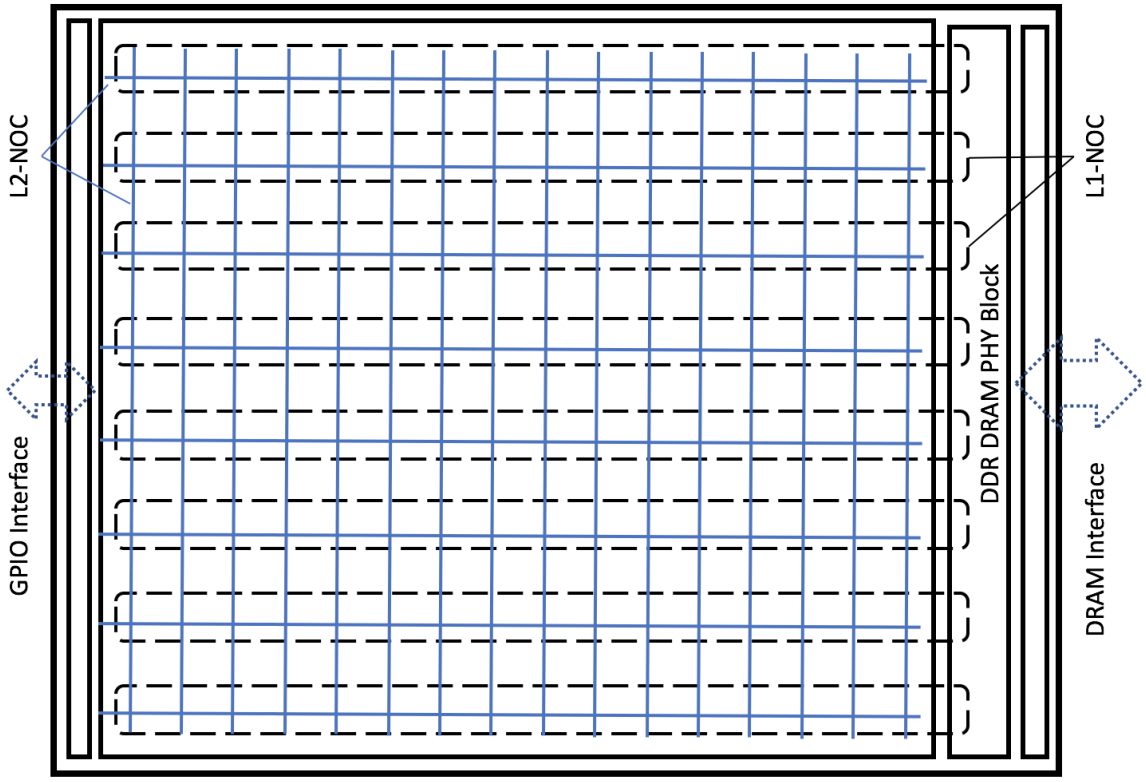


Figure 8.1: The nano-Abacus chiplet core architecture.

The design contains a mix of digital and analog circuits performing various types of unconventional processing techniques. For example, along with performing some conventional mathematics the chip also utilizes stochastic techniques[75,76] to reduce power consumption for problems not requiring high precision computations.

Handling the processing units and coordinating their efforts can be a bit complicated, especially when taking into account the fact that they can all communicate with one another and access 3D dynamic random access memory (DRAM) on each chip through the NoCs. In order to keep the chips flexible for as many applications as possible there must be away to program the chip and control all the blocks to achieve

varying goals. Thus, the ARM Cortex M0 architecture described in this chapter was designed to achieve all these goals. It is fully programmable from an external computer, has interfaces to connect to all the processing units and the main memory through the NoCs, can communicate back to an external device such as an FPGA or computer, can utilize off-chip memory to expand its storage capacity, and can perform other computations that may be inconvenient to do using all the specialized unconventional processing blocks located on the chips. In addition, although this chapter describes the general architecture for one M0 that is connected to a Serial Parallel Interface (SPI) module, a second M0 that does not have SPI capabilities is included on each chip to provide further flexibility.

8.1 Overall Architecture and Features

The ARM Cortex M0[98] is a 3-stage pipeline 32-bit fixed-point processor available from ARM as a Verilog design that can be incorporated into custom FPGA and ASIC designs. It has a hardware multiplier which as configured for this project takes 32 cycles to complete a multiplication but does not have a hardware divider. Hardware interrupts provide the ability to handle events that occur during the execution of the software loaded on the M0.

The M0 and all its peripherals are connected to an AHB-Lite bus (see Figure 8.2). The M0 boots from a read-only memory (ROM) which in this case contains a compiled

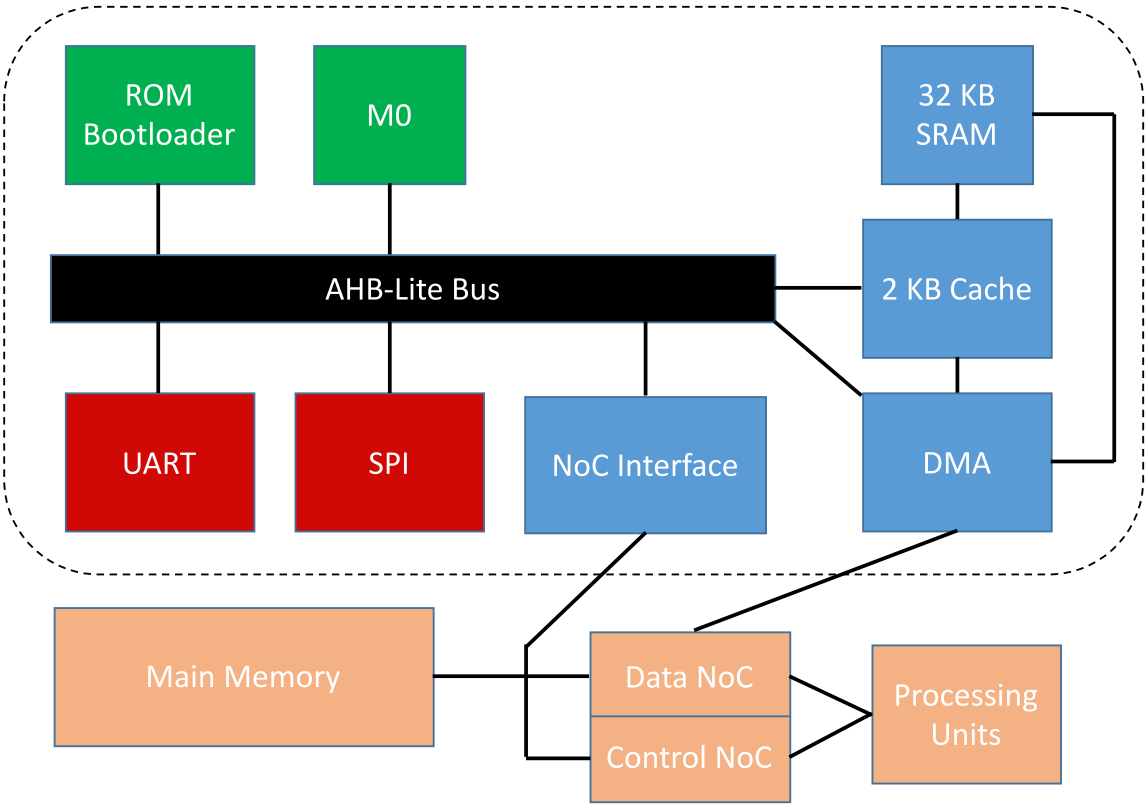


Figure 8.2: The ARM Cortex M0 Architecture.

bootloader that waits for instructions to be programmed from an external device. There is a 32 KB block of static random access memory (SRAM) as well as a 2 KB cache that speeds up accessing the SRAM data. The chips contain two NoCs that enable access to the 3D DRAM as well as the arrays of processing units. A NoC interface was designed to enable communication with the control NoC from the M0 so that all the processing units can be configured. A direct memory access (DMA) controller was created so that transfers between the main memory and the SRAM can be achieved through the data NoC without tying up the M0 with those operations.

The M0 can directly communicate off the chip using the Universal Asynchronous

Receiver-Transmitter (UART) module as well as the SPI block. The UART is used primarily for programming the M0 after the processor boots and also sending and receiving data to a separate device. The SPI module is intended to be used with external memory so that the storage capabilities of the design can be expanded beyond the 32 KB of SRAM and the main memory or to transfer data into the chips in a convenient manner.

The layout of one of these chips, called the Salamis chip multiprocessor, can be see in Figure 8.3. The M0 architecture is shown in the blue squares. One M0 has access to the SPI module and the other does not, but other than that the two architectures can access their own copy of the same peripherals.

The bootloader, UART, and SPI modules were primarily designed as part of this thesis while the blocks in blue in Figure 8.2 were primarily created by Alejandro Pasciaroni. The NoCs, processing units, and the rest of the chip work was done by other members of the lab including Alejandro.

Peripheral	**Starting Address**
ROM	0x00000000
Cache	0x60000000
UART	0x52000000
NoC Write	0x53000000
NoC Read	0x54000000
DMA	0x57000000
SPI	0x58000000

Table 8.1: Starting addresses for peripherals in Cortex M0 architecture's memory map.

All peripherals for the M0 are memory-mapped, so accessing all the modules is

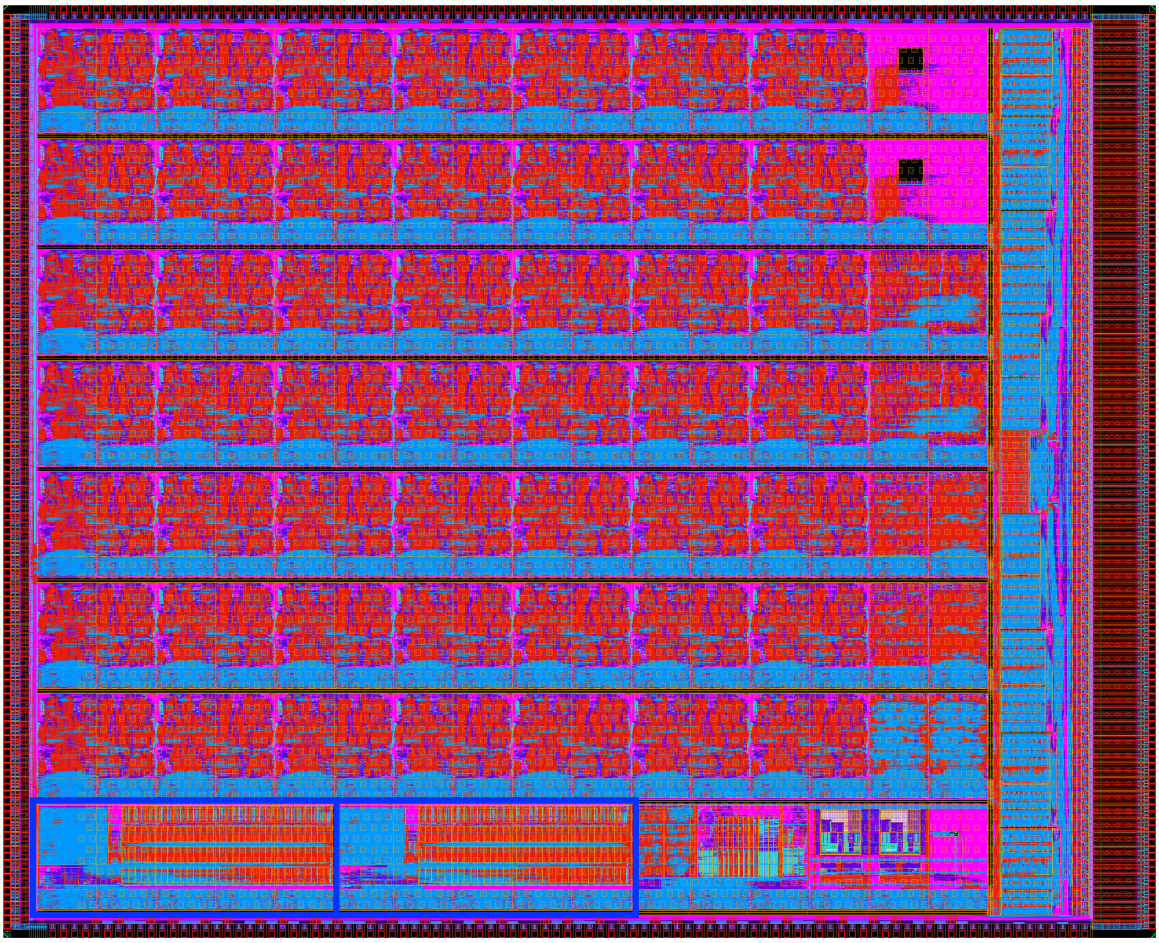


Figure 8.3: Salamis chip multiprocessor architecture.

done by accessing certain memory addresses. An address decoder in the AHB-Lite bus determines which peripheral should be selected and controls the flow of data. On the way back all the output data heading back to the M0 go through a multiplexor so that only data from the selected module are read into the M0's registers. Table 8.1 shows the starting address for each peripheral the M0 can access. The cache addresses are in a different area than the other peripherals because the cache must be executable in addition to being readable and writeable, and the M0 can execute instructions from

that range. The ability to execute instructions from the cache is essential for enabling the bootloader to program the chips and then run the code that is programmed. There are multiple configuration addresses starting with the addresses in the table for most of the blocks shown in Figure 8.2, and to perform an operation these addresses must be written to with the required information to proceed. More details about the modules and the process for using them follow in the next sections.

8.2 ROM, UART and Bootloader

When the chip boots up the M0 begins reading instructions to execute starting at address 0x00000000, so the ROM is located at that address. The ROM contains all the compiled code for the bootloader which is the first thing the processor runs on boot. The function of the bootloader is to read instructions sent to it from an external device, place those instructions in RAM, and execute those instructions. The bootloader must also handle the programmable interrupts by ensuring that external devices can set the behavior of the interrupts each time the chips are booted.

The chips have two UART wires that allow data to be transferred back and forth from an external device, one byte at a time. The baud rate of this UART module is dependent on the clock speed of the M0, so changing the M0's clock also affects the communication speed over UART. Since the M0 is a 32-bit processor with 32-bit addresses and supports 32-bit instructions, these bytes must be shifted in one at a

time to form the complete 32-bit instructions that are placed in memory.

The bootloader was written using assembly because it is possible to write the entire bootloader so data are stored only in the M0's registers rather than requiring a stack and a heap in RAM. Leaving the SRAM empty provides maximum flexibility when the program is loaded into the RAM and then later executed. The program that is loaded onto the chip then knows exactly how much space is taken up by its instructions at the beginning of RAM and is free to allocate the rest of the space for the stack, heap, and any other storage required for execution of its tasks.

An additional constraint for the bootloader is that it polls the UART rather than use the UART's interrupt capabilities. Polling achieves the same goal of not programming the bootloader in C which is that no RAM must be used for the stack because no interrupts are triggered. However, once the M0 is programmed with the code loaded through the bootloader, it is free to communicate over the UART using interrupts. Sending data via the UART is accomplished by simply writing a byte at a time to a memory-mapped register in the UART module. Data can be received by polling that same address or by waiting until the UART interrupt is triggered and then reading from that register to determine the value that was sent to the M0.

The first 4 bytes sent over UART to the chip are the number of instructions. After those values are shifted in, the bootloader knows how many 32-bit words to expect from the device sending the compiled code. All these words are shifted in. As each word arrives it is placed into the next position in RAM starting from the

bottom. Once all the instructions are placed in RAM the M0's main stack pointer is set to the address stored in the first word in memory which is the default location for the stack pointer. Then the M0 jumps to the address specified by the second word which corresponds to the beginning of the code that should execute when everything is ready.

ARM's Keil uVision Integrated Development Environment (IDE) and compiler were used to implement the bootloader as well as the code that is loaded into memory and executed by the bootloader. However, different configurations are required for each situation. The bootloader is configured to boot from ROM at address 0x00000000 whereas the code that is loaded onto the processor later is configured to execute in RAM starting at address 0x60000000 (see Table 8.1). The code loaded into the RAM also uses a scatter file to choose the exact locations of the stack and heap so that the code loaded into memory does not get overwritten by anything during program execution. The scatter file also provides more flexibility to, for example, leave some RAM open for DMA transfers or other uses.

The interrupt service routines (ISRs) must be part of the bootloader in order for them to function correctly. Since they are located in RAM instead of in ROM, each ISR jumps to the corresponding location in RAM and executes the code starting at the address stored in that RAM location. This technique allows for full programmability of the ISRs for each boot of the chips.

8.3 SPI

The Serial Parallel Interface is used to connect the M0 to off-chip memory to enable a multitude of options. For example, one use is to expand the amount of memory available to the M0, and another is to send data to and from the M0 using that memory. Basic SPI[99] has a master device and a slave device, and the implementation[100] created for the UPSIDE chips uses four wires: MOSI, MISO, $\overline{\text{CS}}$, and SCK. MOSI is master out slave in, MISO is master in slave out, $\overline{\text{CS}}$ is chip select, and SCK is slave clock. MOSI and MISO are used for sending actual data while $\overline{\text{CS}}$ is used to tell the slave module that it is active and communication will begin soon. SCK is the clock used to drive the data transfer and is sent along with the data to the slave module.

The particular memory devices the SPI block in this thesis was designed for are the Microchip 23A1024 and 23LC1024 SRAMs.[100] They support various modes of operation, but the UPSIDE ASICs were designed to use the byte operation modes. Every mode works in the same basic manner: the $\overline{\text{CS}}$ signal goes low and the SCK signal starts toggling. The data are always valid on the rising edges of the SCK signal. When the data transfer is complete the SCK signal goes back to its default (low) state and the $\overline{\text{CS}}$ signal goes back to its default (high) state. Data are always transferred in order of most significant bit to least significant bit.

The first 8 bits sent to the slave device (the memory) from the M0 are always the instruction. Reading a byte (see Figure 8.4) consists of sending the read instruction (0x03) and a 24-bit location that specifies the read address, both on the MOSI line,

and the SPI memory then sends back the byte of data specified on the MISO line. Writing a byte (see Figure 8.5) involves sending the write instruction (0x02), the 24-bit address, and finally the 8 bits of data that are to be written, all on the MOSI line.

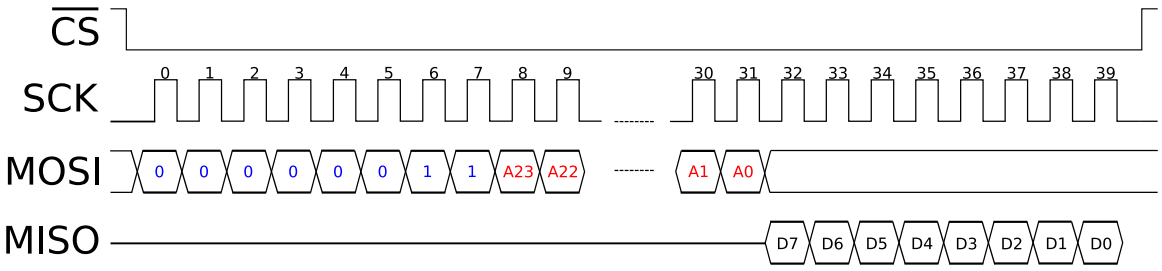


Figure 8.4: SPI timing diagram for reading a byte from an external device. The bits in blue are the instruction for reading, the values $A23$ through $A0$ comprise the 24-bit address, and the values $D7$ through $D0$ are the byte that is sent back from the external device.

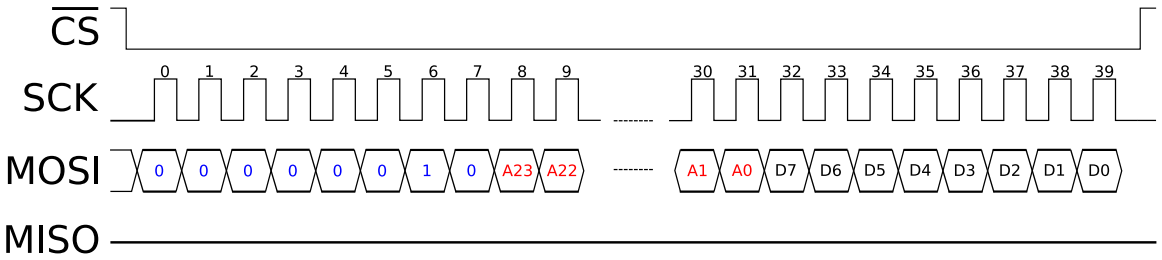


Figure 8.5: SPI timing diagram for writing a byte to an external device. The bits in blue are the instruction for writing, the values $A23$ through $A0$ comprise the 24-bit address, and the values $D7$ through $D0$ are the byte that is written to the external device.

Reading and writing a byte at a time must first be enabled by setting the mode register of the memory to correspond to byte mode. Reading and writing the mode register is similar to reading and writing a byte except that there is no address in the transaction. Reading consists of sending the 8-bit instruction for reading the mode

register on the MOSI line and then receiving the 8 bits of data that indicate the current mode on the MISO line. Writing involves sending the instruction for writing to the mode register and then sending the 8 bits corresponding to byte mode, both on the MOSI line. Once the mode register is set, data can be read and written to the external device as described above.

The M0 is a 32-bit processor, so rather than interacting with only one SPI memory device by sending or receiving one byte at a time, this architecture is designed to interface with four devices at once. All the devices share the $\overline{\text{CS}}$ and SCK signals as well as the instruction and address they specify, but the data are unique to each of the four modules. With 32 bits available, one register's value can be set which programs all four bytes at once in order to increase the throughput of one read or write operation.

Operating the SPI involves setting one or two register values depending on the mode. When writing to the M0, the 32-bit register containing each of the four bytes to be simultaneously written to the four external memory devices must be set. Then the register containing the shared instruction and address is set and the write to the external devices begins. On the other hand, reading requires simply setting the shared instruction and address register. When either a read or a write is completed, the corresponding interrupt is triggered. For a read operation the values read from external memory can be accessed by reading another 32-bit memory-mapped register, and for a write operation the next step depends on the goals of the program and

whether it was waiting for the write to finish before accomplishing another task.

8.4 SRAM, Cache, DMA, and NoC Interface

Alejandro Pasciaroni designed the SRAM, cache, DMA controller, and NoC interface, while testing and integration was a collaborative effort. These peripherals greatly extend the capabilities of the Cortex M0 by allowing it to effectively communicate with the rest of the processing units on the chips as well as access the main 3D DRAM.

The 32 KB of SRAM are used to store the instructions to be executed after the chips are programmed, the stack memory, the heap memory, and any other memory storage the programmer wants to use such as values copied from main memory using the DMA controller. However, rather than connect the SRAM directly to the AHB-Lite bus, a separate 2 KB cache was created to speed up memory accesses. The cache is also connected to the DMA controller so that when values are copied from main memory to the SRAM, the cache knows to invalidate any data it holds corresponding to the addresses involved in the DMA transfer.

The DMA controller is used to copy data either from the main 3D DRAM on the chips to the M0's SRAM or to copy data from the SRAM to the 3D DRAM. The DMA functionality is particularly useful for setting up all the processing units

and implementing the processing flow for various applications. Some processing units rely on values stored in main memory, so the M0 can use the DMA controller to write values to the DRAM, program the processing units through the NoC, and then start the algorithmic pipeline. By definition, the DMA block accesses both the main memory and the M0's SRAM without the involvement of the CPU, which allows for quicker setup when configuring the chips. While data are being shuttled using the DMA controller, the rest of the processing units can be configured by the processor through the NoC interface. When a DMA transfer is complete, there are two interrupts that can be triggered depending on the operation. One interrupt is for reading from the SRAM and the other is for writing to the SRAM. These interrupts as well as all the other M0 capabilities can be fully programmed each time the chips are reset as part of the bootstrap process using the M0 bootloader.

Programming the DMA controller involves setting some register values in the peripheral by writing to the appropriate addresses, all of which are located in the block specified by its starting address in Table 8.1. The main memory is addressed using 40 bits, so two registers are required each for the starting and ending addresses in the 3D DRAM. The starting and ending addresses in the local SRAM must also be specified as well as the direction of the data transfer.

The NoC interface allows the M0 to communicate with all the other devices using the two NoCs found on the chips, one NoC dedicated to data transfers with the main memory and the other designed to control the processing units and exchange data

with them. Since the NoCs can send packets containing 256 bits of data, there are 8 registers of data that are written to specify the packet contents. An address is specified, and a packet type is chosen because two different types of packets can be transmitted. One register is set for one type of packet and another register is set for the other. At that point the NoC interface works on sending the packet, and once the data are sent an interrupt is triggered to let the M0 know that the task has been completed. On the other hand, when a packet arrives, one of three interrupts is triggered depending on the type of incoming data. One interrupt lets the M0 know that data have arrived, and the second one occurs when a special acknowledgement packet has been received. The final NoC interrupt is reserved for the case when a control packet has been received. When data packets arrive, the M0 can read from eight registers to handle the full contents (256 bits) of the incoming packets.

8.5 Interrupts Overview

This M0 architecture design includes 9 hardware interrupts which enable a range of functionality described in the preceding sections. The following is a list of the interrupts available to the developer every time the chips are booted and programmed:

- UART byte received
- DMA read transfer completed
- DMA write transfer completed

- Network data packet received
- Network acknowledgement packet received
- Network control packet received
- Network packet sent
- SPI packet received
- SPI packet sent

Each interrupt can be programmed and sent onto the board using the bootloader, enabling a great amount of flexibility for the programmer to utilize. Communication with other processing units on the board as well as external devices can be achieved using the M0's peripherals and the interrupt signals described above. The M0 makes it straightforward to program the many processing units found on the chips so that a wide variety of tasks can be accomplished.

8.6 Programming the M0

The code written for the M0 has all been compiled using the ARM Keil tools, specifically the μVision V5.18.0.0 tools. The following sections describe how the bootloader was programmed as well as how to use the DMA, SPI, and NoC peripherals.

8.6.1 Bootloader

As described in Section 8.2, the bootloader was programmed using assembly so that no RAM is required for utilizing a stack and heap. Restricting all the basic operations to only the M0's internal registers provides the most amount of flexibility for the programmer using the chip.

The following block shows the beginning of the bootloader assembly code where the M0's vector table is established. This vector table contains 32 entries. The first entry is the beginning stack pointer address followed by multiple interrupts that are standard for the M0. The ones specifically defined for this project are the reset handler and the hard fault handler. The reset handler is the code that runs when the M0 is booted, so that handler contains the important bootloader code itself. The hard fault handler contains the code that runs when the M0 encounters a hard fault and cannot continue running. The next block of 16 external programmable interrupts correspond to custom hardware interrupts designed for this M0 architecture. This code is flexible to allow for more interrupts than are actually enabled in the hardware, but it does not hurt to have them established in software regardless.

```
PRESERVE8
THUMB

AREA RESET, DATA, READONLY ; First 32 WORDS is VECTOR TABLE
EXPORT  __Vectors

; Set up main vector table here. The first entry isn't really
; correct because it should be the initial stack pointer.
; However, since there is no stack it doesn't really matter.
; The second is the reset handler, and the fourth is the
```

```
; hard fault handler.
__Vectors
DCD Reset_Handler
DCD Reset_Handler
DCD 0
DCD HardFault_Handler
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0

; External Interrupts - These are set up so that they all
; jump to whatever they are configured to be in the
; application code loaded onto the M0 later on.
DCD One
DCD Two
DCD Three
DCD Four
DCD Five
DCD Six
DCD Seven
DCD Eight
DCD Nine
DCD Ten
DCD Eleven
DCD Twelve
DCD Thirteen
DCD Fourteen
DCD Fifteen
DCD Sixteen

AREA |.text|, CODE, READONLY
ENTRY
```

The code below is the reset handler which actually loads the data from the UART and places it into memory. The code begins by turning off interrupts and then polls the UART waiting for 4 bytes containing the amount of instructions to read. Then all those instructions are read from the UART one byte at a time and shifted into a 4-byte value to form each full instruction. Once each instruction is ready it is placed into RAM. Finally, after all the instructions are read, the stack pointer is updated to be the location specified in the compiled code sent through the UART, and the M0 jumps to the location of the new reset handler stored in memory. Then all the new code is executed.

```
; The reset handler is what runs when the processor boots and starts
; running.
Reset_Handler   PROC
    EXPORT  Reset_Handler

    ; Turn off interrupts for polling the UART instead and store the
    ; compiled application code in memory.
    LDR R1, =0xE000E100
    LDR R0, =0x00000000
    STR R0, [R1]

    ; Store the address of the UART in R0.
    LDR R0, =0x52000000

    ; Poll the UART to determine the number of instructions to be
    ; received.
    ; R1 will store the number of instructions.
    LDR R1, =0x0
    ; R3 is the current left shift amount for the current received
    ; byte.
    LDR R3, =24
    ; R4 is the number of bytes left to receive to make a 32-bit
    ; word.
    LDR R4, =0x4
```

```
; l1 is the loop for continuously polling the UART until data
; arrives.
l1
    ; Load the UART value into R2.
    LDR R2, [R0]
    ; Shift the value right 8 bits and put the result into R5 to
    ; check if the data is valid.
    MOVS R5, R2, LSR #8
    ; If the 9th LSB is 0 then data is good; otherwise poll again.
    BNE l1

    ; Now have the current received byte.
    ; Shift the value left by the current shift amount and
    ; add it to R1 which stores the current 32-bit value.
    LSLS R2, R3
    ADDS R1, R2

    ; Subtract 8 from the shift amount so the next value goes
    ; into the correct byte position in R1.
    SUBS R3, #8

    ; Subtract one from the number of bytes left and check
    ; if there are any left before continuing.
    SUBS R4, #1
    BNE l1

    ; Now have the total number of instructions so
    ; can do the same thing but add an extra outside loop
    ; over the number of instructions. R7 stores the current
    ; position in RAM which is where to place the next
    ; instruction received.
    LDR R7, =0x60000000

; This loop goes over all instructions.
instr_loop
    ; Reset the shift, number of bytes left, and current 32-bit
    ; value (here R6) for the current instruction being received.
    LDR R3, =24
    LDR R4, =0x4
    LDR R6, =0
```

```
; l2 is similar to l1 above.
l2
    LDR R2, [R0]
    MOVS R5, R2, LSR #8
    BNE l2

    LSLS R2, R3
    ADDS R6, R2

    SUBS R3, #8

    SUBS R4, #1
    BNE l2

    ; Store the current instruction into the address stored
    ; in R7 (RAM location). Then add 4 to the address (R7)
    ; so the next instruction goes in the next RAM address.
    STR R6, [R7]
    ADDS R7, #4

    ; Subtract one off the total number of instructions and
    ; when it hits zero it's done.
    SUBS R1, #1
    BNE instr_loop

    ; Set the main stack pointer to the value stored in R0
    ; which is set by the application code.
    LDR R0, =0x60000000
    LDR R1, [R0]
    MSR MSP, R1

    ; Branch to the application code which has been loaded
    ; into RAM.
    LDR R0, =0x60000004
    LDR R1, [R0]
    BX R1

ENDP
```

The next bit of code is all the interrupts in the bootloader. These interrupts

are generic because they must be programmable. The 32-word long vector table is always the first 32 words in the compiled code, so when the compiled code is placed in memory the bootloader knows which of the 32 words correspond to the ISRs. Thus, each interrupt in the bootloader loads the address of the corresponding compiled ISR, and then the processor branches to that address after taking care of the stack. When the ISR returns, the stack is also returned to its original state. Each interrupt is very similar except that the address for loading the ISR location is different in each one.

```
; Here are all the possible interrupt handlers. They are all similar
; so only the first one is commented below.
One PROC
    EXPORT One
    ; Need to push the link register which stores the location
    ; to return to once the interrupt handler is done.
    PUSH {LR, R1, R2}

    ; Load the address storing the application's first event
    ; handler address into R0.
    LDR R0, =0x60000040
    ; Load the actual address from that location into R1.
    LDR R1, [R0]
    ; Branch to R1 and let the application handle the interrupt.
    BLX R1

    ; Pop the program counter off the stack and return back
    ; to the original code that is being executed.
    POP {PC, R1, R2}
ENDP

Two PROC
    EXPORT Two
    PUSH {LR, R1, R2}

    LDR R0, =0x60000044
    LDR R1, [R0]
    BLX R1
```

```
    POP {PC, R1, R2}
ENDP

Three PROC
    EXPORT Three
    PUSH {LR, R1, R2}

    LDR R0, =0x60000048
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Four PROC
    EXPORT Four
    PUSH {LR, R1, R2}

    LDR R0, =0x6000004C
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Five PROC
    EXPORT Five
    PUSH {LR, R1, R2}

    LDR R0, =0x60000050
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Six PROC
    EXPORT Six
    PUSH {LR, R1, R2}

    LDR R0, =0x60000054
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP
```

```
Seven PROC
    EXPORT Seven
    PUSH {LR, R1, R2}

    LDR R0, =0x60000058
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Eight PROC
    EXPORT Eight
    PUSH {LR, R1, R2}

    LDR R0, =0x6000005C
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Nine PROC
    EXPORT Nine
    PUSH {LR, R1, R2}

    LDR R0, =0x60000060
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Ten PROC
    EXPORT Ten
    PUSH {LR, R1, R2}

    LDR R0, =0x60000064
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Eleven PROC
```

```
    EXPORT Eleven
    PUSH {LR, R1, R2}

    LDR R0, =0x60000068
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Twelve PROC
    EXPORT Twelve
    PUSH {LR, R1, R2}

    LDR R0, =0x6000006C
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Thirteen PROC
    EXPORT Thirteen
    PUSH {LR, R1, R2}

    LDR R0, =0x60000070
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Fourteen PROC
    EXPORT Fourteen
    PUSH {LR, R1, R2}

    LDR R0, =0x60000074
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Fifteen PROC
    EXPORT Fifteen
    PUSH {LR, R1, R2}
```

```
    LDR R0, =0x60000078
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP

Sixteen PROC
    EXPORT Sixteen
    PUSH {LR, R1, R2}

    LDR R0, =0x6000007C
    LDR R1, [R0]
    BLX R1
    POP {PC, R1, R2}
ENDP
```

Finally, the hard fault ISR is triggered when the M0 encounters a problem that it cannot recover from. For the UPSIDE chips the M0 is setup to simply send a specific network packet indicating that the processor cannot continue running. The ISR is shown in the next code block.

```
HardFault_Handler PROC
    EXPORT HardFault_Handler

    ; Write to L2 Network. First 8 32-bit slots are
    ; for the 256-bit value being written to the network.
    LDR R0, =0x53000000
    LDR R1, =0x12345678
    STR R1, [R0]

    LDR R0, =0x53000004
    LDR R1, =0x11111111
    STR R1, [R0]

    LDR R0, =0x53000008
    LDR R1, =0x24242424
    STR R1, [R0]
```

```
    LDR R0, =0x5300000C
    LDR R1, =0xAAAAAAAA
    STR R1, [R0]

    LDR R0, =0x53000010
    LDR R1, =0xCCCCCCCC
    STR R1, [R0]

    LDR R0, =0x53000014
    LDR R1, =0x97979797
    STR R1, [R0]

    LDR R0, =0x53000018
    LDR R1, =0xDEADBEEF
    STR R1, [R0]

    LDR R0, =0x5300001C
    LDR R1, =0x87654321
    STR R1, [R0]

    ; The next two addresses are the location to send
    ; the network packet. The first consists of the least
    ; significant bits and the second contains the most.
    LDR R0, =0x53000028
    LDR R1, =0x01
    STR R1, [R0]

    LDR R0, =0x5300002C
    LDR R1, =0x00
    STR R1, [R0]

    ; Finally tell the network interface to send the packet.
    LDR R0, =0x53000020
    LDR R1, =0x02
    STR R1, [R0]
ENDP
```

8.6.2 Custom Applications

The bootloader enables many types of programs to be executed on the M0 architecture. These programs can be written in assembly, C, or a combination of both. However, some care must be taken using the ARM Keil tools to ensure that the programs are executed correctly.

The M0 typically expects to boot from address 0 and execute code from that location. However, it is possible to configure Keil such that the compiled code will properly run from a different location. One way is to specify a "scatter file," which tells the compiler exactly where the compiled instructions are located as well as where the stack and the heap should go in memory. Since the compiled code is copied to the beginning of RAM, space must be allocated in the scatter file for that code, and then the stack and the heap can be configured in the rest of the memory space as desired.

A basic scatter file is shown below. This file sets up a memory region starting at address 0x60000000 that is size 0x8000 bytes, the size of the memory available to each M0 on the chips. Within that memory region the first 0x4000 bytes are reserved for compiled code which is specified by the files ending in ".o" as well as some other basic functionality. Then a separate region starting at address 0x60004000 of size 0x4000 bytes is reserved for the stack and the heap, each of which starts at an opposite end of the region growing toward each other.

```
LR_2 0x60000000 0x8000
{
```

```
    LEDSTUFF 0x60000000 0x4000
    {
        *.o (RESET, +First)
        *(InRoot$$Sections)
        startup.o (+RO)

        .ANY (+RO +RW +ZI)

        led_main.o (+RO +RW +ZI)
    }

    ARM_LIB_STACKHEAP 0x60004000 EMPTY 0x4000
    {}
  }
```

Other scatter files can be specified with different configurations as long as there is enough space at the beginning of the memory space for the compiled code as well as enough room for the stack and the heap. The stack and heap can be configured separately if desired, too.

Now, in order to create a custom program, certain interrupts must be configured in the vector table. The first value in the table is the initial stack pointer. This value must correspond to the location of the start of the stack in the scatter file, which in this case begins at the very end of the memory area. A new reset handler must also be established so that when the bootloader jumps to address 0x60000004 and starts executing, the bootloader runs the code specified here since that is the address where the reset handler will be placed in memory.

Then, any external hardware interrupts can be specified as well. As described in Section 8.5, multiple interrupts can be triggered by the hardware to be handled

by this custom software. They are all listed in the external interrupts of the startup assembly code for any Keil project that wants to use them, and they are shown below.

```
PRESERVE8
THUMB

AREA    RESET, DATA, READONLY ; First 32 WORDS is VECTOR TABLE
EXPORT  __Vectors

__Vectors
DCD 0x60008000
DCD Reset_Handler
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0
DCD 0

; External Interrupts
DCD UART_Handler
DCD DMA_RD
DCD DMA_WR
DCD NETW_RD_DATA
DCD NETW_RD_ACK
DCD NETW_RD_CONTROL
DCD NETW_WR
DCD SPI_READ
DCD SPI_WRITE
DCD 0
DCD 0
DCD 0
DCD 0
```

```
DCD 0
DCD 0
DCD 0

AREA |.text|, CODE, READONLY

Reset_Handler PROC
    EXPORT  Reset_Handler             [WEAK]

    IMPORT  __main
    LDR R0, =__main
    BX      R0
ENDP

DMA_WR PROC
    EXPORT DMA_WR
    IMPORT DMA_WR_ISR
    PUSH {R0,R1,R2,LR}

    BL DMA_WR_ISR
    POP {R0,R1,R2,PC}
ENDP

DMA_RD PROC
    EXPORT DMA_RD
    IMPORT DMA_RD_ISR
    PUSH {R0,R1,R2,LR}

    BL DMA_RD_ISR
    POP {R0,R1,R2,PC}
ENDP

NETW_WR PROC
    EXPORT NETW_WR
    IMPORT NETW_WR_ISR
    PUSH { R0,R1,R2,LR}

    BL NETW_WR_ISR
    POP {R0,R1,R2,PC}
ENDP
```

```
NETW_RD_DATA PROC
    EXPORT NETW_RD_DATA
    IMPORT NETW_RD_DATA_ISR
    PUSH {R0,R1,R2,LR}

    BL NETW_RD_DATA_ISR
    POP {R0,R1,R2,PC}
ENDP

NETW_RD_ACK PROC
    EXPORT NETW_RD_ACK
    IMPORT NETW_RD_ACK_ISR
    PUSH {R0,R1,R2,LR}

    BL NETW_RD_ACK_ISR
    POP {R0,R1,R2,PC}
ENDP

NETW_RD_CONTROL PROC
    EXPORT NETW_RD_CONTROL
    IMPORT NETW_RD_CONTROL_ISR
    PUSH {R0,R1,R2,LR}

    BL NETW_RD_CONTROL_ISR
    POP {R0,R1,R2,PC}
ENDP

UART_Handler PROC
    EXPORT UART_Handler
    IMPORT UART_ISR
    PUSH {R0,R1,R2,LR}

    BL UART_ISR
    POP {R0,R1,R2,PC}
ENDP

SPI_READ PROC
    EXPORT SPI_READ
    IMPORT SPI_READ_ISR
    PUSH {R0,R1,R2,LR}

    BL SPI_READ_ISR
```

```
    POP {R0,R1,R2,PC}
ENDP

SPI_WRITE PROC
    EXPORT SPI_WRITE
    IMPORT SPI_WRITE_ISR
    PUSH {R0,R1,R2,LR}

    BL SPI_WRITE_ISR
    POP {R0,R1,R2,PC}
ENDP

END
```

The assembly code shown above is configured to run a function from a separate C file in the Keil project for each interrupt that is triggered. The reset handler calls the function main() so that the C code can execute as intended, whereas the other interrupts call their respective C functions to perform the duties they are configured to do. For example, at the bottom of the code, the SPI_WRITE interrupt is configured to call the C function SPI_WRITE_ISR.

Remember, though, that all these interrupts are indirectly called from the original bootloader assembly code. The M0 still uses the original vector table for its interrupts, so when an interrupt is called the bootloader reads the corresponding vector table entry from RAM. Then it branches to that address's code and the custom interrupt runs, taking care of the stack and the link register along the way so the original program execution can continue once the ISR has finished running.

The following sections go into a bit more detail regarding how to program the SPI peripheral, the DMA controller, and the NoC interface. Some brief code snippets are

provided to show examples.

8.6.2.1 SPI

There are a few important registers that can be used to configure the SPI, and they are shown below. The instruction address is used to program the instruction and address sent on the SPI MOSI line (see Section 8.3), and the write value is the value to be written if the instruction is a write instruction. The write value consists of four separate bytes, each one going to an independent SPI MOSI line off the chip to four separate external SPI devices that all share the same instruction and address. Once an instruction is done being executed, the read register is used to read the value back from the SPI. This 32-bit register stores the four separate bytes from four devices with a shared instruction and address value, just as is done for write instructions. Two registers can be used to clear the interrupts for reading and writing because the peripheral triggers those interrupts once a read or write operation is completed. Finally, the SPI clock signal (SCK) can be configured by programming the counter for the clock divider feeding the SPI module.

```
#define SPI_INSTRUCTION_ADDRESS (0x58000000)
#define SPI_WRITE_VALUE (0x58000004)
#define SPI_READ (0x58000008)
#define SPI_CLEAR_READ_IRQ (0x5800000C)
#define SPI_CLEAR_WRITE_IRQ (0x58000010)
#define SPI_CLK_DIVIDER (0x58000014)
```

Programming the SPI clock divider is very simple. The following example shows how to set the counter limit to 3.

```
*(volatile unsigned int*)SPI_CLK_DIVIDER = 3;
```

The following code writes one byte to each of four separate SPI devices at once. When the instruction address register is written, the SPI starts working on sending the value. The loop in the code waits until the ISR corresponding to the SPI write being done is run. When the SPI is done writing, its write interrupt is triggered and the interrupt updates the global flag SPI_WRITE_DONE. Then the code below sets that flag back to its default value for the next time the SPI is written to. It is also possible to perform other processing while waiting for the SPI write to complete rather than looping and waiting in this manner.

```
*(volatile unsigned int*)SPI_WRITE_VALUE = 0x9876CFDA;
*(volatile unsigned int*)SPI_INSTRUCTION_ADDRESS = 0x0234abcd;

// Then wait until the write has finished and the ISR
// clears the flag.
while (SPI_WRITE_DONE == 0) {} SPI_WRITE_DONE = 0;
```

The next block shows the contents of the ISR handling a write completion by clearing the interrupt and setting a flag saying the write is done.

```
void SPI_WRITE_ISR()
{
    *(volatile unsigned int*)SPI_CLEAR_WRITE_IRQ = 1;
    SPI_WRITE_DONE = 1;
}
```

Reading from the SPI devices is done in a similar manner. In the following block it is assumed that the variable “result” has already been declared to be an unsigned integer, and it stores the four bytes read from the four SPI devices.

```
// Read from the SPI device.
*(volatile unsigned int*)SPI_INSTRUCTION_ADDRESS = 0x03ccbbaa;
// Then wait until the read is done before trying to read.
while (SPI_READ_DONE == 0) {} SPI_READ_DONE = 0;
// Get the actual values.
result = *(volatile unsigned int*)SPI_READ;
```

The ISR for reading from the SPI is set up just as the write ISR is for this example.

```
void SPI_READ_ISR()
{
    *(volatile unsigned int*)SPI_CLEAR_READ_IRQ = 1;
    SPI_READ_DONE = 1;
}
```

Before the byte reads and writes described above are done, the mode must be set to enable byte mode as shown below. When writing the mode, the SPI device writes the instruction (0x01) and the first byte of the address. That first byte of the address must contain the mode value that is to be written. In the example below the mode value is 0x00 which corresponds to turning on byte mode for the peripheral.

```
// Test writing the mode for the SPI memory.
*(volatile unsigned int*)SPI_INSTRUCTION_ADDRESS = 0x01000000;
while (SPI_WRITE_DONE == 0) {} SPI_WRITE_DONE = 0;
```

The mode can also be read by the SPI peripheral in a similar manner to how normal values can be read except that the instruction value is 0x05.

```
// Test reading the mode for the SPI memory.
*(volatile unsigned int*)SPI_INSTRUCTION_ADDRESS = 0x05000000;
while (SPI_READ_DONE == 0) {} SPI_READ_DONE = 0;

result = *(volatile unsigned int*)SPI_READ;
```

Reading and writing the mode triggers the same ISRs as reading and writing a byte from/to the external device, so other computations can be again performed while waiting for the read or write to finish.

8.6.2.2 DMA

The DMA controller addresses data in terms of 256-bit words, so the addresses the M0 uses must be converted to these 256-bit units by dividing the byte location by 32. Transfers can also only occur at proper aligned edges of these 256-bit word locations. Writing to main memory is shown below. The write goes from address 0 to address 1, and there are low and high bits because the address has 40 bits. So the low address is the lowest 32 bits and the high address contains the most significant 8 bits of the full address.

To start a write to main memory using the DMA controller, four registers are programmed to set the address range in main memory. The starting and ending addresses in the SRAM for the M0 must also be set. In the following example, four 256-bit words are written from the RAM address 0x60001200 onward into main memory. Once all the addresses are set, the transfer type is specified for whether the DMA writes to main memory or reads from main memory. The DMA write ISR is set to update the flag for whether the write has completed.

```
// First write to main memory (MM). All these addresses are inclusive
// so the last 256-bit word is also included.
*(volatile unsigned int*)0x57000004 = 132; // MM address 0 low
*(volatile unsigned int*)0x57000008 = 0; // MM address 0 high
```

```
*(volatile unsigned int*)0x5700000C = 136; // MM address 1 low
*(volatile unsigned int*)0x57000010 = 0; // MM address 1 high

// SRAM address 0
*(volatile unsigned int*)0x57000014 =
        (0x60001200 - 0x60000000) / 32;
// SRAM address 1
*(volatile unsigned int*)0x57000018 =
        (0x60001200 + 8*4 * 4 - 0x60000000) / 32;

// transfer type (2'b00 for read from MM, 2'b10 for write to MM)
*(volatile unsigned int*)0x5700001C = 2;

// Wait for the transfer to complete.
while (DMA_WR_DONE == 0) {} DMA_WR_DONE = 0;
```

Below is the implementation of the DMA write ISR.

```
void DMA_WR_ISR()
{
    // Clear the interrupt.
    *(volatile unsigned int*)DMA_WRITE_CLEAR_INTERRUPT = 1;

    // Change the flag.
    DMA_WR_DONE = 1;
}
```

Reading from the main memory using the DMA proceeds in a similar manner except that the transfer type is changed so that it is a read, and the read ISR is triggered to update the read flag instead of the write flag. Then the values can be used however the programmer chooses.

8.6.2.3 NoC

Reading data from the network is accomplished by implementing the appropriate ISR for the NoCs. The following code shows an example of an ISR that can be used to read an incoming data packet from the control network. This ISR shown below does not do anything with the values read from memory, but the values could easily be used for various purposes. Once the incoming data is read and used, the ISR can tell the NoC interface to turn off its interrupt.

```
void NETW_RD_DATA_ISR()
{
    unsigned int result;

    result = *((volatile unsigned int*)NETWORK_READ_P0_ADDRESS);
    result = *((volatile unsigned int*)NETWORK_READ_P1_ADDRESS);
    result = *((volatile unsigned int*)NETWORK_READ_P2_ADDRESS);
    result = *((volatile unsigned int*)NETWORK_READ_P3_ADDRESS);
    result = *((volatile unsigned int*)NETWORK_READ_P4_ADDRESS);
    result = *((volatile unsigned int*)NETWORK_READ_P5_ADDRESS);
    result = *((volatile unsigned int*)NETWORK_READ_P6_ADDRESS);
    result = *((volatile unsigned int*)NETWORK_READ_P7_ADDRESS);

    // Clear the interrupt.
    *(volatile unsigned int*)NETWORK_READ_DATA_CLEAR_INTERRUPT = 1;
}
```

The constants are defined in the following way for that ISR.

```
#define NETWORK_READ_P0_ADDRESS (0x54000000)
#define NETWORK_READ_P1_ADDRESS (0x54000004)
#define NETWORK_READ_P2_ADDRESS (0x54000008)
#define NETWORK_READ_P3_ADDRESS (0x5400000C)
#define NETWORK_READ_P4_ADDRESS (0x54000010)
#define NETWORK_READ_P5_ADDRESS (0x54000014)
#define NETWORK_READ_P6_ADDRESS (0x54000018)
#define NETWORK_READ_P7_ADDRESS (0x5400001C)
```

```
#define NETWORK_READ_DATA_CLEAR_INTERRUPT (0x54000028)
```

Similar ISRs can be implemented for reading a network acknowledgement packet or a network control packet. The register addresses for the incoming data are the same, but the different types of packets can be handled different ways by creating multiple ISRs, one for each type.

On the other hand, writing to the network is accomplished by first setting the eight 32-bit words to be placed in the packet. Then the internal address for the processing unit is written, the address of the device on the network is specified, and finally the type of transfer is set. This transfer type (data or command) can be used to tell the processing unit which type of packet is incoming so it knows how to react. The loop at the end follows a similar convention as the other ISRs shown earlier in this chapter because it waits until a flag is set saying that the write is done before proceeding. Then it clears the flag again so it is ready for the next network write.

```
// Set the data in the packet.
*((volatile unsigned int*)0x53000000) = 3493;
*((volatile unsigned int*)0x53000004) = 3615;
*((volatile unsigned int*)0x53000008) = 7001;
*((volatile unsigned int*)0x5300000C) = 7500;
*((volatile unsigned int*)0x53000010) = 0;
*((volatile unsigned int*)0x53000014) = 0;
*((volatile unsigned int*)0x53000018) = 0;
*((volatile unsigned int*)0x5300001C) = 0;

// Internal address used by the processing unit.
*((volatile unsigned int*)0x5300002C) = 0;

// Set horizontal (X) and vertical (Y) addresses.
```

```
// X is least significant 5 bits and Y is next 3 bits.
// Here sending to (X, Y) = (2, 0)
*((volatile unsigned int*)0x53000030) = 2 + (0 << 5);

// Specify data or command - 0x20 is data and 0x24 is command.
// Setting this register also sends the packet.
*((volatile unsigned int*)0x53000024) = 1;

// Wait for packet to be written.
while (NETW_WR_DONE == 0) {} NETW_WR_DONE = 0;
```

Bibliography

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 4, pp. 114–117, 1965.

[2] G. E. Moore *et al.*, "Progress in Digital Integrated Electronics," in *Electron Devices Meeting*, vol. 21, 1975, pp. 11–13.

[3] A. Szalay and J. Gray, "2020 Computing: Science in an Exponential World," *Nature*, vol. 440, no. 7083, pp. 413–414, Mar. 2006.

[4] D. Geer, "Chip Makers Turn to Multicore Processors," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[5] B. Smith, "ARM and Intel Battle Over the Mobile Chip's Future," *IEEE Computer*, vol. 41, no. 5, pp. 15–18, 2008.

[6] S. Herculano-Houzel, "Scaling of Brain Metabolism with a Fixed Energy Budget per Neuron: Implications for Neuronal Activity, Plasticity and Evolution," *PLoS ONE*, vol. 6, no. 3, p. e17514, Mar. 2011.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[9] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning Hierarchical Features for Scene Labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1915–1929, 2013.

[10] J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler, "Joint Training of a Convolutional Network and a Graphical Model for Human Pose Estimation," in *Advances in Neural Information Processing Systems*, 2014, pp. 1799–1807.

[11] S. Ji, W. Xu, M. Yang, and K. Yu, "3D Convolutional Neural Networks for Human Action Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 1, pp. 221–231, 2013.

[12] J. Craley, T. S. Murray, D. R. Mendat, and A. G. Andreou, "Action Recognition Using Micro-Doppler Signatures and a Recurrent Neural Network," in *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, March 2017, pp. 1–5.

[13] T. S. Murray, D. R. Mendat, P. O. Pouliquen, and A. G. Andreou, "The Johns

Hopkins University Multimodal Dataset for Human Action Recognition," in *Proceedings of SPIE: Radar Sensor Technology XIX; and Active and Passive Signatures VI*, May 2015, pp. 79–94.

[14] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to End Learning for Self-Driving Cars," *arXiv Preprint arXiv:1604.07316*, 2016.

[15] K. D. Fischl, G. Tognetti, D. R. Mendat, G. Orchard, J. Rattray, C. Sapsanis, L. F. Campbell, L. Elphage, T. E. Niebur, A. Pasciaroni, V. E. Rennoll, H. Romney, S. Walker, P. O. Pouliquen, and A. G. Andreou, "Neuromorphic Self-driving Robot with Retinomorphic Vision and Spike-based Processing/Closed-loop Control," in *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, March 2017, pp. 1–6.

[16] C. Mead, "Neuromorphic Electronic Systems," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, 1990.

[17] A. S. Cassidy, J. Georgiou, and A. G. Andreou, "Design of Silicon Brains in the Nano-CMOS Era: Spiking Neurons, Learning Synapses and Neural Architecture Optimization," *Neural Networks*, vol. 45, pp. 4–26, 2013.

[18] F. C. Morabito, A. G. Andreou, and E. Chicca, "Neuromorphic Engineering: From Neural Systems to Brain-Like Engineered Systems," *Neural Networks*, vol. 45, 2013.

[19] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith, "Nengo: a Python Tool for Building Large-Scale Functional Brain Models," *Frontiers in Neuroinformatics*, vol. 7, 2013.

[20] C. Eliasmith and C. H. Anderson, *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. MIT Press, 2004.

[21] S. Furber, F. Galluppi, S. Temple, and L. Plena, "The SpiNNaker Project," *Proceedings of the IEEE*, pp. 1–17, 2014.

[22] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the SpiNNaker System Architecture," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2454–2467, Dec. 2013.

[23] X. Jin, M. Lujan, L. A. Plana, S. Davies, S. Temple, and S. B. Furber, "Modeling Spiking Neural Networks on SpiNNaker," *Computing in Science & Engineering*, vol. 12, no. 5, pp. 91–97, 2010.

[24] F. Galluppi, S. Davies, S. Furber, T. Stewart, and C. Eliasmith, "Real Time On-Chip Implementation of Dynamical Systems with Spiking Neurons," in *The 2012 International Joint Conference on Neural Networks (IJCNN)*, June 2012, pp. 1–8.

[25] A. Mundy, J. Knight, T. C. Stewart, and S. Furber, "An Efficient SpiNNaker

Implementation of the Neural Engineering Framework," in *2015 International Joint Conference on Neural Networks (IJCNN)*, July 2015, pp. 1–8.

[26] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany," *arXiv.org*, Dec. 2014.

[27] C. Kohn, "Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices," Tech. Rep. XAPP1159, Jan. 2013.

[28] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell, "Programming the Adapteva Epiphany 64-Core Network-on-Chip Coprocessor," in *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2014, pp. 984–992.

[29] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.

[30] A. S. Cassidy, R. Alvarez-Icaza, F. Akopyan, J. Sawada, J. V. Arthur, P. A. Merolla, P. Datta, M. G. Tallada, B. Taba, A. Andreopoulos, A. Amir, S. K. Esser, J. Kusnitz, R. Appuswamy, C. Haymes, B. Brezzo, R. Moussalli,

R. Bellofatto, C. Baks, M. Mastro, K. Schleupen, C. E. Cox, K. Inoue, S. Millman, N. Imam, E. McQuinn, Y. Y. Nakamura, I. Vo, C. Guo, D. Nguyen, S. Lekuch, S. Asaad, D. Friedman, B. L. Jackson, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "Real-Time Scalable Cortical Computing at 46 Giga-Synaptic OPS/Watt with 100 Speedup in Time-to-Solution and 100,000 Reduction in Energy-to-Solution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE Press, Nov. 2014.

[31] J. Backus, "Can Programming be Liberated from the von Neumann Style?: A Functional Style and its Algebra of Programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[32] A. S. Cassidy and A. G. Andreou, "Beyond Amdahl's Law: An Objective Function That Links Multiprocessor Performance Gains to Delay and Energy," *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1110–1126, 2012.

[33] N. Chater, J. B. Tenenbaum, and A. Yuille, "Probabilistic Models of Cognition: Conceptual Foundations," *Trends in Cognitive Sciences*, vol. 10, no. 7, pp. 287–291, 2006.

[34] C. Kemp and J. B. Tenenbaum, "The Discovery of Structural Form," *Proceedings of the National Academy of Sciences*, vol. 105, no. 31, pp. 10 687–10 692, 2008.

[35] K. P. Körding and D. M. Wolpert, "Bayesian Decision Theory in Sensorimotor Control," *Trends in Cognitive Sciences*, vol. 10, no. 7, pp. 319–326, 2006.

[36] K. P. Kording and D. M. Wolpert, "Bayesian Integration in Sensorimotor Learning," *Nature*, vol. 427, no. 6971, p. 244, 2004.

[37] M. O. Ernst and M. S. Banks, "Humans Integrate Visual and Haptic Information in a Statistically Optimal Fashion," *Nature*, vol. 415, no. 6870, pp. 429–433, 2002.

[38] D. Alais and D. Burr, "The Ventriloquist Effect Results from Near-Optimal Bimodal Integration," *Current Biology*, vol. 14, no. 3, pp. 257–262, 2004.

[39] B. J. Fischer and J. L. Peña, "Owl's Behavior and Neural Representation Predicted by Bayesian Inference," *Nature Neuroscience*, vol. 14, no. 8, pp. 1061–1066, 2011.

[40] M. L. Platt and P. W. Glimcher, "Neural Correlates of Decision Variables in Parietal Cortex," *Nature*, vol. 400, no. 6741, p. 233, 1999.

[41] J. I. Gold and M. N. Shadlen, "Neural Computations that Underlie Decisions about Sensory Stimuli," *Trends in Cognitive Sciences*, vol. 5, no. 1, pp. 10–16, 2001.

[42] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*, 1st ed. The MIT Press, Jul. 2009.

BIBLIOGRAPHY

[43] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[44] J. Pearl, "Bayesian Networks," *MIT Encyclopedia of the Cognitive Sciences*, 2001.

[45] K. P. Murphy, *Machine Learning: a Probabilistic Perspective*. MIT Press, Sep. 2013.

[46] ——, "Dynamic Bayesian Networks: Representation, Inference and Learning," Ph.D. dissertation, University of California Berkeley, 2002.

[47] C. Andrieu, N. De Freitas, A. Doucet, and M. Jordan, "An Introduction to MCMC for Machine Learning," *Machine Learning*, vol. 50, pp. 5–43, 2003.

[48] J. R. Norris, *Markov Chains*, ser. Statistical and Probabilistic Mathematics. Cambridge University Press, 1998.

[49] G. Casella and E. I. George, "Explaining the Gibbs Sampler," *The American Statistician*, vol. 46, no. 3, pp. 167–174, 1992.

[50] D. A. Levin, Y. Peres, and E. L. Wilmer, *Markov Chains and Mixing Times*. American Mathematical Society, Oct. 2008.

[51] D. Pecevski, L. Buesing, and W. Maass, "Probabilistic Inference in General Graphical Models through Sampling in Stochastic Networks of Spiking Neurons," *PLOS Computational Biology*, vol. 7, no. 12, p. e1002294, Dec. 2011.

[52] L. Buesing, J. Bill, B. Nessler, and W. Maass, "Neural Dynamics as Sampling: A Model for Stochastic Computation in Recurrent Networks of Spiking Neurons," *PLOS Computational Biology*, vol. 7, no. 11, p. e1002211, Nov. 2011.

[53] K. P. Murphy, "The Bayes Net Toolbox for Matlab," *Computing Science and Statistics*, 2001.

[54] J. Navaridas, M. Luján, J. Miguel-Alonso, L. A. Plana, and S. Furber, "Understanding the Interconnection Network of SpiNNaker," in *Proceedings of the 23rd International Conference on Supercomputing*. ACM, 2009, pp. 286–295.

[55] D. R. Mendat, S. Chin, S. Furber, and A. G. Andreou, "Markov Chain Monte Carlo Inference on Graphical Models Using Event-Based Processing on the SpiNNaker Neuromorphic Architecture," in *2015 49th Annual Conference on Information Sciences and Systems (CISS)*, March 2015, pp. 1–6.

[56] ——, "Neuromorphic Sampling on the SpiNNaker and Parallella Chip Multiprocessors," in *2016 IEEE 7th Latin American Symposium on Circuits and Systems (LASCAS)*, Feb 2016, pp. 399–402.

[57] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin, "Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees," in *International Conference on Artificial Intelligence and Statistics*, 2011, pp. 324–332.

[58] I. A. Beinlich, H. J. Suermondt, R. M. Chavez, and G. F. Cooper, *The ALARM*

Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks. Springer, 1989.

[59] M. Scutari, "Learning Bayesian Networks with the bnlearn R Package," *Journal of Statistical Software*, vol. 35, no. 3, pp. 1–22, 2010. [Online]. Available: http://www.jstatsoft.org/v35/i03/

[60] ——. (2015) Bayesian Network Repository. [Online]. Available: http://www.bnlearn.com/bnrepository/

[61] D. J. Spiegelhalter and R. G. Cowell, "Learning in Probabilistic Expert Systems," *Bayesian Statistics*, vol. 4, pp. 447–465, 1992.

[62] D. C. Knill and D. Kersten, "Apparent Surface Curvature Affects Lightness Perception," *Nature*, vol. 351, no. 6323, p. 228, 1991.

[63] J. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[64] A. Munshi, "The OpenCL specification," Tech. Rep. 1.0, Oct. 2009.

[65] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza, E. McQuinn, B. Shaw, N. Pass, and D. S. Modha, "Cognitive Computing Programming Paradigm: A

Corelet Language for Composing Networks of Neurosynaptic Cores," in *Proceedings of the 2013 International Joint Conference on Neural Networks (IJCNN)*, 2013, pp. 1–10.

[66] R. Preissl, T. M. Wong, P. Datta, M. Flickner, R. Singh, S. K. Esser, W. P. Risk, H. D. Simon, and D. S. Modha, "Compass: A Scalable Simulator for an Architecture for Cognitive Computing," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society, Nov. 2012, pp. 1–11.

[67] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing," *arXiv.org*, Mar. 2016.

[68] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7, pp. 436–444, May 2015.

[69] A. G. Andreou, A. A. Dykman, K. D. Fischl, G. Garreau, D. R. Mendat, G. Orchard, A. S. Cassidy, P. Merolla, J. Arthur, R. Alvarez-Icaza, B. L. Jackson, and D. S. Modha, "Real-time Sensory Information Processing Using the TrueNorth Neurosynaptic System," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, p. 2911.

[70] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *arXiv preprint arXiv:1301.3781*, 2013.

[71] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119. [Online]. Available: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[72] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic Regularities in Continuous Space Word Representations," in *NAACL HLT*, vol. 13, 2013, pp. 746–751.

[73] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A Neural Probabilistic Language Model," *Journal of Machine Learning Research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[74] O. Levy and Y. Goldberg, "Dependency-Based Word Embeddings," in *ACL*. Citeseer, 2014, pp. 302–308.

[75] B. R. Gaines *et al.*, "Stochastic Computing Systems," *Advances in Information Systems Science*, vol. 2, no. 2, pp. 37–172, 1969.

[76] A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2s, p. 92, 2013.

[77] European Machine Vision Association and others, "Standard for Characterization of Image Sensors and Cameras," *EMVA Standard*, vol. 1288, 2010.

[78] J. G. Harris and Y.-M. Chiang, "Nonuniformity Correction of Infrared Image Sequences Using the Constant-Statistics Constraint," *IEEE Transactions on Image Processing*, vol. 8, no. 8, pp. 1148–1151, 1999.

[79] D. R. Mendat, J. E. West, S. Ramenahalli, E. Niebur, and A. G. Andreou, "Audio-Visual Beamforming with the Eigenmike Microphone Array an Omni-Camera and Cognitive Auditory Features," in *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, March 2017, pp. 1–4.

[80] L. Itti, C. Koch, and E. Niebur, "A Model of Saliency-Based Visual Attention for Rapid Scene Analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 11, pp. 1254–1259, Nov 1998.

[81] A. F. Russell, S. Mihalaş, R. von der Heydt, E. Niebur, and R. Etienne-Cummings, "A Model of Proto-Object Based Saliency," *Vision Research*, vol. 94, pp. 1–15, 2014.

[82] J. L. Molin, A. F. Russell, S. Mihalas, E. Niebur, and R. Etienne-Cummings, "Proto-Object Based Visual Saliency Model with a Motion-Sensitive Channel,"

in *2013 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Oct 2013, pp. 25–28.

[83] J. L. Molin, R. Etienne-Cummings, and E. Niebur, "How is Motion Integrated into a Proto-Object Based Visual Saliency Model?" in *2015 49th Annual Conference on Information Sciences and Systems (CISS)*, March 2015, pp. 1–6.

[84] J. L. Molin and R. Etienne-Cummings, "Live Demonstration: Real-Time Implementation of a Proto-Object-Based Dynamic Visual Saliency Model," in *2015 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Oct 2015, p. 1.

[85] L. Shestopalova, T. M. Bőhm, A. Bendixen, A. G. Andreou, J. Georgiou, G. Garreau, B. Hajdu, S. L. Denham, and I. Winkler, "Do Audio-Visual Motion Cues Promote Segregation of Auditory Streams?" *Frontiers in Neuroscience*, vol. 8, 2014.

[86] J. Georgiou, P. Pouliquen, A. Cassidy, G. Garreau, C. Andreou, G. Stuarts, C. d'Urbal, A. G. Andreou, S. Denham, T. Wennekers *et al.*, "A Multimodal-Corpus Data Collection System for Cognitive Acoustic Scene Analysis," in *Information Sciences and Systems (CISS), 2011 45th Annual Conference on*. IEEE, 2011, pp. 1–6.

[87] J. Meyer and G. Elko, "A Highly Scalable Spherical Microphone Array Based on an Orthonormal Decomposition of the Soundfield," in *Acoustics, Speech, and*

Signal Processing (ICASSP), 2002 IEEE International Conference on, vol. 2. IEEE, 2002, pp. II–1781–II–1784.

[88] "Eigenmike Microphone Array." [Online]. Available: http://www.mhacoustics.com

[89] "VisiSonics Audiovisual Camera." [Online]. Available: http://www.visisonics.com

[90] M. C. Chan, "Theory and Design of Higher Order Sound Field Recording," *Department of Engineering, FEIT, ANU, Honours Thesis*, 2003.

[91] E. G. Williams, *Fourier Acoustics: Sound Radiation and Nearfield Acoustical Holography.* Academic Press, 1999.

[92] S. O. Petersen, "Localization of Sound Sources Using 3D Microphone Array," *University of Southern Denmark, MS Thesis*, 2004.

[93] Z. Li and R. Duraiswami, "Flexible and Optimal Design of Spherical Microphone Arrays for Beamforming," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 15, no. 2, pp. 702–714, 2007.

[94] M. Taylor, "Cubature for the Sphere and the Discrete Spherical Harmonic Transform," *SIAM Journal on Numerical Analysis*, vol. 32, no. 2, pp. 667–670, 1995.

[95] "Boost C++ Libraries." [Online]. Available: http://www.boost.org

[96] "Unconventional Processing of Signals for Intelligent Data Exploitation (UPSIDE)," https://www.darpa.mil/program/unconventional-processing-of-signals-for-intelligent-data-exploitation, Accessed: October 12, 2017.

[97] A. G. Andreou, T. Figliolia, K. Sanni, T. S. Murray, G. Tognetti, D. R. Mendat, J. L. Molin, M. Villemur, P. O. Pouliquen, P. Julian, R. Etienne-Cummings, and I. Doxas, "Bio-inspired System Architecture for Energy Efficient, BIGDATA Computing with Application to Wide Area Motion Imagery," in *2016 IEEE 7th Latin American Symposium on Circuits Systems (LASCAS)*, Feb 2016, pp. 1–6.

[98] J. Yiu, *The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors*. Academic Press, 2015.

[99] *SPI Block Guide V03.06*, Motorola, 2003.

[100] *1Mbit SPI Serial SRAM with SDI and SQI Interface*, Microchip, 2015.

Vita

Daniel Richard Mendat was born during 1988 in New Jersey. He received a B.S. degree in Electrical and Computer Engineering with a double major in Computer Science from Rutgers University in 2010. He enrolled in the Electrical and Computer Engineering Ph.D. program at Johns Hopkins University that same year, where he completed an M.S. degree in Electrical and Computer Engineering in 2011. He received the Johns Hopkins University Electrical and Computer Engineering (JHU ECE) Department Fellowship as well as the JHU ECE Bodmer Fellowship, both in 2010-2011. He was later supported by the JHU Applied Physics Laboratory Fellowship during 2013-2017. His research focuses on neuromorphic hardware/software architectures for performing unconventional parallel processing.