APPROVAL SHEET

Title of Dissertation: A General Purpose Neural Processor

Name of Candidate: David Jerome Mountain
Doctor of Philosophy, 2017

Dissertation and Abstract Approved: _____

Anupam Joshi
Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

ABSTRACT

David J. Mountain, PhD. Dissertation 2017

Dissertation directed by: Dr. Anupam Joshi, Professor
Department of Computer Science and
Electrical Engineering

Computer applications are evolving from traditional scientific and numerical calculations, to a more diverse set of uses including speech recognition, robotics, and analytics. This has created a fertile environment for the investigation of non-traditional programming approaches and models of computing inspired by neuroscience, often termed neuromorphic computing. Neural nets have emerged as one of the primary neuromorphic computing approaches; von Neumann architectures, conceived for scientific computing applications are not optimized for neural nets [1].

This research focuses on developing a general purpose computer architecture optimized for neural net based applications. The architecture is useful for a variety of learning algorithms, and is evaluated across a spectrum of potential applications. Both traditional and emerging technologies are explored, with trade-offs being made based on the most important system level metrics.

A General Purpose Neural Processor

by

David Jerome Mountain

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland, Baltimore County in partial fulfillment
of the requirements for the status of
Doctor of Philosophy
2017

DEDICATIONS

To my wife, Diane, who has been my love and my friend for many years. Your support has inspired me ever since we first met.

To my children, Michael, Molly, Charlotte, and Jack. My loftiest goal is to be a good father to you.

To my parents, John and Patricia, who always emphasized the value of education.

To Mike, Jim, Tom, Rich, and Bill. Thank you for your brotherly love.

To my oldest brother Patrick (December 1, 1957 - March 19, 2015). My pioneer in life. You are missed, but remembered with joy.

ACKNOWLEDGMENTS

Many thanks to my advisor, Dr. Anupam Joshi, for agreeing to help an atypical graduate student accomplish a lifelong goal. I would like to express my gratitude to my co-advisor, Dr. Tarek Taha, for his support and guidance. I also want to thank my committee for their many helpful discussions and feedback during the course of my research.

I am fortunate to have been supported by a sizable number of collaborators; I particularly wish to acknowledge the help of Dr. Christopher Krieger and Mr. Mark McLean in my research explorations.

Table of Contents

List of Tables

List of Figures

List of Abbreviations and Acronyms

0T1M	Zero Transistors One Memristor
1T1M	One Transistor One Memristor
2D	Two Dimensional
3D	Three Dimensional
A2A	All To All
ADC	Analog to Digital Converter
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
BP	Backpropagation
CLA	Concurrent Learning Algorithm
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DARPA	Defense Advanced Research Projects Agency
DPE	Dot Product Engine
DV	Differential Voltage
DZ	Differential 1/High Z
eDRAM	embedded Dynamic Random Access Memory
FET	Field Effect Transistor
FIFO	First In First Out
FPGA	Field Programmable Gate Array
Gbps	Giga bits per second
GP	General Purpose
GPU	Graphics Processing Unit
HP	Hewlett-Packard
IARPA	Intelligence Advanced Research Projects Agency
IBM	International Business Machines
L	Length
LPD	Limited Purpose Design
MACC	Multiply Accumulate
MB	Mega Byte
MNIST	Mixed National Institute of Standards and Technology database
NIST	National Institute of Standards and Technology
NMC	Neuromorphic Computing
NN	Neural Net
NOC	Network On Chip
NSA	National Security Agency
PFET	Positive-charge Field Effect Transistor
SNN	Spiking Neural Net
SPD	Special Purpose Design
SPICE	Simulation Program with Integrated Circuit Emphasis
SRAM	Static Random Access Memory

SV	Single-ended Voltage
T-gate	Transmission Gate
T/W	Throughput per Watt
T/A	Throughput per Area
TaOx	Tantalum Oxide
TGN	Threshold Gate Network
W	Width
W/A	Watts per Area

Chapter 1: Introduction

1.1 Motivation

As a result of more than 50 years of technological advancement exemplified by Moore's Law and Dennard scaling [2,3], computers have become a ubiquitous feature of modern life. In contrast to their original use for tasks requiring precise scientific calculations, today computers are used for an incredible variety of applications, including social media, pictures and video, and speech processing. A smartphone, available for $500 or less, is expected to be capable of "computing" in a multitude of ways. Extensive use of cognitive computing is anticipated [4].

These new and emerging applications do not optimally map themselves to traditional computing paradigms developed for very precise scientific calculations [1]. In the existing paradigm, a pre-defined algorithm, implemented using a computer language that completely specifies the sequence of calculations to be performed, is used [5]. However, this approach is less than optimal when the algorithm is not precisely specified and therefore the exact set of calculations are not known. For these newer applications, researchers are looking to the human brain for inspiration, an area of exploration that is termed neuromorphic computing (NMC) [6]. The

understanding of how our brains adapt and adjust to the environment has led to significant growth in the use of artificial neural nets for these emerging applications [7–9].

These approaches take advantage of the availability of large amounts of data (ironically, from the same technological revolution that created the need for these applications in the first place), to "train" the computer to learn the correct answer, as opposed to programming the computer to calculate the correct answer [10, 11]. One of the more popular approaches uses neural networks. This approach is inspired by the functionality of neurons in the brain; Figure 1.1 depicts the mathematical operation of a neuron in this approach. Each input to the neuron is weighted, which provides a mechanism for assigning levels of importance to the inputs. The weighted inputs are then summed and evaluated. A typical evaluation function is shown, where the output equals "1" if the value of the multiply-accumulate (MACC) operation exceeds a threshold. Another term for this design is a Threshold Gate Network (TGN).

A large number of these neurons can be connected in a hierarchy, which enables the neural net to abstract information from the provided data. These deep neural nets have proven to be very effective, particularly in the area of image classification [7–9, 12, 13].

Just as the nature of programming computers for these new applications is significantly altered, traditional computer architectures are not optimal for implementing neural nets or other new models of computing. The traditional von Neumann architecture contains both the program and data stored in a memory that is separate

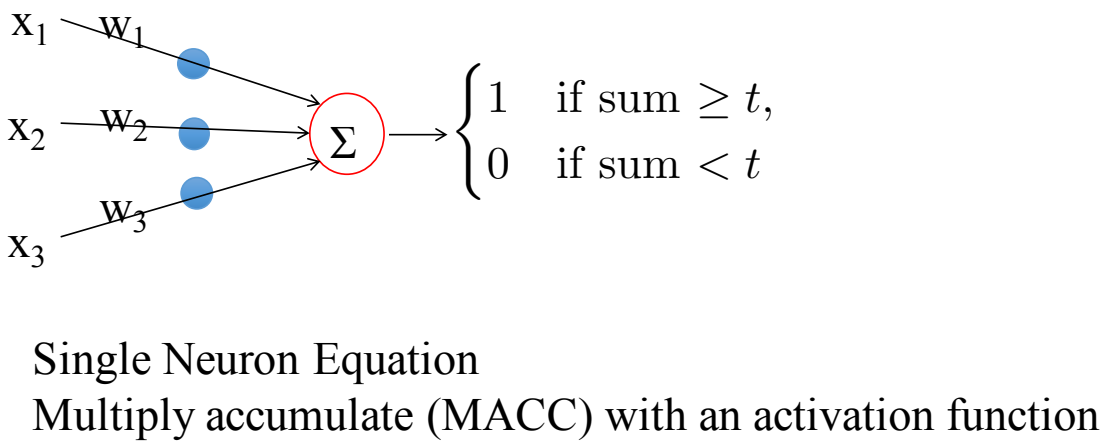

Figure 1.1: Simple neuron.

from the processor [14]. Even for scientific calculations, the time and energy cost of moving the data to the processor, "the von Neumann bottleneck" or "memory wall", is well-known [15]. Over time, a variety of novel processor-in-memory architectures have been proposed as solutions to this problem [16, 17]. Neural nets, where the memory (weights) and processing (MACC plus evaluation) are highly integrated, will benefit from novel architectures as well [1].

Even within the context of traditional models of computing, there are a variety of computer architectures available. The widely used central processing unit (CPU), as typified by Intel's x86 family, is the most common [18]. For more specialized applications such as gaming and film editing, graphics processing units (GPUs) have proven to be highly effective [19]. GPUs trade off more limited functionality with impressive performance for these applications. Field Programmable Gate Arrays (FPGAs) are used by many in the scientific and engineering fields, where the flexibility and performance of programming at the micro-architectural level is a benefit

that outweighs the much more complex programming needed [20]. Finally, Application Specific Integrated Circuits (ASICs) provide the ultimate in performance, but the high cost of fabrication and extreme specialization in capabilities relegate this approach to only a few applications [21]. Even within these groupings, a range of concepts, enabled by new technologies or innovative ideas, are possible. Analog vs. digital computation is one example of these possibilities.

In the neuromorphic computing community, an open question is the design of optimal architectures for implementing neural nets. Because the computing industry has mostly focused on optimizing architectures and technologies for von Neumann architectures and scientific applications, the opportunity for innovation and optimization in neural processors is high. It would be very valuable if a general purpose neural net architecture was available. Since much of the neural net research has been focused on image processing or related applications, it is not clear that architectures defined for them are sufficiently general purpose. Microprocessors are designed for good processing capability across a wide range of applications; a general purpose neural processor should also provide this capability.

1.2 Thesis Statement

A general purpose neuromorphic chip can have capabilities sufficiently close to individually optimized chips to be worth building.

1.3 Primary Contributions

In addition to the design and evaluation of a general purpose neural processing architecture, contributions of this research include the following:

Tiled array concept

The variety of applications used in this research require a variety of crossbar array sizes for efficient mapping of neural nets onto the hardware. Finding a single array size that can fit this variety, desirable for a general purpose neural processor, is problematic. We describe the development and use of tiled arrays to overcome this limitation. A tile can be a complete array (when small arrays are efficient), and they can also be combined to create much larger arrays, which are necessary for certain neural networks. This novel contribution is a key component of the general purpose neural processor.

New comparator design

The tiled array is made possible by the use of a 1/High Z neuron architecture, and the design of a compact, power efficient, fast comparator. Many memristor-based neural nets use a very fast, but large and power hungry analog-to-digital converter (ADC) for neuron evaluation [22]. This would be an impractical approach with the tiled array concept; the new comparator design can efficiently support small tiles, which is an important attribute.

Communication networks

The overall capability of a general purpose neural processor will be heavily dependent on key aspects of the communication network. In this research we explore and quantify the leveraging of sparsity in neural communication, and evaluate two potential on-chip network designs: a hierarchical network of all-to-all switches, and a more traditional 2D mesh.

Limitations on array sizes

Various parasitic or other effects can limit the size of an array; we explore a range of possible limitations, quantifying their effects. For the more severe limitations, we identify techniques or design approaches that can be used to enable larger arrays by eliminating and/or minimizing their impact.

Array architecture

A variety of memristor-array architectures have been proposed in the literature. We explore and quantify the energy efficiency of three specific concepts, demonstrating that one concept (1/High Z) is the most power efficient.

Enhancements to existing models and techniques

This research uses a well-known memristor model [23], but we identify specific improvements for its use in analyzing the general purpose neural processor. This research relies heavily on a versatile yet immature "neural compiler" called Loom [24]; our work reveals its limitations and introduces some enhancements to its capabilities.

1.4 Outline of Dissertation

Chapter 2 will provide background and present related research. The background will give a brief overview of neuromorphic computing, focused primarily on deep neural networks. Examples will be given of the use of neural nets for applications [24, 25]. Memristor technology will be briefly described. Related work will cover an overview of computer architectures for neural nets, with specific examples of an advanced digital design and one utilizing memristor crossbars. Chapter 3 will describe the two learning algorithms (backpropagation and the concurrent learning algorithm) and the three applications (MNIST, CognitiveShield, and AES-256) used to develop and analyze the general purpose neural processor. Chapter 4 will present the general architecture to be used, with a detailed examination of its practical limitations and the key components that comprise the overall design. The tiled array concept will be introduced here, and the details of the communication networks will be examined. Chapter 5 will present the details of using Loom to map the applications onto both special purpose and general purpose architectures, and analyze these designs based on varying the tile size. Chapter 6 will describe the methodology used to select the architectures included for full evaluation, and the details of the evaluation process. Evaluation results will be presented. Chapter 7 will state the conclusions of the research.

Chapter 2: Background

Designing and evaluating a general purpose neural processor requires understanding and applying knowledge from multiple areas. In this chapter, we briefly review the key elements of neural networks and the use of memristor technology and crossbar arrays for building them. A brief description of work related to this research is provided, and two specific designs are discussed more thoroughly: the IBM True North chip [26–28] and the HP dot product engine [22, 29].

2.1 Overview

Neuromorphic computing is based on applying techniques abstracted from neuroscience, in particular approaches implemented as artificial neural nets. Memristor technology, which is being explored as a possible replacement for current memories, is a promising candidate for neuromorphic architectures. These architectures typically take the form of a crossbar array, which is expected to efficiently implement the MACC function depicted in Figure 1.1. Key components of these architectures include the specific memristor technology employed, the circuit design used for the evaluation function, the on-chip network, and the choice of memristor programming approaches.

2.1.1 Neural nets

Neural nets are a specific implementation of neuromorphic computing. A concise description can be found in [30]:
"A standard neural network (NN) consists of many simple, connected processors called neurons, each producing a sequence of real-valued activations. Input neurons get activated through sensors perceiving the environment, other neurons get activated through weighted connections from previously active neurons. Some neurons may influence the environment by triggering actions. Learning or credit assignment is about finding weights that make the NN exhibit desired behavior, such as driving a car. Depending on the problem and how the neurons are connected, such behavior may require long causal chains of computational stages, where each stage transforms (often in a non-linear way) the aggregate activation of the network. Deep Learning is about accurately assigning credit across many such stages." This deeper hierarchy of features tends to make classification more stable.

Neural nets have been shown to be very useful for a variety of image recognition/image processing applications [7, 9, 12, 13, 31]. They have also been shown to be useful in malware analysis [24, 32], and the use of deep neural nets (typified by artificial neural nets, spiking neural nets, recurrent neural nets, and convolutional neural nets) for applications is increasing rapidly [33]. For example, AlphaGo is a deep neural net trained to play the board game Go, which recently defeated a world class expert [34]. Useful neural nets can range in size from very small (a few

hundred neurons), to very large (hundreds of thousands of neurons). They can also have very few layers of connected neurons (2-5) to very many (up to 100 have been trained [33]). The AlphaGo neural net has 15 layers of neurons, where the input layer is a 19x19x48 three-dimensional tensor; the Go game is on a 19x19 grid, and 48 specific features are used to represent the board and the game situation at any given time [34]. Algorithms used to assign the weights of the neural network are called learning algorithms that "train" the network; backpropagation, to be described in Chapter 3, is a well-known learning algorithm [35,36]. Training is an extremely computationally intense process; AlphaGo used 50 GPUs and required 1 day to play 1 million Go games as part of its training process. Using the neural net to perform its trained task is called inference; this is much less computationally demanding. The fouce of this research is on designing and evaluating a general purpose neural processor for inference.

2.1.2 Memristors

In 1971, Leon Chua published a paper [37] hypothesizing the existence of a fourth basic circuit element, which he called a memristor (a resistor with memory). He reasoned that a fourth circuit element should exist in order to complete basic linear relationships between the four state variables (V, I, Q, and φ). Chua described the basic current-voltage relationship for a memristor in [38]:

$$I = G(X, V, t) * V \tag{2.1}$$

where G is the device conductance, and is dependent on a state variable X.

$$dX/dt = f(X, V, t) \tag{2.2}$$

The rate of change of X is dependent upon the current value of X and other inputs. It is the interplay of G and dX/dt that gives the memristor its unique properties. After Hewlett-Packard published a well-known Nature paper in 2008 reporting the discovery of a memristor [39], Chua published additional work [40, 41] stating that all resistance switching memories follow memristive behavior, and then identifying three specific characteristics (or fingerprints) that need to exist for a device to be a memristor. These three fingerprints include:

1. Pinched hysteresis in the I-V curve (I=0 at V=0 for an ideal device);
2. Reduction in the lobe areas of the device as frequency increases;
3. Straight line behavior (single resistive value) as the frequency approaches infinity.

Figure 2.1 provides an example of a hysteresis curve.

Because of their programmable conductance capability, memristors can function as dynamic weights in neuromorphic computing designs [42]. Referencing Figure 1.1, memristors can be programmed as the weights (w_i) applied to the inputs (x_i); since the programming process can be controlled using feedback from the output, it is possible for memristor-based designs to be modified during operation, integrating learning into the system [43].

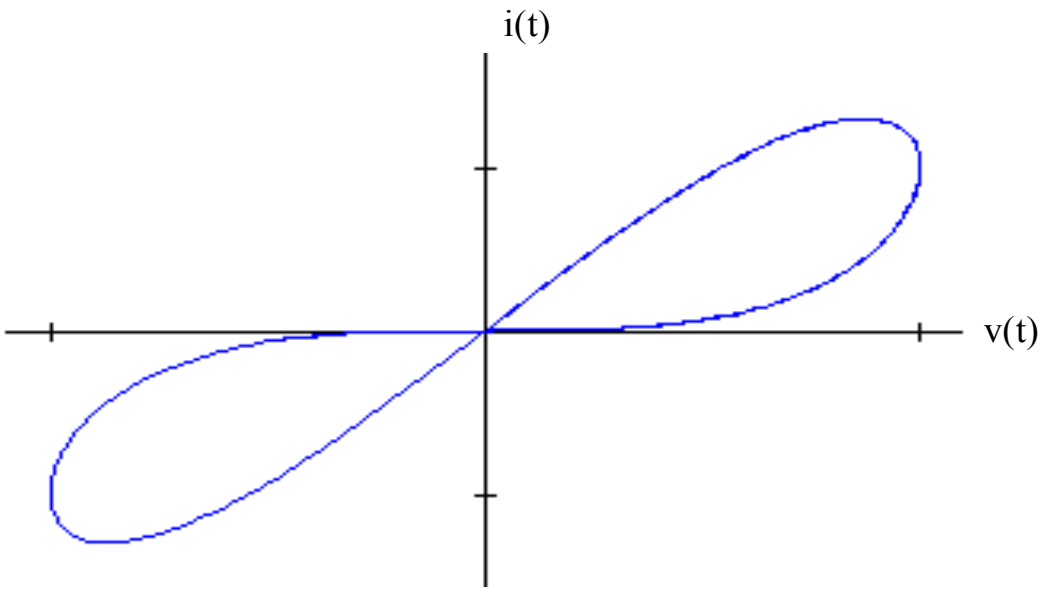

Figure 2.1: General hysteresis curve for a memristor.

A comprehensive overview of memristor technology can be found in [44].

2.2 Related work in neuromorphic architectures

There are a number of groups implementing neural nets in a variety of architectures and technologies, including floating gate devices [45], GPUs [12], FPGAs [13], ASIC designs [46], and even custom analog circuits [47]. While CPU and GPU based systems are typically used for the learning phase, the more specialized architectures promise significant improvements in energy efficiency and area when used as the inference engines during the operation of the neural net [45]. If a specific learning algorithm, such as back propagation, is expected to be used, the specialized architectures can also implement on-chip training [25].

2.2.1 TrueNorth

TrueNorth is a digital ASIC, developed by IBM under the DARPA SYNAPSE program. Detailed information about this processor can be found in [26–28]. The design has been fabricated in a 28 nm process, and features a 64 x 64 array of neurosynaptic cores; each core is comprised of 256 inputs and 256 neurons (over 1 million total neurons). Communication is handled by a 2D mesh network, which enables the design to easily scale to more than one chip (a 16 chip board is available). It is probably the most complete neuromorphic computing system in existence. A full ecosystem of applications, system software and firmware (corelet programming and placement), and training is available for users; the design workflow has been exercised by a growing number of researchers. Applications demonstrated on the system include, image classification, video tracking, neural circuit modeling and a variety of robotics/autonomous systems. The system was originally developed to enable the exploration of a wide variety of neuron implementations; the flexibility of the design for this purpose limits its capabilities in certain ways (for example, the on-chip neurons are limited to a maximum output frequency of 1 kHz).

Our research focuses on a specific neuron type (TGN), which dramatically improves its performance; our neuron can have an ouput frequency $\geq$ 250 MHz. We also use analog computing in a memristor crossbar array to reduce the neuron area and power.

2.2.2 Memristor-based crossbar arrays

Compact size, capability for storing large numbers of weights in the form of conductance values, low energy operation, and technology scaling opportunities make memristor based crossbar arrays a very appealing option for neural nets [48–53]. Figure 2.2 shows the basic structures used for creating a memristor-based crossbar array. The upper left portion shows a two layer neural net (three inputs, seven neurons in the first layer, fully connected to two neurons in the second layer). To the left is an abstracted drawing of one of the neurons in the input layer, where each blue dot represents a synapse which is implemented as a 1T1M (one transistor, one memristor) circuit, shown to the right. The bit line sums all the weighted inputs: the current running in the bit line is equal to the sum of the individual components, and each component is directly proportional to the conductance (weight) of the memristor. This circuit therefore directly (and efficiently) performs the multiply-accumulate function needed, using Ohm's Law to make a direct calculation:

$$I_{\text{total}} = G_{\text{total}} * V_{\text{mem}} \tag{2.3}$$

where V_{mem} is the voltage across the memristor.

The use of two synapses for each input enables negative weights to be represented in a simple manner, and the differential comparator at the bottom of the bit line performs the evaluation function of the threshold gate network. The bottom figure is the memristor crossbar array version of the two layer neural net.

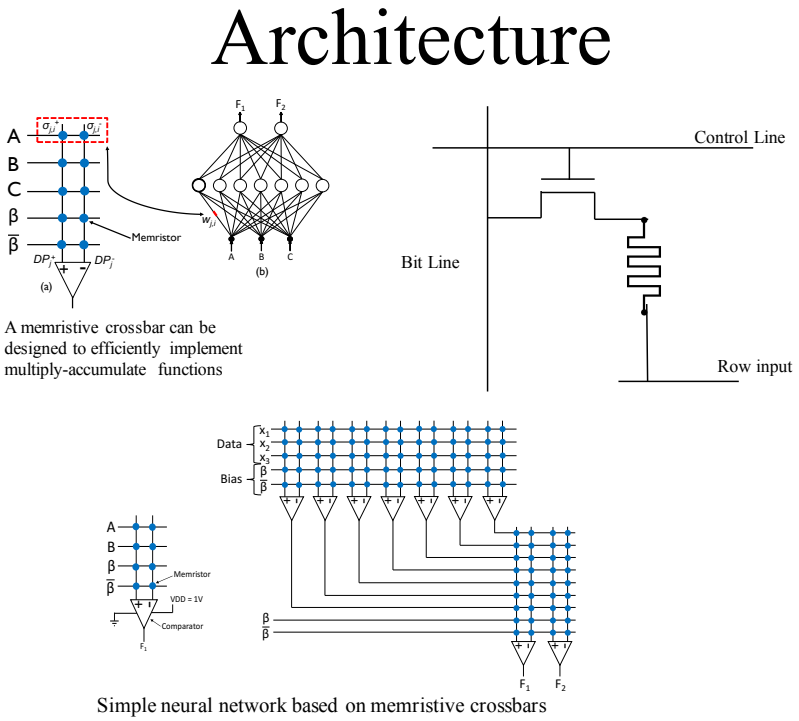

Figure 2.2: Basics of a 2 layer neural net architecture using memristors.

2.2.3 Dot Product Engine

The Dot Product Engine (DPE) is a neuromorphic computing system design by HP based on analog processing in memristor crossbars, developed as part of an IARPA program [22, 29]. It uses the crossbar for matrix multiplication, and incorporates DAC (digital to analog converter) and ADC circuits to enable analog computation with digital communication. They have refined their memristor technology to enable up to 32 states to be precisely programmed (1% accuracy), analyzed the effect of programming on computing accuracy, and characterized the memristors for temperature, noise and stochasticity effects. Based on this work, they have developed a chip architecture (called ISAAC) for a neural processor. The ISAAC design is very detailed, including an analysis of on-chip storage with eDRAM (embedded Dynamic Random Access Memory), the analog processing crossbar, and peripheral circuits (programming, evaluation, inputs, etc.). A breakdown of the components, along with estimates of power, area, and timing, are provided. A comparison of this design to digital approaches is made, using system metrics such as computational efficiency (Ops/mm^2), power efficiency (Ops/W), and storage efficiency (capacity of synaptic weights in MB/mm^2).

Our research incorporates a specialized comparator design in lieu of a general purpose ADC, which significantly improves the area and performance of the processor. We also evaluate our neural processor against a more diverse set of applications, not just image and face recognition. We also create and evaluate special purpose, limited purpose, and general purpose designs within the same architecture.

Chapter 3: Conceptual approach

To analyze architectural options, the typical procedure is to use benchmarks and proxy apps [54]. The benchmarks and proxies are chosen to represent the range of applications that the processor is expected to execute. The choice of applications is an important determinant of the capabilities of the processor. Because most research in neural architectures to date have used primarily image processing or similar applications [7, 9, 12, 13, 31], the spectrum of applications explored has been relatively narrow. Explorations of general purpose architectures would be well-served by looking at a wider range of applications. We will briefly describe the two neural net learning algorithms used in our analysis, along with the three applications chosen. These applications range from traditional (MNIST) to extremely novel and highly digital (AES-256), providing a much wider spectrum of applications that our general purpose architecture must handle.

3.1 Learning algorithms

One of the most prevalent algorithms for training neural nets is error back propagation (BP) [35, 36]. This technique requires that the output of the neruon y_i be a differentiable function of its inputs x_j and weights w_{ji}. This technique takes

the error from the NN (desired minus actual value) and propagates it backwards through the NN by first calculating the partial derivative for each of the outputs

$$\frac{\partial E}{\partial y_{\mathrm{i}}} = y_{\mathrm{i}} - d_{\mathrm{i}} \tag{3.1}$$

where E is the total error. By applying the chain rule you can calculate the effect of the inputs on the total error.

$$\frac{\partial E}{\partial x_{\mathrm{j}}} = \frac{\partial E}{\partial y_{\mathrm{i}}} \cdot \frac{dy_{\mathrm{i}}}{dx_{\mathrm{j}}} \tag{3.2}$$

For a set of inputs with weights w_{ji} connecting input x_j to output y_i the error contribution can be found.

$$\frac{\partial E}{\partial y_{\mathrm{i}}} = \sum_{j} \frac{\partial E}{\partial x_{\mathrm{j}}} \cdot w_{\mathrm{ji}} \tag{3.3}$$

but the inputs x_j are themselves just outputs from a previous layer, so the error from any input can also be calculated (or propagated backwards) to its inputs and weights. The error contribution of any weight can also be calculated.

$$\frac{\partial E}{\partial w_{\mathrm{ji}}} = \frac{\partial E}{\partial x_{\mathrm{j}}} \cdot y_{\mathrm{i}} \tag{3.4}$$

After a training phase is completed, individual weights can then be modified in order to reduce the overall error, and the process is repeated.

$$\delta w_{\mathrm{ji}} = -\epsilon \cdot \frac{\partial E}{\partial w_{\mathrm{ji}}} \tag{3.5}$$

The process of inputting a set of training data into the NN, calculating the errors and adjusting the weights is termed a training epoch. Multiple training epochs are used so that the total error is reduced; training is completed when:

- The total error is reduced to some specific value; or
- The change in error between epochs is less than a specified amount; or
- A pre-determined number of epochs has been completed.

After training, the NN is validated by inputting a set of new data (previously unseen by the NN) and calculating the error. As described, BP is a supervised learning algorithm; it uses correctly classified (or labeled) data to find the error. Figure 3.1 provides an example of how the error is reduced over many epochs; however, excessive training can lead to increased errors during validation of the neural net. Therefore, selection of the error parameters for training is important.

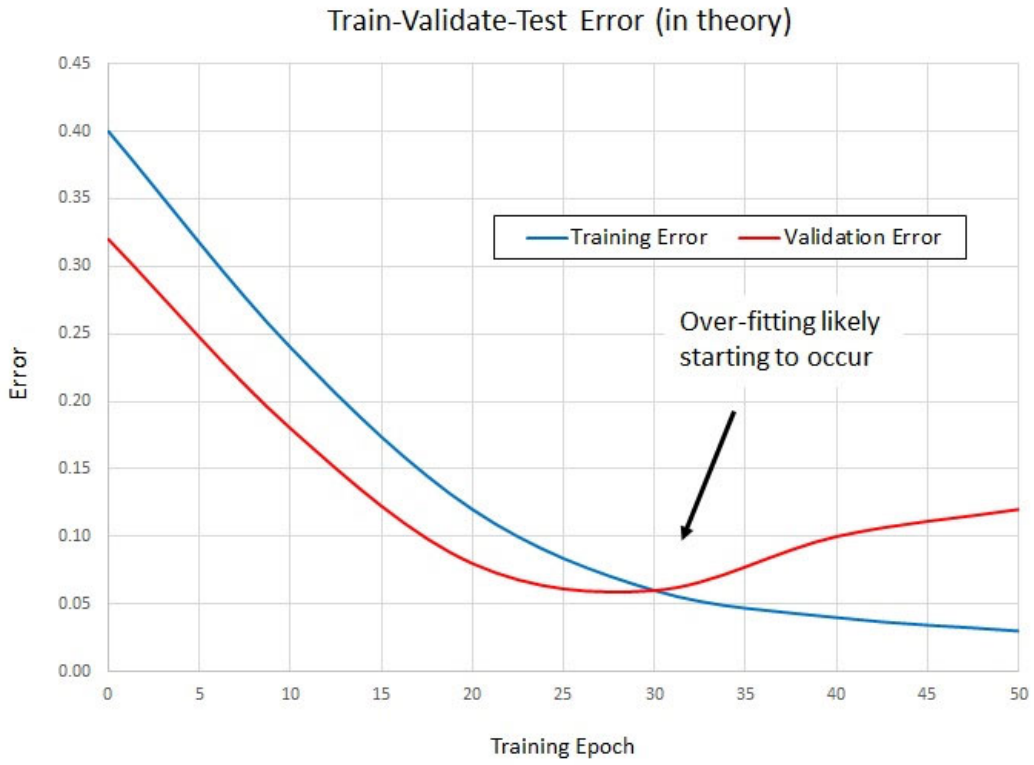

Figure 3.1: Example graph of training and validation errors.

Practical issues with using BP such as exponential decay for deep neural nets [55] have led researchers to develop variations on this approach [56]. One variation [57] is the concurrent learning algorithm (CLA); this is also a supervised learning algorithm. In this approach, the inputs to the neurons have an additional attribute termed the influence. The influence is calculated when the nodes of the neural net are created. The nodes are mapped into a 3D space, and the distance between any two nodes (analogous to a Euclidean distance) can be calculated and the influence is found as:

$$S = \frac{1}{1 + d^2} \tag{3.6}$$

Any two nodes that have an influence below a set value will not be connected; traditional feedforward networks are completely connected. The use of influence creates two valuable conditions:

- Far fewer connections are made, enabling much faster training epochs.
- Nodes separated by an intermediate layer are allowed to be connected, which also speeds training.

Additionally, each node in the hidden layers (stages that are not inputs or outputs) is required to have a minimum of one connection to a delta (error) node; the strength and existence of that connection is also determined by the influence. This enables parallelization of the weight updates during the learning phase, hence the name concurrent learning algorithm. The use of CLA does have the drawback of usually requiring more training epochs, and potentially more layers in the network.

Regardless of the learning algorithm selected, the end result is the creation of a specific neural network: input, hidden, and output layers, with node connectivity and weights quantified. This means that a general purpose neural processor should be able to execute a neural network trained using any method. Differences in the specific neural network created by varying learning algorithms can be quite small, Figure 3.2 shows the distribution of weights for the MNIST application trained with both BP and CLA; they are extremely similar.

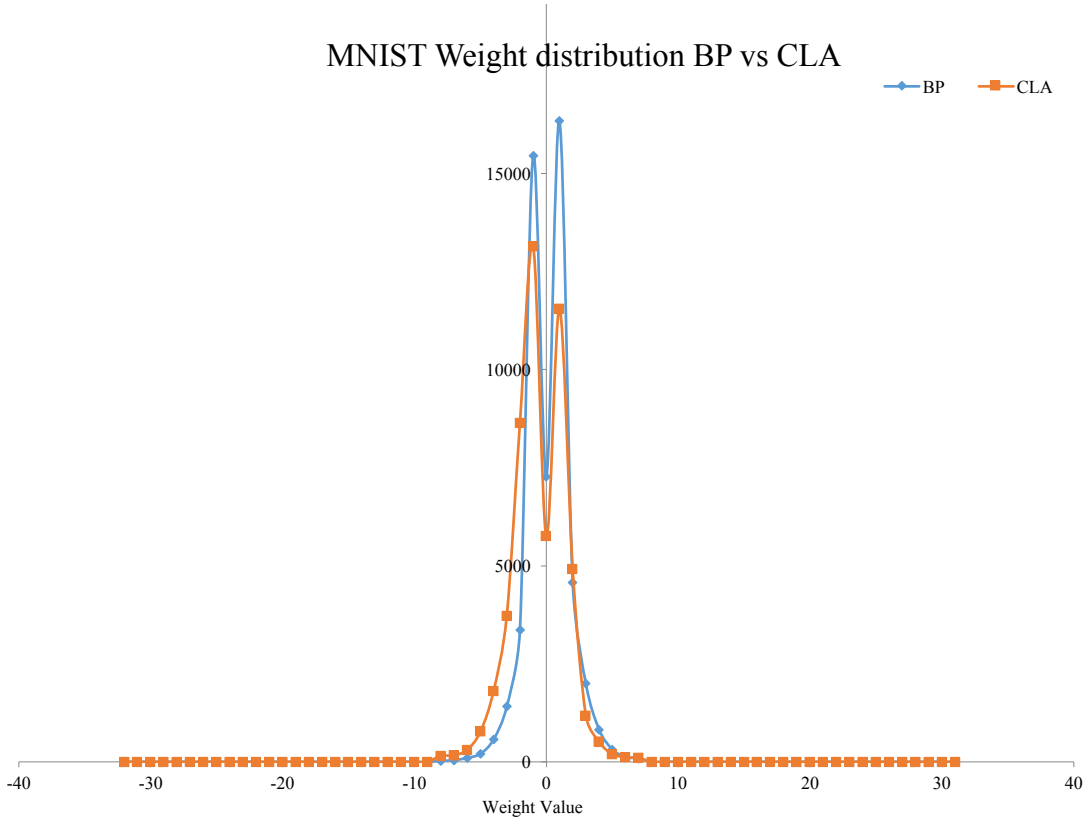

Figure 3.2: Distribution of neural network weights for MNIST, using both back-propagation and concurrent learning algorithms.

3.2 Application description

Our research exploration includes the development of multiple neuromorphic computing architectures. Three of the architectures will be special purpose designs (SPD), each optimized for one specific application. The applications can be thought of as representing clearly different points on the computing spectrum for neuromorphic computing. The applications chosen are:

1. MNIST, which represents a very well-known baseline application. This ensures traditional neural net applications are part of the exploration. Since MNIST is a relatively simple and small neural net, it will also be used to describe the various analyses done as well.
2. Malware analysis, which represents a hybrid neural net pattern classification application [24, 32]. Parts of this application are essentially digital functions, but the final layers perform more traditional classifier functions. This application is very relevant to cybersecurity.
3. AES-256, to demonstrate that complete digital functionality is possible using a neural net approach [24]. This is a highly unusual neural net application, and can be expected to levy requirements on the architetcure that are very different from MNIST. This application is also relevant to cybersecurity.

These were chosen because they represent a wider range of applications than

just variations on traditional image processing, and a general purpose neural processor capable of running these applications efficiently would be valuable for a variety of cybersecurity needs, which is expected to be a relevant domain for neuromorphic computing [4].

3.2.1 MNIST

MNIST is a commonly used application for evaluating various neural network properties and design options [28, 48, 58]. The application is used to identify handwritten digits from 0 to 9; most neural nets for MNIST will have accuracies above 90%. The NN used in this exploration consists of two layers (see Figure 3.3).

- An input layer, consisting of 768 inputs and 256 outputs (neurons), where the inputs represent a binary value (dark pixel or light pixel)
- An output layer, consisting of 256 inputs and 10 outputs, where each output directly represents one of the possible digits

A specific output neuron will fire (output = 1) to indicate the classification of the input signals as representing its digit. This is a small neural net (266 neurons), and requires about 50 epochs to train.

3.2.2 CSlite

Cognitive Shield is a neural network designed to identify malware using n-grams as inputs [24, 32]. An n-gram is a piece of the code (in binary); the CSlite

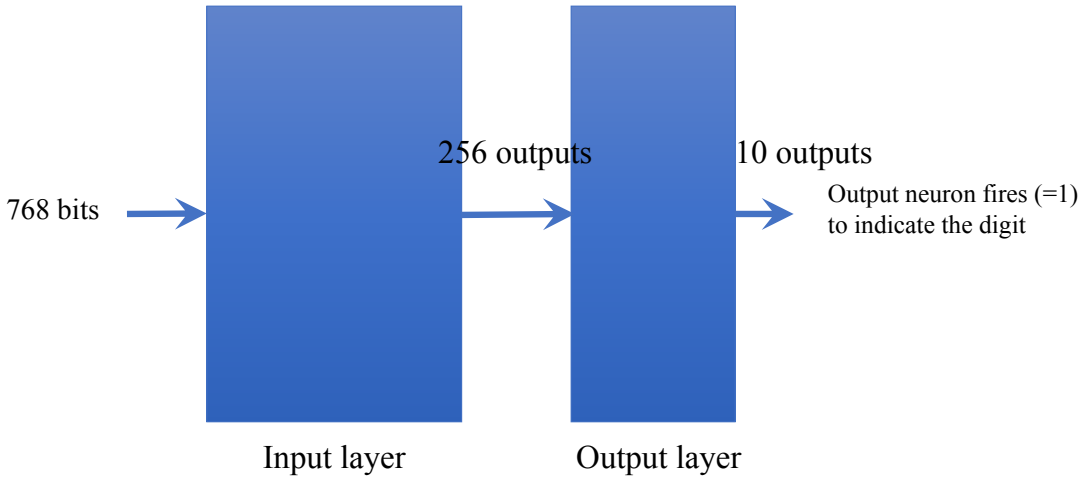

Figure 3.3: MNIST neural net.

neural net uses 6 byte (48 bit) n-grams for its inputs. Malware detection accuracy of 80-90% is expected. CSlite is the streamlined version of Cognitive Shield, designed for use in an embedded computing system that needs to handle high volumes of input data. The use of neural nets for malware classification, instead of signature based approaches, can be of high value for finding new malware without the need to painstakingly deconstruct the code [32]. Rapidly identifying malware in high speed data streams is an important cybersecurity application. The availability of a compact, energy efficient neural processor would be extremely useful. The NN for CSlite is shown in Figure 3.4. There are four distinct components:

- A decoder circuit (48 inputs, 1536 neurons), which takes each input byte and fires one neuron (of 256) to represent its unique binary representation

- A signature pattern matcher (1536 inputs, 2000 neurons), which compares the value against 2000 stored features; a feature represents an n-gram that is

highly significant for indicating malware or goodware

- A latch (2000 inputs, 2000 outputs), which keeps track of which 2000 features have been found in a given input stream; there is also a reset input to return all outputs to zero

- The detector stage (2000 inputs, 1 output), which is actually a three layer NN classifier (2000:256, 256:128, 128:1); the output fires when the input stream has been identified as malware

Three of the components of this NN (decoder, signature, latch) are essentially digital functions; only the detector component is a traditional NN. CSlite represents a hybrid application – part digital, part neural. The entire application requires approximately 5800 neurons in 6 layers, which makes it a small-to-medium sized NN; it requires about 100 epochs to train.

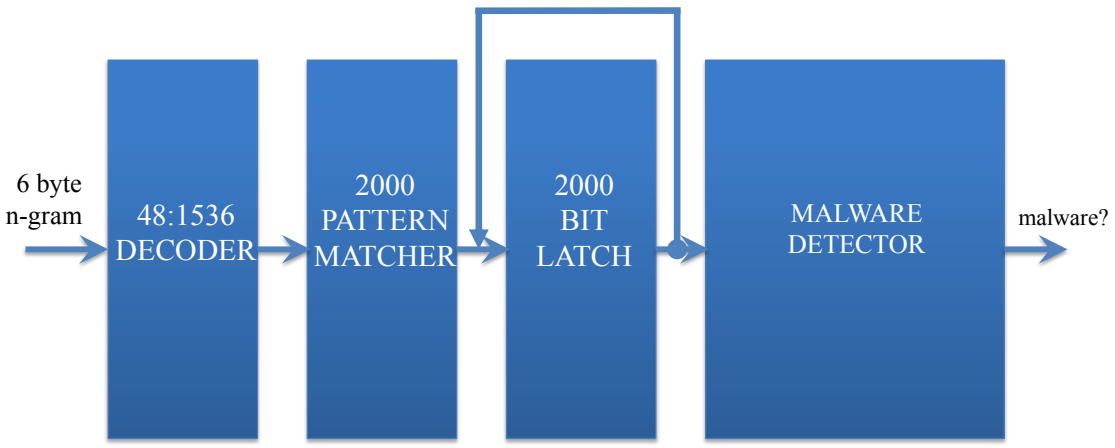

Figure 3.4: Conceptual diagram of the CSlite malware detection neural net.

3.2.3 AES-256

The Advanced Encryption Standard is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001 [59]. AES-256 is a block cipher with a block size of 128 bits and a key length of 256 bits. AES became effective as a federal government standard on May 26, 2002 after approval by the Secretary of Commerce. AES is the first (and only) publicly accessible cipher approved by the National Security Agency (NSA) for top secret information when used in an NSA approved cryptographic module. Since AES was developed for traditional digital computing architectures, it represents an application that may be difficult to implement in a neural net since it must be 100% accurate, but it has been done [24]. The implementation takes advantage of digital hierarchical neural nets, where a large function can be decomposed into smaller individual function and then re-assembled. This decomposition enables the NN to be completely trained against all possible inputs; once the NN trains to no errors, the function is known to be correctly implemented (100% accuracy). The decomposition enables a large digital function to be trained in a reasonable time:

- A 64 input NN would require 1.6 x 10^{19} training vectors for complete training
- Partitioning this into 16 input components reduces this to 2.56 x 10^{5} training vectors

The NN for AES-256 is shown in Figure 3.5. The figure shown is replicated 4 times to handle the entire 128 bit input plain text, and output the complete 128 bit

cipher text. There are 5 distinct components:

- A 32 bit multiplexer (64 inputs , 32 outputs), which selects either the original plain text (first pass) or an intermediate cipher text (all other passes) to send to the next component
- A SubBytes or partial S-Box (8 inputs, 8 outputs), which performs specified arithmetic functions to perform a substitution
- Mix Columns A and B (16 inputs, 32 outputs), which perform a polynomial function on their inputs; in this NN, Mix Columns A and B are the same NN with differing weights
- Mix Column C (16 inputs, 8 outputs), which performs a similar function but is a completely different neural network
- The state machine (5 inputs, 6 outputs) which controls the overall functionality of the NN data flow

The entire application requires approximately 12,500 neurons in 8 layers, which is a medium sized NN. The individual components take between 100-500 epochs to train; once trained, they can be composed into a single application.

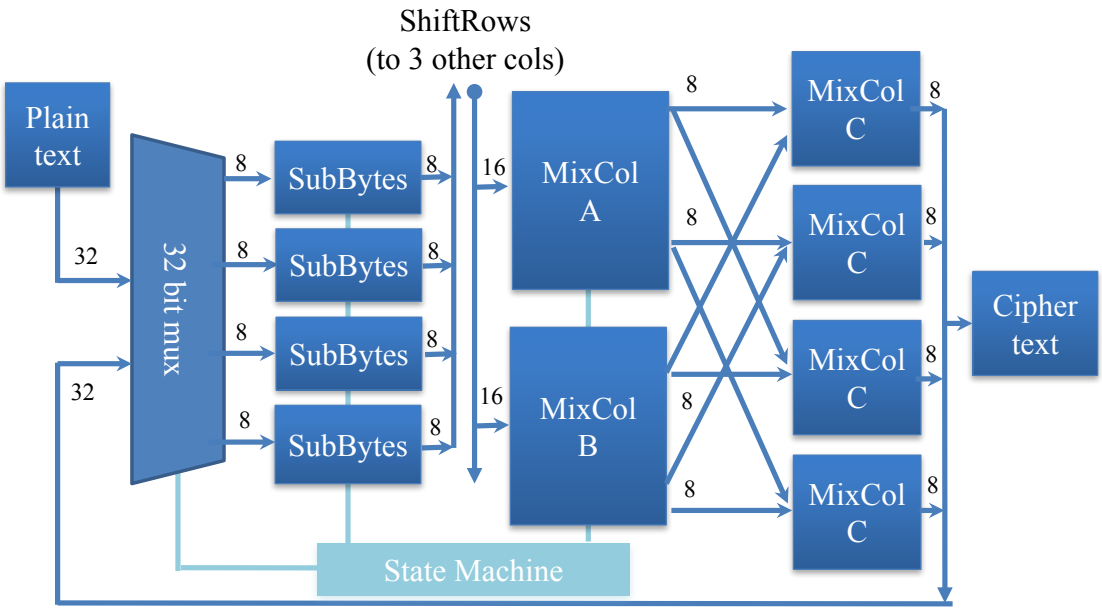

Figure 3.5: Conceptual diagram of the AES-256 neural net.

Chapter 4: Basic architecture and key components

Figure 4.1 is a block diagram of the basic architecture to be studied. This architecture assumes off-chip learning to enable algorithm flexibility, and incorporates 1T1M unit cells to provide for more precise control of the memristors during programming and operation. Comparators are used for the evaluation circuit to enable a digital communication network and digital input circuits. Each of the key components will be described, with specific emphasis on the comparators and the on-chip network; potential limitations to building arrays of arbitrary size will also be defined. This building block can be replicated multiple times to create a full chip version for large neural processing applications. In the analyses below, we are generally assuming a 256 x 64 array; this refers to the number of inputs and neurons, the number of columns (128 in this case) is twice that, since we are using a differential current architecture. This size array is well suited for neural networks evaluating the MNIST application [58]. This array size, fabricated in 45 nm technology, is the nominal design point for these analyses. If other design points are used, they will be explicitly noted. Simulations were done using LTspice [60] and 45nm technology device parameters [61] [62].

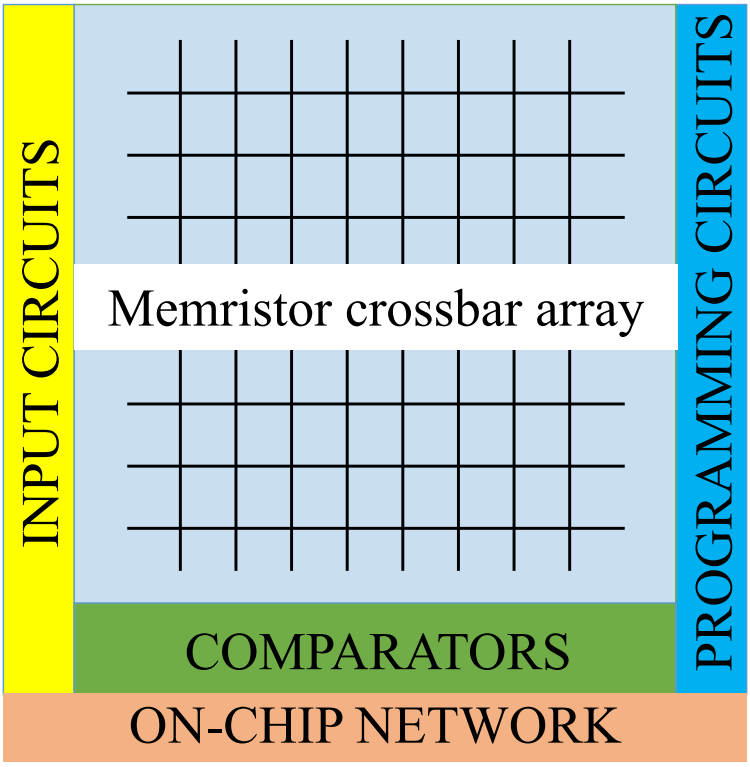

Figure 4.1: Block diagram of the NMC architecture showing the key components.

4.1 Analysis of neuron architectures

Memristor arrays implementing TGN can be organized in multiple ways; below we analyze three variations. We will make specific assumptions and simplifications to the variations to enable relative power comparisons to be made.

4.1.1 Single-ended voltage

Figure 4.2 is an example of a single-ended voltage architecture (SV); every input and its inverse, along with any bias inputs, are connected to a single voltage rail through a memristor (represented by the blue circle) with a specific programmed conductance. This architecture has been used to design and analyze embedded neural network processors [58]. G_0 represents an "off" state, or extremely low conductance, G_1 represents an "on" state, or high conductance. A typical on/off ratio can be $\geq$ 100 [63]. In our TGN, all the weights are integer values, which can be represented in this simplified SV architecture as one or more inputs with $w_i = G_1$. For example, $w_i = 2$ can be represented by two inputs with $w_i = G_1$ (for the MNIST application, over 90% of the weights are -1, 0 , or +1). This representation is used here merely to simplify the analysis; for the actual design a single memristor is programmed with a conductance value (G_i) that represents the desired weight for that particular input and neuron. The voltage V_{in} is compared to a reference voltage with a differential voltage comparator to create the threshold gate. The circuit in

Figure 4.2 can represent any neuron and set of inputs in a TGN by selecting the proper values for α_{11}, α_{10}, α_{01}, α_{00}, and n.

The equivalent circuit for this architecture is also shown in Figure 4.2. G_{eff}^{+} represents the sum of all conductances connected to V_{dd}, and G_{eff}^{-} represents the sum of conductances connected to V_{ss}.

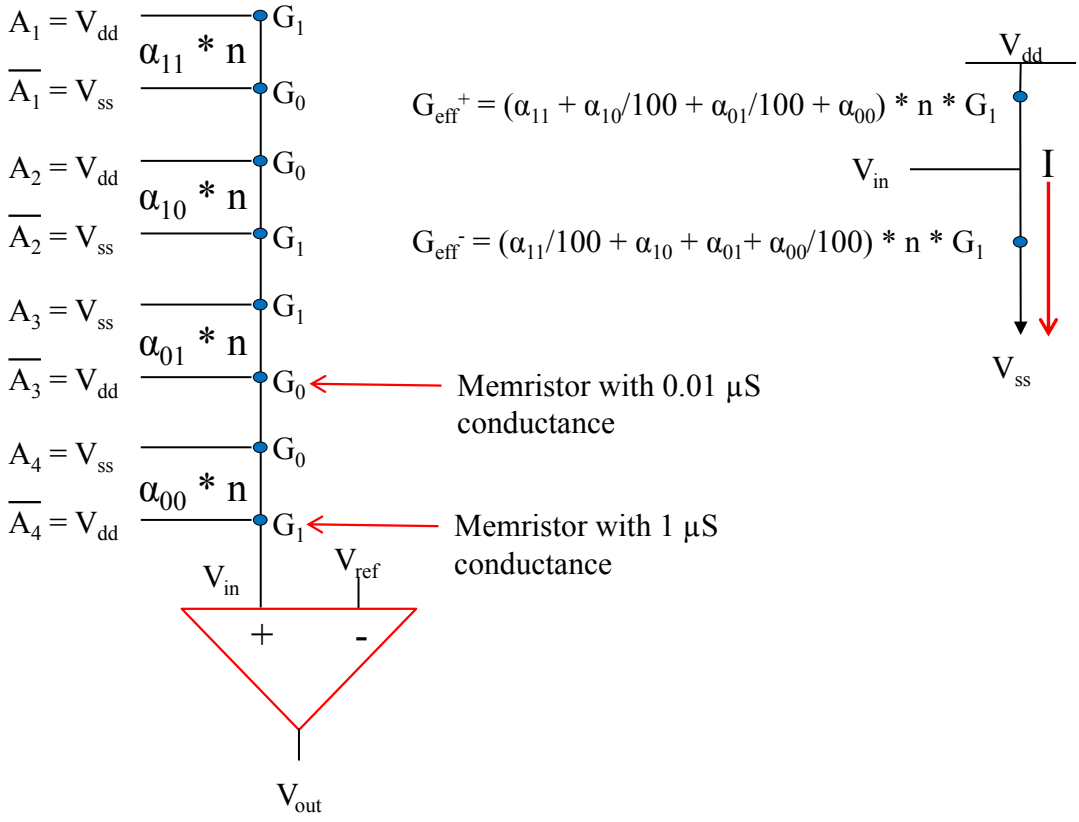

Figure 4.2: Single-ended voltage architecture (SV) and equivalent circuit.

We make the following assumptions for this analysis.

n = total number of inputs

α_{11}, α_{10}, α_{01}, and α_{00} represent the fraction of inputs of each type

($\alpha_{11} + \alpha_{10} + \alpha_{01} + \alpha_{00} = 1$ by definition)

The two possible conductances are G_1 = 1 μS and G_0 = 0.01 μS. The important condition for this analysis is that $G_0 = G_1/100$. Given this G_0/G_1 ratio, for the G_{eff}^+ estimate we will assume (α_{11}, α_{00}) >> ($\alpha_{10}/100$, $\alpha_{01}/100$), and for the G_{eff}^- estimate we will assume (α_{10}, α_{01}) >> ($\alpha_{11}/100$, $\alpha_{00}/100$). To demonstrate the validity of this assumption, consider the following example: let $\alpha_{11} = 0.1$ and $\alpha_{00} = 0.1$; if α_{10} or $\alpha_{01} = 0.8$, then the assumption is $0.1 >> 0.008$, which is reasonable. For equal probability of inputs, this assumption is essentially never violated (much less than 1% of the time).

Using these assumptions, we can simplify G_{eff}^+ and G_{eff}^-

$$G_{\text{eff}}{}^{+} = (\alpha_{11} + \alpha_{00}) * n * G_1 \tag{4.1}$$

$$G_{\text{eff}}{}^{-} = (\alpha_{10} + \alpha_{01}) * n * G_1 \tag{4.2}$$

Assuming $V_{ss} = 0$ Volts,

$$V_{\text{in}} = V_{\text{dd}} * (\alpha_{11} + \alpha_{00})/(\alpha_{11} + \alpha_{10} + \alpha_{01} + \alpha_{00}) = V_{\text{dd}} * (\alpha_{11} + \alpha_{00}) \tag{4.3}$$

$$I = G_{\text{eff}}{}^{-} * V_{\text{in}} = n * G_1 * V_{\text{dd}} * (\alpha_{11} + \alpha_{00}) * (\alpha_{10} + \alpha_{01}) \tag{4.4}$$

$$P_{\text{s}} = V_{\text{dd}} * I = n * G_1 * V_{\text{dd}}{}^2 * [(\alpha_{11} + \alpha_{00}) * (\alpha_{10} + \alpha_{01})] \tag{4.5}$$

P_s can now be calculated for a variety of input conditions. For example:

P_{max} can be found by using the substitutions

$$\gamma = \alpha_{11} + \alpha_{00} \tag{4.6}$$

and

$$1 - \gamma = \alpha_{01} + \alpha_{10} \tag{4.7}$$

then

$$P_s = n * G_1 * {V_{dd}}^2 * [(\gamma) * (1 - \gamma)] \tag{4.8}$$

$P_s = P_{max}$ when

$$\frac{\partial P_s}{\partial \gamma} = 0 \tag{4.9}$$

this occurs when

$$\gamma = 0.5 \tag{4.10}$$

So for

$$(a_{11} + a_{00}) = (a_{10} + a_{01}) = 0.5 \tag{4.11}$$

or

$$\alpha_{11} = \alpha_{10} = \alpha_{01} = \alpha_{00} = 0.25 \tag{4.12}$$

we find

$$P_s = P_{max} = 0.25 * n * G_1 * {V_{dd}}^2 \tag{4.13}$$

4.1.2 Differential voltage

Figure 4.3 is an example of a differential voltage architecture (DV); every input and bias is connected to both a positive voltage rail and negative voltage rail conductance. The equivalent circuits for this architecture are shown. The two voltages are compared with a differential voltage comparator to create the threshold gate. A processor using this architecture to implement a character recognition application was analyzed in [48].

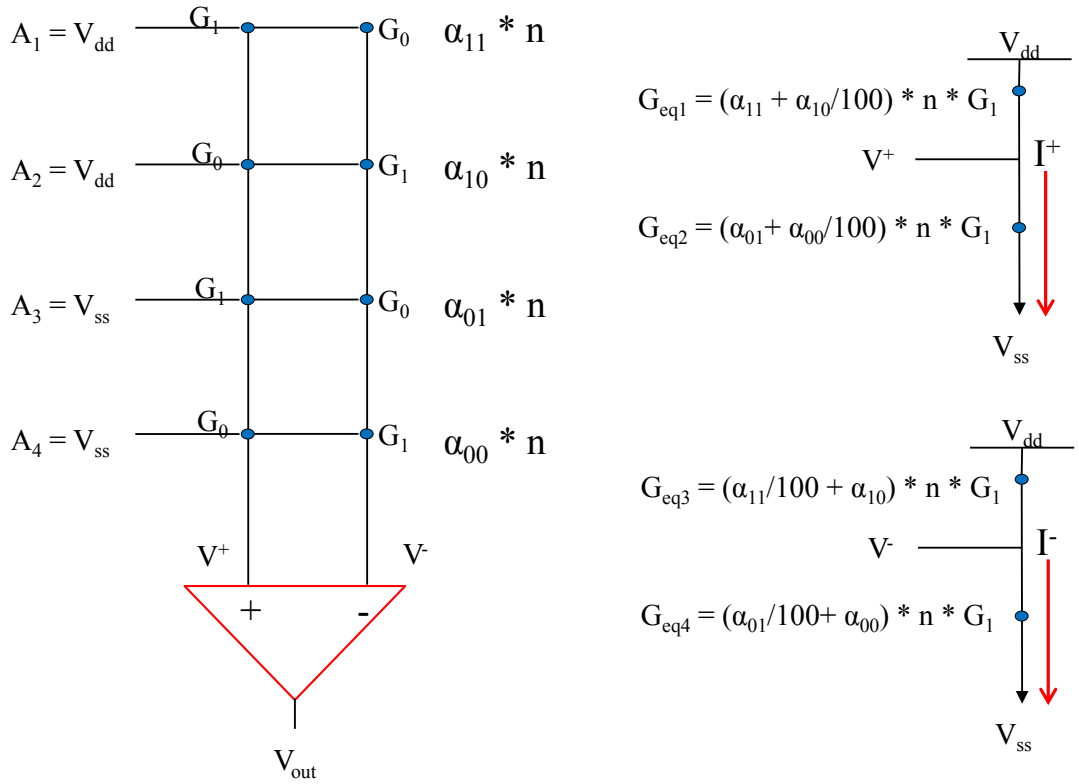

Figure 4.3: Differential voltage architecture (DV) and equivalent circuits.

Using the same assumptions as before, we find:

$$V^+ = V_{\mathrm{dd}} * (\alpha_{11})/(\alpha_{11} + \alpha_{01}) \tag{4.14}$$

$$I^+ = n * G_1 * V^+ * \alpha_{01} = n * G_1 * V_{\mathrm{dd}} * (\alpha_{11} * \alpha_{01})/(\alpha_{11} + \alpha_{01}) \tag{4.15}$$

$$P^+ = n * G_1 * {V_{\mathrm{dd}}}^2 * (\alpha_{11} * \alpha_{01})/(\alpha_{11} + \alpha_{01}) \tag{4.16}$$

$$V^- = V_{\mathrm{dd}} * (\alpha_{10})/(\alpha_{10} + \alpha_{00}) \tag{4.17}$$

$$I^- = \alpha_{00} * n * G_1 * V^- = (\alpha_{10} * \alpha_{00} * n * G_1 * V_{\mathrm{dd}})/(\alpha_{10} + \alpha_{00}) \tag{4.18}$$

$$P^- = (\alpha_{10} * \alpha_{00} * n * G_1 * {V_{\mathrm{dd}}}^2)/(\alpha_{10} + \alpha_{00}) \tag{4.19}$$

$$P_{\mathrm{d}} = P^+ + P^- = (n * G_1 * {V_{\mathrm{dd}}}^2) * [(\alpha_{11} * \alpha_{01})/(\alpha_{11} + \alpha_{01}) + (\alpha_{10} * \alpha_{00})/(\alpha_{10} + \alpha_{00})] \tag{4.20}$$

P_{d} is generally not equal to P_{s}; however for

$$\alpha_{11} = \alpha_{10} = \alpha_{01} = \alpha_{00} = 0.25 \tag{4.21}$$

$$P_{\mathrm{d}} = P_{\mathrm{s}} = 0.25 * n * G_1 * {V_{\mathrm{dd}}}^2 \tag{4.22}$$

$P_{\mathrm{s}} > P_{\mathrm{d}}$ when $(\alpha_{11} + \alpha_{10}) > (\alpha_{01} + \alpha_{00})$ or $(\alpha_{11} + \alpha_{10}) < (\alpha_{01} + \alpha_{00})$; lots of inputs are one or zero (see Figure 4.4). The exact $P_{\mathrm{s}}/P_{\mathrm{d}}$ value will depend on the specific distribution of the four possible types of inputs. For a given x-axis value (fraction of ones) in Figure 4.4, the $P_{\mathrm{s}}/P_{\mathrm{d}}$ value for a range of $\alpha_{\mathrm{i,j}}$ has been plotted.

It is straightforward to determine the ratio of ones $(\alpha_{11} + \alpha_{10})$ and zeroes $(\alpha_{01} + \alpha_{00})$ for each neuron in an array, at least for small neural nets, by keeping track of

them during the neural net training and/or validation. The nominal value (fraction of ones) for the crossbar arrays in a neural network running the MNIST application are 5%, 5%, 20%, 20%, and 62% for the four input layers and the one output layer, respectively. Assuming that the ones are equally split between α_{11} and α_{10} (with a similar assumption for the zeroes), we find that $P_s = 2.26 * P_d$.

A differential voltage architecture should also have improved common mode noise rejection. This analysis indicates DV is a better choice than SV.

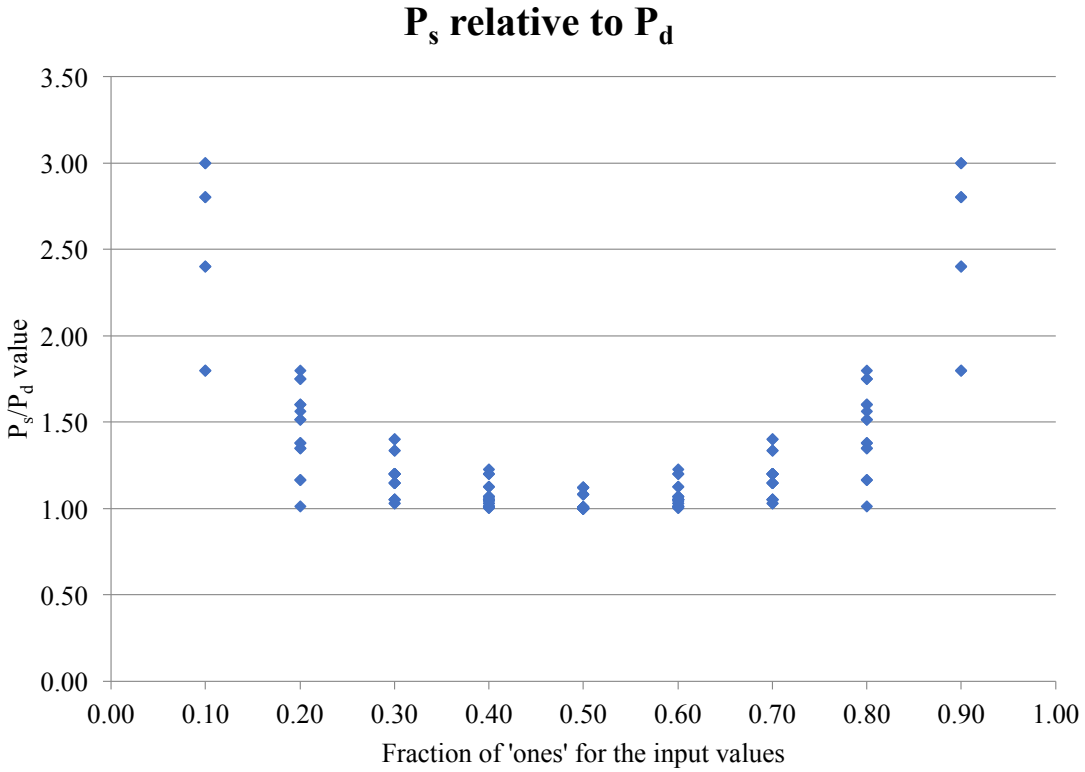

Figure 4.4: Relative power of SV (P_s) vs. DV (P_d) architectures for varying input ratios.

4.1.3 Differential current with 1/High Z inputs

Figure 4.5 is an example of a differential current architecture using 1/High Z inputs (DZ). When the input = 0, the output of the row driver circuit is a high impedance node (High Z), rather than V_{ss}. Every input and bias is connected to both a positive rail and negative rail conductance. In contrast to the other two architectures, the outputs are two currents, each summed separately on its own bit line. The two currents are compared using a differential current comparator to make the threshold gate. Only input = 1 conditions create current (and consume power). The CSlite malware detection application was analyzed using this architecture [24, 32].

Repeating the analysis using our previous assumptions, we find

$$I^{+} = n * V_{\text{mem}} * G_1 * \alpha_{11} \tag{4.23}$$

$$I^{-} = n * V_{\text{mem}} * G_1 * \alpha_{10} \tag{4.24}$$

where V_{mem} is the voltage across the memristor. Assuming V^{+} and V^{-} are held relatively constant,

$$V_{\text{mem}} = \beta * V_{\text{dd}} \tag{4.25}$$

the value of β can be estimated by examining simulations from our comparator design, which will be described and analyzed later in this chapter. It will normally

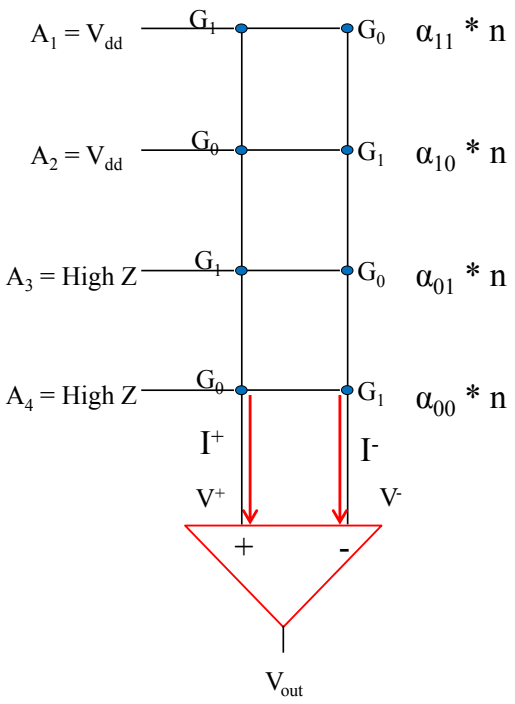

Figure 4.5: Differential current architecture with 1/HighZ inputs(DZ).

be 0.25 - 0.30. Therefore

$$P^{+} = n * G_1 * {V_{\mathrm{dd}}}^2 * \alpha_{11} * \beta \tag{4.26}$$

$$P^{-} = n * G_1 * {V_{\mathrm{dd}}}^2 * \alpha_{10} * \beta \tag{4.27}$$

$$P_{\mathrm{z}} = (n * G_1 * {V_{\mathrm{dd}}}^2) * [(\alpha_{10} + \alpha_{10}) * \beta] \tag{4.28}$$

Typically, $P_d > P_z$ as the fraction of high inputs is reduced (see Figure 4.6, which assumes $\beta = 0.25$).

Again using the same values (fraction of ones) for each crossbar array in a neural network running the MNIST application (5%, 5%, 20%, 20%, 62%), we find

$P_d = 2.98 * P_z$, and $P_s = 6.73 * P_z$.

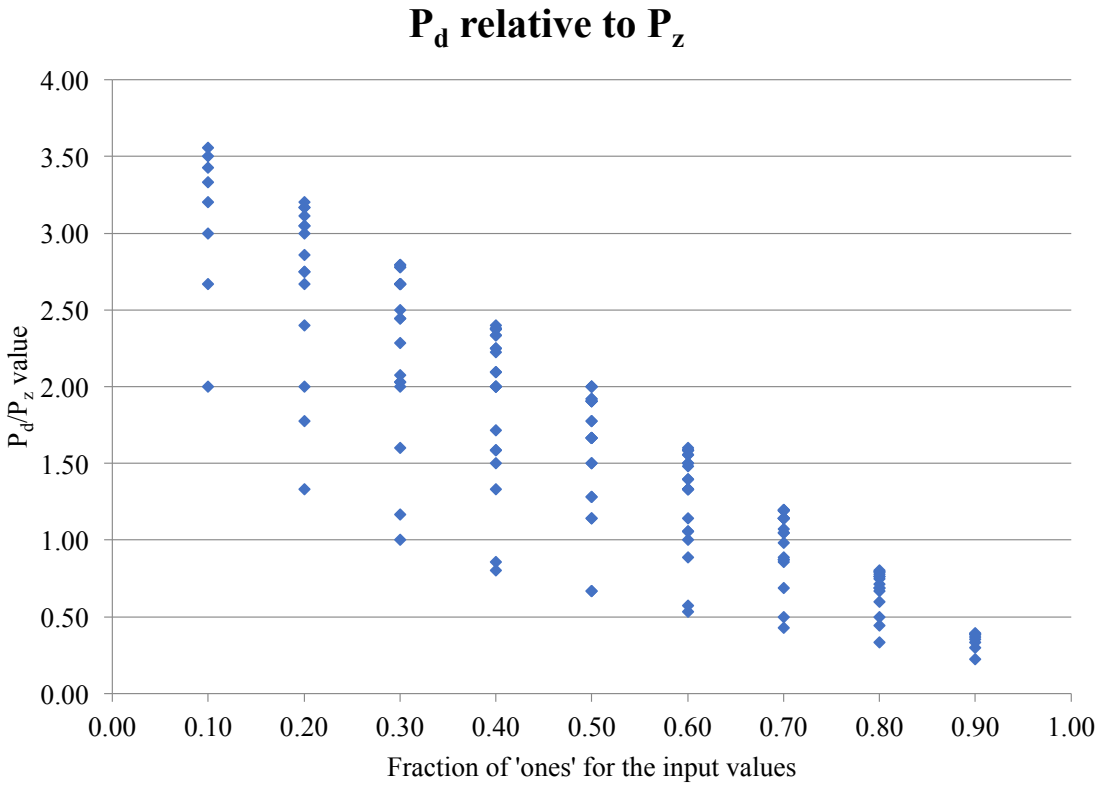

Figure 4.6: Relative power of DV (P_d) vs. DZ (P_z) architectures for varying input ratios.

This analysis indicates the 1/High Z differential current architecture (DZ) is the most power efficient. It also has a desirable property for circuit analysis: the current for each input is directly proportional to the weighted input for the neuron. This enables certain mathematical properties of the TGN to be verified as correctly implemented via simple analysis or simulation of the circuits. The DZ architecture will also be an important factor in the development of the tile feature, described in the next chapter.

4.2 Input circuits

The input circuits have 4 main functions: address decoding, input data multiplexing, a latch to enable short-term data storage, and row drivers to transfer data into the array. For our architecture, these are all based on traditional designs [64]; the only significant design choice to be made is associated with the row driver.

4.2.1 Row driver

The row driver (see Figure 4.7) must supply current to all inputs in a given row, and also controls the access PFET in the unit cell; the design shown implements the 1/High Z architecture.

The row driver needs to supply current for all its neurons (m), without a large voltage drop

$$I_{\text{row}} = m * w_{\text{ave}} * G_1 * V_{\text{mem}} \tag{4.29}$$

For m = 64, w_{ave} = 4, G_1 = 1 μS, V_{mem} = 0.2; we find I_{row} = 51.2 μA. This is assumed to be a worst-case scenario.

For a PFET device in 45 nm, ΔV_{ds} will be 1.94 mV per μA if W = 200 nm, L = 45 nm. (See Table 4.1). The device is biased in the linear region, so linear extrapolations of this are very accurate. Example: for I_{row} = 51.2 μA, W = 1200 nm, estimated ΔV_{ds} = 16.6 mV, actual ΔV_{ds} = 16.7 mV. So the device size can be specified by estimating the worst case current required, and the maximum ΔV_{ds} allowed for that current. Our design with 64 neurons will have a row driver

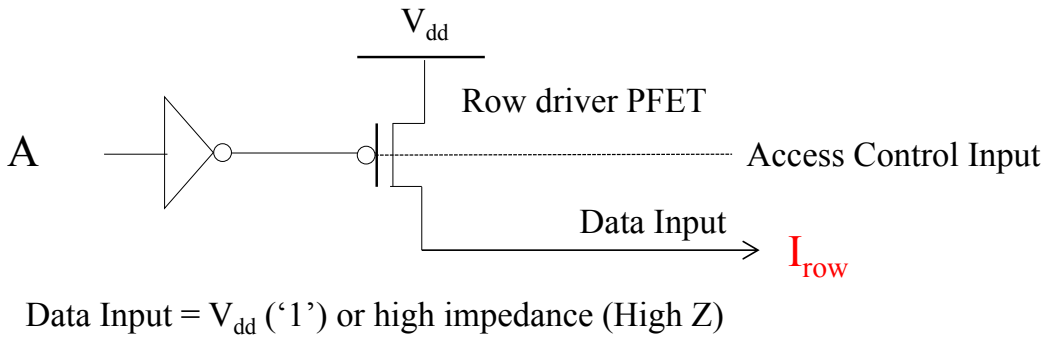

Figure 4.7: Row driver circuit schematic that implements the 1/High Z input.

Table 4.1: I_{ds} (in μA) as a function of allowed ΔV_{ds} and PFET width (L = 45 nm).

Width (nm)	ΔV_{ds} = 5 mV	ΔV_{ds} = 10 mV	ΔV_{ds} = 20 mV
200	2.578 μA	5.137	10.170
500	6.566	13.083	25.905
1000	13.210	26.323	52.124
2000	26.499	52.804	104.560 μA

with W/L = 1200/45 nm; for other array sizes, the driver width can be adjusted appropriately. There will be control PFETs in the comparator; they will be sized in a similar fashion; details will be discussed as part of the comparator design section.

4.3 Unit cell

The next issue explored is the unit cell design; we have chosen to use a 1T1M unit cell (circuit schematic, layout, and layer stack up are shown in Figure 4.8, Figure 4.9, and Figure 4.10). Figure 4.9 actually shows two unit cells; the dashed box indicates the size (60 F^2) of a single unit cell.

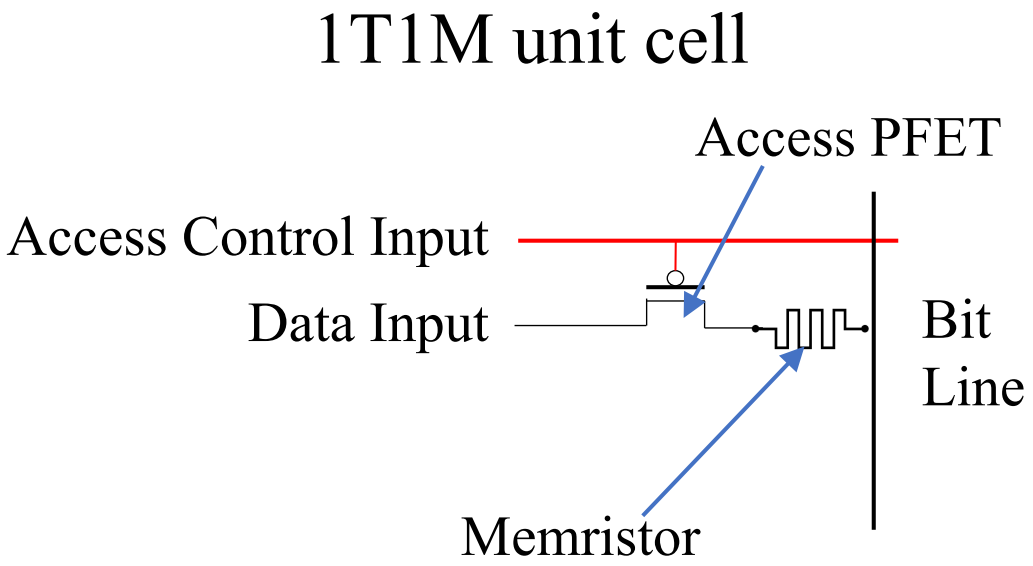

Figure 4.8: Unit cell circuit schematic.

The value of the 1T1M design is that the access PFET eliminates sneak path issues [65] and enables highly precise programming of conductances into the mem-

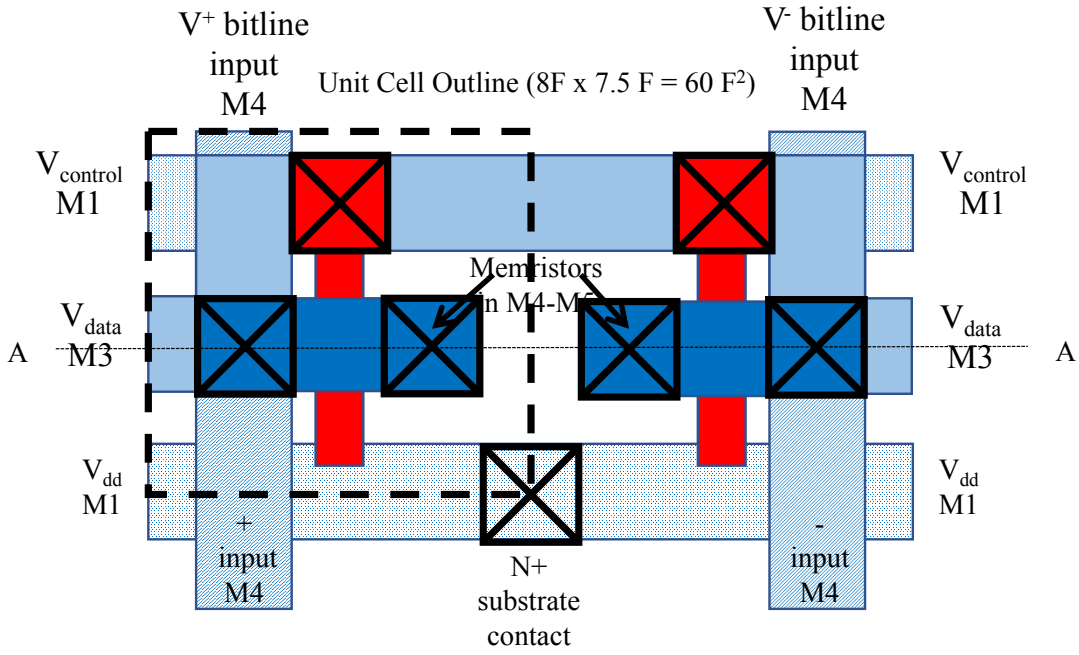

Figure 4.9: Unit cell layout.

ristor [66]. It comes at a cost of greatly increased area for the unit cell (60 F^2 for the layout shown vs. 4 F^2 for a memristor only design). We will show later that the overall area cost is significantly lower, given the size of the peripheral circuits. This is true even if you assume the peripheral circuits can be placed underneath the memristor unit cells in the 4 F^2 layout.

The access PFET is sized to be the largest possible within the unit cell layout (W/L = 90/45 nm). Here the requirement is to understand the limitations of this design choice. Our array assumes integer weight values up to 32 are possible (w_{max} = 32, corresponding to G_{max} = 32 μS), based on the measured performance of TaOx

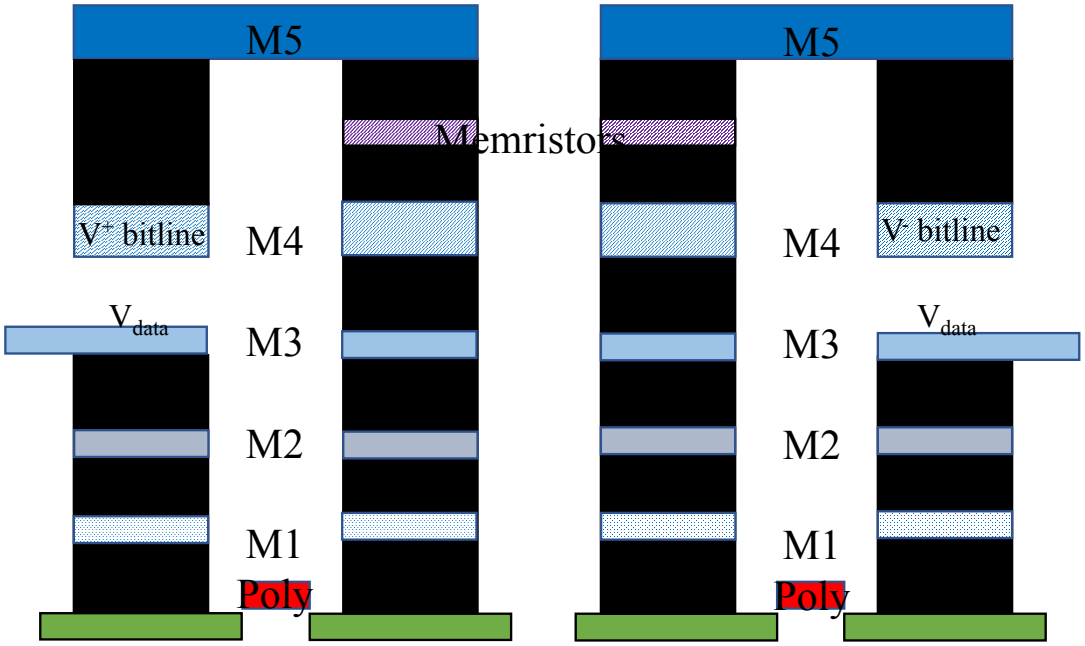

Figure 4.10: Unit cell layer stack (cross-section at A – A in Figure 4.9).

devices [29]. The worst case current flow (assuming $V_{mem} = 0.2$ V) is:

$$I = w_{max} * G_1 * V_{mem} = 6.4\mu A \tag{4.30}$$

Using LTspice [60] and 45nm technology device parameters for this design [61] [62], $\Delta V_{ds} = 27.6$ mV.

A second issue that needs consideration is that the access FET has a finite conductance, so the programmed conductance should be adjusted to compensate for this.

$$G_{PFET} \backsim 218\mu S \tag{4.31}$$

so

$$G_{\text{eff}} = (G_{\text{mem}} * G_{\text{PFET}})/(G_{\text{mem}} + G_{\text{PFET}}) \tag{4.32}$$

or

$$G_{\text{mem}} = (G_{\text{eff}} * G_{\text{PFET}})/(G_{\text{PFET}} - G_{\text{eff}}) \tag{4.33}$$

where G_{mem} is the value actually programmed into the memristor, and G_{eff} is the desired value.

It should be noted that the effect of G_{PFET} is small until G_{mem} is fairly large. Without correction, if

$$G_{\text{mem}} = 1\mu S, I = 199.1nA \tag{4.34}$$

If

$$G_{\text{mem}} = 8\mu S, I = 1543.4nA \tag{4.35}$$

This would cause a 3% error in the actual weighting ratio (7.75/1) from the desired weighting (8/1) if not corrected.

It should also be noted that the correction only works if $G_{\text{eff}} << G_{\text{PFET}}$; if $G_{\text{eff}} \approx G_{\text{PFET}}$, then the correction becomes inaccurate very rapidly; if $G_{\text{eff}} \geq G_{\text{PFET}}$ it is not possible to correct at all. If $G_{\text{eff}} \geq G_{\text{PFET}}$ is needed, then one (or more) of these 4 options could be used:

- The access PFET can be widened to increase its conductance (at a cost of larger unit cell area)

- A control input signal < 0.0 V can be used to increase gate drive (requires an extra voltage island)
- A PFET with a lower threshold voltage can be used (requires the device fabrication alternative to be available)
- Multiple inputs can be used, each with a smaller conductance; when combined, they equal G_{eff} (increases the array size)

4.4 Array size limitations

There are a variety of practical issues with implementing memristor crossbar arrays that can limit their size. Parasitic effects on timing and accuracy, along with programming precision, are explored below.

4.4.1 Parasitic effects

For the 45 nm process we are using, the nominal wire resistance and capacitance values are:

C = 0.2 fF/μm of length for all lines

R = 0.214 Ω/μm of length for long lines

R_s = 0.239 Ω/square for short lines

These are consistent with other published values [67]. From the 1T1M layout, we calculate the following values for each unit cell:

Input line to the access FET: $R_{input} = 0.956\ \Omega$, $C_{input} = 72$ aF

Data line across a row: $R_{horizontal} = 0.956\ \Omega$, $C_{horizontal} = 72$ aF

Bit line down the column: $R_{vertical} = 0.896\ \Omega$, $C_{vertical} = 67.5$ aF

In order to accurately analyze the transient conditions in the array, all relevant parasitics should be incorporated. These include resistance and capacitance on the control input signal, the input data lines, and the output bit lines, along with any memristor parasitics. Figure 4.11 depicts a nominal 2 x 1 array, with parasitic elements included.

Memristor parasitics Figure 4.12 is the circuit model for the memristor. For our simulations, we use the memristor model in [23], with some minor modifications described below. This model has been validated with various memristor devices, and the parameters can be extracted using a straightforward test procedure [68]. It is also less computationally complex than other models [68], which will be important for simulating large memristor arrays. One minor issue with this model is that under certain simulation conditions the device can be driven to zero conductance (or infinite resistance). We correct for this by placing an additional conductance in parallel with the memristor, the value is the off-state conductance (G_0) on measured devices. For our analysis we use $G_0 = 0.01$ μS [63]. To account for the contact resistance of the memristor, a 100 Ω resistor is placed in series with the device as part of the memristor model.

Our layout enables a memristor of size 4 F^2 to be fabricated; this may be

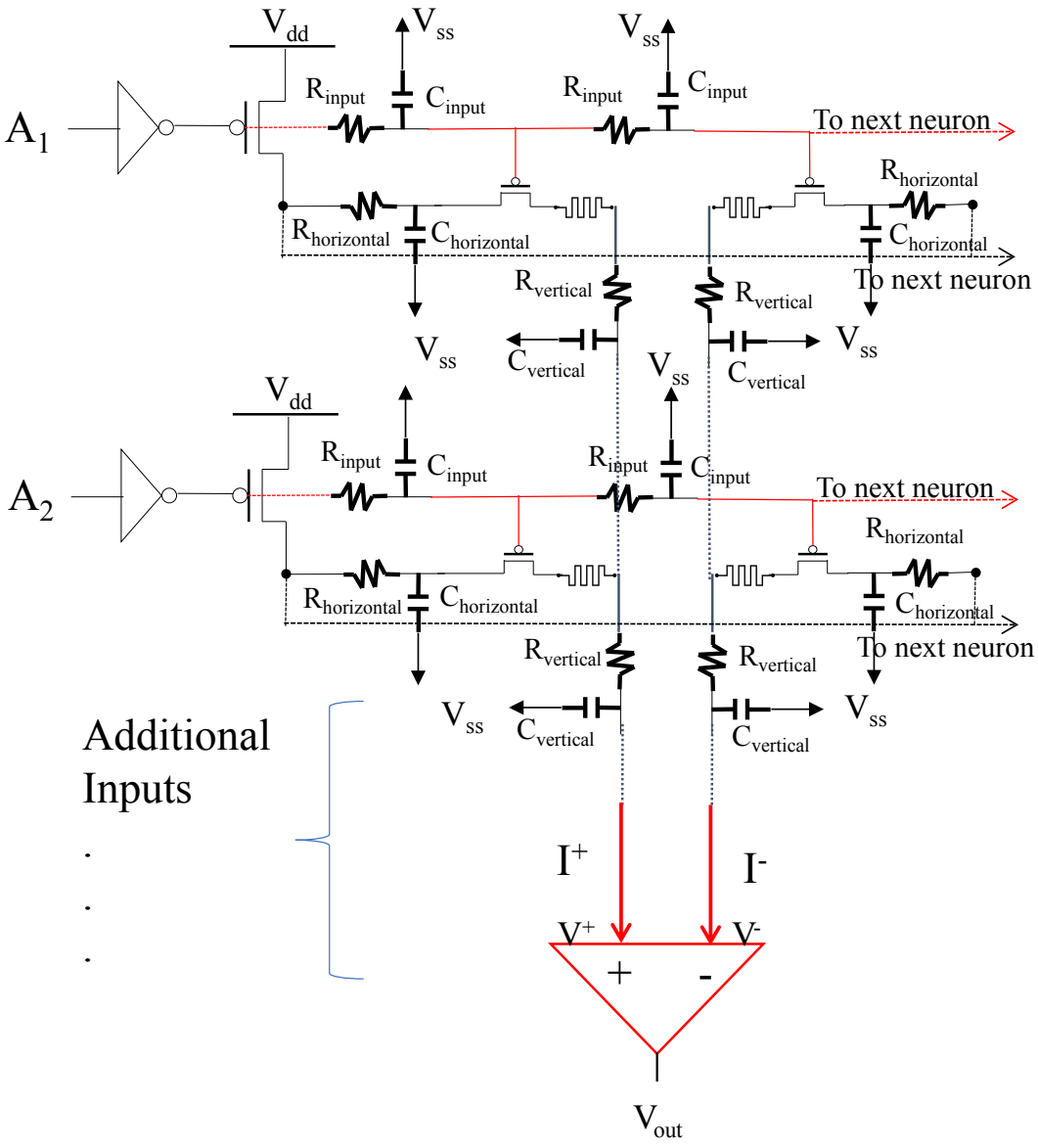

Figure 4.11: Nominal 2 x 1 array, including parasitic elements.

desirable for manufacturing and yield reasons. However, we have found that the memristor capacitance can have an impact on the length of the transient response of the bit line to changes in the input vector (up to 1 ns), so this capacitance should be minimized if possible (reduce area and increase dielectric thickness).

$$C_{\text{mem}} = (\epsilon_0 * K * A_{\text{mem}})/t_{\text{mem}} \tag{4.36}$$

The nominal value for a TaOx memristor device (K = 25), of minimum size (45 nm x 45 nm) and with a nominal dielectric thickness of 20 nm, is 22.4 aF. Actual capacitances can be found using straightforward measurement techniques on fabricated devices [69]. This capacitance is placed in parallel with the memristor in the device model. With this capacitance value and a 1T1M unit cell, transient effects are on the order of 200 - 250 ps.

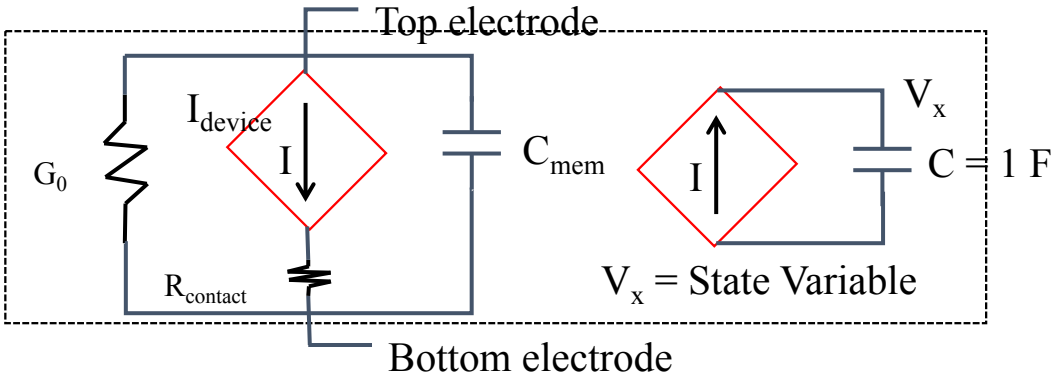

Figure 4.12: Memristor model with extensions.

Timing effects of wire parasitics The RC time constant across a neuron is very small

$$\tau = 2 * R_{\text{horizontal}} * C_{\text{horizontal}} = 1.4x10^{-4} \text{ ps} \tag{4.37}$$

Even if we assume 10τ (1.4 x 10^{-3} ps) is the timing cost for a neuron, the effect is small; τ= 10 ps when m = 7264.

4.4.2 Limitations on the maximum number of inputs

Absolute voltage drop down the bit line The total voltage drop down the bit line can be estimated by assuming we have equal weights for each input; the conductance for each input will be G_{ave}. Then:

$$\Delta V_{\text{bit}} = \sum_{j=1}^{n} I_{\text{j}} * R_{\text{j}} \tag{4.38}$$

since

$$R_{\text{j}} = R_{\text{vertical}} \tag{4.39}$$

and

$$I_{\text{j}} = I_{\text{unit}} * j = G_{\text{ave}} * V_{\text{mem}} * j \tag{4.40}$$

then

$$\Delta V_{\text{bit}} = G_{\text{ave}} * V_{\text{mem}} * R_{\text{vertical}} * \sum_{j=1}^{n} j \tag{4.41}$$

$$\Delta V_{\text{bit}} = I_{\text{unit}} * R_{\text{vertical}} * n * (n+1)/2 \tag{4.42}$$

For large n, we can use the approximation

$$n^2 \backsim n * (n+1) \tag{4.43}$$

so

$$\Delta V_{\text{bit}} = G_{\text{ave}} * V_{\text{mem}} * R_{\text{vertical}} * n^2/2 \tag{4.44}$$

or

$$n_{\max} = [2 * \Delta V_{\text{bit}}/(G_{\text{ave}} * V_{\text{mem}} * R_{\text{vertical}})]^{1/2} \tag{4.45}$$

Using the following values:

$G_{ave} = 2$ μS, $V_{mem} = 0.2$ V, $R_{vertical} = 0.896$ Ω

We can calculate the n_{max} that will keep ΔV_{bit} below any desired maximum value (see Table 4.2). This suggests that the absolute voltage drop on the wire might place a practical limit of the number of inputs.

Table 4.2: n_{max} as a function of the maximum desired ΔV_{bit}

ΔV_{bit} (mV)	n_{max}
1	74
2	105
5	167
10	236
20	334
50	528

Unequal bit line voltage drop and its effect on accuracy The next analysis considers the following scenario:

The first row has a high positive weight

$$G_{\mathrm{a}}{}^{+} = G_1 * w_{\max} \tag{4.46}$$

The last row has a high negative weight

$$G_{\mathrm{z}}{}^{-} = G_1 * (w_{\max} - 1) \tag{4.47}$$

The rows between $G_{\mathrm{a}}{}^{+}$ and $G_{\mathrm{z}}{}^{-}$ are assumed to have alternating weights between +1 and -1.

Because the current through $G_{\mathrm{a}}{}^{+}$ sees higher total bit line resistance than $G_{\mathrm{z}}{}^{-}$, this input will have a slightly smaller V_{mem}, and therefore less current than expected. We consider the case where this voltage difference is just enough to make the two currents equal, even though they have differing weights.

Expected

$$I_{\mathrm{a}} = G_{\mathrm{a}} * V_{\mathrm{mem,a}} \tag{4.48}$$

Actual

$$I_{\mathrm{a}}{}^{*} = G_{\mathrm{a}} * V_{\mathrm{mem,a}}{}^{*} = G_{\mathrm{a}} * (V_{\mathrm{mem,a}} - \Delta V_{\mathrm{bit,a}}) \tag{4.49}$$

$$I_{\mathrm{z}} = G_{\mathrm{z}} * V_{\mathrm{mem,z}} \tag{4.50}$$

We will approximate V_{mem} as the nominal voltage across the memristors

$$V_{\mathrm{mem,a}} \backsim V_{\mathrm{mem,z}} = V_{\mathrm{mem}} \tag{4.51}$$

For

$$I_{\mathrm{a}}{}^{*} = I_{\mathrm{z}} \tag{4.52}$$

$$G_{\mathrm{a}} * (V_{\mathrm{mem,a}} - \Delta V_{\mathrm{bit,a}}) = G_{\mathrm{z}} * V_{\mathrm{mem,z}} \tag{4.53}$$

and

$$\Delta I_{\mathrm{a}} = I_{\mathrm{a}} - I_{\mathrm{a}}{}^{*} = G_{\mathrm{a}} * \Delta V_{\mathrm{bit,a}} \tag{4.54}$$

$$\Delta V_{\mathrm{bit,a}} \backsim R_{\mathrm{vertical}} * G_{\mathrm{ave}} * V_{\mathrm{mem}} * n^2/2 \tag{4.55}$$

so

$$\Delta I_{\mathrm{a}} = R_{\mathrm{vertical}} * G_{\mathrm{a}} * G_{\mathrm{ave}} * V_{\mathrm{mem}} * n^2/2 \tag{4.56}$$

We will need to adjust $\mathrm{G_a}$ so that it counterbalances the $\Delta \mathrm{V_{bit,a}}$ effect

$$G_{\mathrm{a,eff}} = G_{\mathrm{a}} + \Delta G_{\mathrm{a}} \tag{4.57}$$

$$\Delta G_{\mathrm{a}} = \Delta I_{\mathrm{a}}/V_{\mathrm{mem}} = R_{\mathrm{vertical}} * G_{\mathrm{a}} * G_{\mathrm{ave}} * n^2/2 \tag{4.58}$$

or

$$n_{\mathrm{max}} = (2 * \Delta G_{\mathrm{a}}/R_{\mathrm{vertical}} * G_{\mathrm{a}} * G_{\mathrm{ave}})^{1/2} \tag{4.59}$$

Using the previous values

$\mathrm{R_{vertical}} = 0.896\ \Omega$

$\mathrm{G_a} = \mathrm{G_{max}} = 32\ \mu\mathrm{S}$

$\mathrm{G_{ave}} = 2\ \mu\mathrm{S}$

Table 4.3 shows that this effect, if not corrected, places a severe limit on the array size. However, the correction value (ΔG_a) can be calculated from values that are known in advance, so this adjustment can be done.

Table 4.3: n_{max} as a function of the maximum acceptable uncorrected ΔG_a

ΔG_a (μS)	n_{max}
0.1	59
0.2	83
0.5	132
1.0	186

If $w_{max} = 8$, not 32, all the n values are doubled. In general, *all* of the conductance values will have to be adjusted in this way. For a given row and conductance value (G_k), the adjustment is as follows:

$$\Delta G_k = G_k * [(G_{ave} * R_{vertical}/2) * (n + k) * (n + 1 - k)] \tag{4.60}$$

Again, all of the values are known in advance, so calculating this adjustment is practical. Here are a couple of examples:

Let n = 256, k = 1, and G_k = 3 μS, G_{ave} = 0.4 μS (this would a possible situation for MNIST)

$$\Delta G_k = 0.035\ \mu S \tag{4.61}$$

This is probably too fine an adjustment to the memristor value to be practical.

Let n = 256, k = 200, and G_k = 32 μS, G_{ave} = 4.0 μS (this would be an outlier case)

$$\Delta G_k = 1.49\ \mu S \tag{4.62}$$

Effect of memristor programming precision How precisely do the memristor devices need to be programmed to ensure functionality? How large can an array be before it incorrectly functions due to inaccurate programming? Consider a majority function, where $V_{out} = V_{dd}$ if the number of high inputs with G = +1 (N^+) is greater than for G = -1 (N^-). If the conductance values programmed into the memristors are not precise, it is possible for the total G^+ to be smaller than G^- even if $N^+ > N^-$, and the output will be incorrect. We will assume an expected value ($\mu = G_{ave}$) that can deviate randomly due to a finite programming precision; ($3 * \sigma/\mu$) is defined as the programming precision.

We define μ_{xy} as the actual difference between the means of the two columns in the neuron, and G_{ave} as the desired mean for each column.

We will assume

$$G_{ave}{}^{+} = G_{ave} - \mu_{xy}/2 \tag{4.63}$$

and

$$G_{ave}{}^{-} = G_{ave} + \mu_{xy}/2 \tag{4.64}$$

then

$$G_{ave}{}^{-} - G_{ave}{}^{+} = \mu_{xy} \tag{4.65}$$

If the average value is known but varies randomly, then μ_{xy} will have a mean of zero and

$$\sigma_{xy} = [\sigma_x{}^2/(n_x) + \sigma_y{}^2/(n_y)]^{1/2} \tag{4.66}$$

For this analysis, we assume

$$\sigma_{\mathrm{x}} = \sigma_{\mathrm{y}} = \sigma \tag{4.67}$$

and

$$n_{\mathrm{x}} \backsim n_{\mathrm{y}} = n/2 \tag{4.68}$$

so

$$\sigma_{\mathrm{xy}} = [2 * 2 * \sigma^2/n]^{1/2} = 2 * \sigma/n^{1/2} \tag{4.69}$$

In order for this random variation to create inaccuracy, then

$$G_{\mathrm{ave}}{}^{+} * (n+1)/2 = G_{\mathrm{ave}}{}^{-} * (n-1)/2 \tag{4.70}$$

$$n * (G_{\mathrm{ave}}{}^{+} - G_{\mathrm{ave}}{}^{-}) + (G_{\mathrm{ave}}{}^{+} + G_{\mathrm{ave}}{}^{-}) = 0, \tag{4.71}$$

or

$$2 * G_{\mathrm{ave}} = n * \mu_{\mathrm{xy}}; \mu_{\mathrm{xy}} = 2 * G_{\mathrm{ave}}/n \tag{4.72}$$

To ensure this is a rare occurrence, the value of μ_{xy} needed to cause this problem should be at least 3 standard deviations from the mean ($\mu_{\mathrm{xy}} = 0$ is the mean). Under this condition:

$$\mu_{\mathrm{xy}} = 3 * \sigma_{\mathrm{xy}} \tag{4.73}$$

Substituting for each side of this equation from above

$$2 * G_{\mathrm{ave}}/n = 6 * \sigma/n^{1/2} \tag{4.74}$$

since

$$\mu = G_{\text{ave}} \tag{4.75}$$

$$\mu/n = 3 * \sigma/n^{1/2} \tag{4.76}$$

$$[(3 * \sigma)/\mu] = n^{-1/2} \tag{4.77}$$

or

$$n = [(3 * \sigma)/\mu]^{-2} \tag{4.78}$$

So if we know the programming precision, that determines the maximum n; or the n desired defines the programming precision required. Table 4.4 provides details.

Table 4.4: Required memristor programming precision as a function of the number of inputs.

n	Precision (σ/μ)	3 σ precision in %
32	0.059	17.6%
64	0.042	12.5%
128	0.029	8.8%
256	0.021	6.25%
512	0.015	4.4%
1024	0.010	3.125%

There is published information [29] for a programming technique that achieves 1% precision; for this value the maximum array can have $\approx (.01)^{-2} = 10{,}000$ rows. This data indicates programming precision will not be a practical limitation on the number of inputs in an array.

There is one additional limit on the size of the array (number of inputs): that is the

amount of input current the comparator can practically handle. This limit will be discussed as part of the comparator design and analysis.

4.4.3 Limitations on the maximum number of neurons

The analysis for this effect will be similar for the number of inputs, with a couple of exceptions:

An array does not necessarily require a large number of neurons, but a large number of inputs can be very useful (the impact of minimizing the array size is less significant here);

The average conductance (G_{ave}) within a neuron is likely to be larger than across many neurons in a given row;

There are two unit cells per neuron, not one per bit line as before.

Absolute voltage drop across the array We will use the same values from the row driver analysis

G_{ave} = 1 μS, V_{mem} = 0.2 V, $R_{horizontal}$ = 0.956 Ω

The voltage drop from the input driver to the last neuron in the array can be estimated assuming an equal current goes into each neuron in a row. Similar to the previous analysis:

$$\Delta V_{row} = G_{ave} * V_{mem} * R_{horizontal} * m^2 \tag{4.79}$$

or

$$m_{\max} = [\Delta V_{\text{row}}/(G_{\text{ave}} * V_{\text{mem}} * R_{\text{horizontal}})]^{1/2} \tag{4.80}$$

We can calculate the m_{max} to keep ΔV_{row} below any desired maximum value (see Table 4.5). This suggests that absolute voltage drop across the row will probably not limit the array size.

Table 4.5: m_{max} as a function of the ΔV_{row} allowed

ΔV_{row} (mV)	m_{max}
1	72
2	102
5	161
10	228
20	323
50	511

Unequal voltage drop across a row and its effect on accuracy If two rows have differing G_{ave} values, then the voltage drop across two memristors in different rows will be slightly different, causing the currents to be slightly different. A worst-case scenario would be the following:

Looking at the last neuron in the array (largest voltage difference due to G_{ave})

One row has

$$G_{\text{a}}{}^{+} = G_1 * w_{\max} \tag{4.81}$$

a different row has

$$G_{\text{b}}{}^{-} = G_1 * (w_{\max} - 1) \tag{4.82}$$

For this worst-case scenario

$$I^{+} = I^{-} \tag{4.83}$$

(and unequal weights give equal currents) if

$$V_{\text{mem}}{}^{+} = V_{\text{mem}}{}^{-} * (w_{\max} - 1)/w_{\max} \tag{4.84}$$

or

$$\Delta V_{\text{mem}} = V_{\text{mem,b}} - V_{\text{mem,a}} = V_{\text{mem,b}}/w_{\max} \backsim V_{\text{mem}}/w_{\max} \tag{4.85}$$

Again using V_{mem} as the nominal voltage across the memristors

$$(V_{\text{mem,a}} \backsim V_{\text{mem,b}} = V_{\text{mem}}) \tag{4.86}$$

The voltage drops can be estimated:

$$\Delta V_{\text{row,a}} \backsim G_{\text{ave,a}} * V_{\text{mem}} * R_{\text{horizontal}} * m^2 \tag{4.87}$$

$$\Delta V_{\text{row,b}} \backsim G_{\text{ave,b}} * V_{\text{mem}} * R_{\text{horizontal}} * m^2 \tag{4.88}$$

$$\Delta V_{\text{mem}} = \Delta V_{\text{row,a}} - \Delta V_{\text{row,b}} \backsim R_{\text{horizontal}} * m^2 * V_{\text{mem}} * \Delta G_{\text{ave}} \tag{4.89}$$

where

$$\Delta G_{\text{ave}} = G_{\text{ave,a}} - G_{\text{ave,b}} \tag{4.90}$$

and the parasitic resistance will cause a problem if

$$R_{\text{horizontal}} * m^2 * V_{\text{mem}} * \Delta G_{\text{ave}} = V_{\text{mem}}/w_{\max} \tag{4.91}$$

or

$$\Delta G_{\text{ave}} = (w_{\max} * R_{\text{horizontal}} * m^2)^{-1} \tag{4.92}$$

or

$$m = (w_{\max} * R_{\text{horizontal}} * \Delta G_{\text{ave}})^{-1/2} \tag{4.93}$$

Realize ΔG_{ave} is likely to drop as m grows larger, since it is the difference of the average G values.

Let us assume the following

$$w_{\max} = 32 \ (G_{\max} = 32 \ \mu S) \tag{4.94}$$

Table 4.6 shows that this effect will be similar to the voltage drop (m_{max} can be reasonably large, since ΔG_{ave} is very likely to be less than 1.0).

If $w_{max} = 8$, not 32, all the m_{max} values are doubled.

The weight for memristors across a row can also be adjusted, along the lines of the weight adjustment described earlier.

$\Delta G_j = G_j * [(G_{ave} * R_{horizontal}) * (j+1) * (j)]$

Let j = 64, G_j = 3 μS, and G_{ave} = 0.1μS (this is a possible situation for

Table 4.6: m_{max} as a function of ΔG_{ave}

ΔG_{ave} (μS)	m_{max}
0.1	572
0.2	404
0.5	255
1.0	180
2.0	127
5.0	80

MNIST), $\Delta G_j = 0.001$ μS; this is too fine an adjustment to the memristor value to be practical.

Let j = 64, $G_j = 32$ μS, and $G_{ave} = 0.5$ μS (this would be an outlier case), $\Delta G_j = 0.06$ μS

The need for weight adjustments in this case is much less. There are two significant reasons for this: the number of neurons in an array is expected to be much smaller than the number of inputs, and G_{ave} across a row will typically be much smaller than within a neuron.

The weight adjustment for any memristor in the array, taking into account both parasitic effects, is:

$$\Delta G_{j,k} = \Delta G_j + \Delta G_k \tag{4.95}$$

$$\Delta G_{j,k} = G_{j,k} * [(G_{ave,j} * R_{horizontal}) * (j+1) * (j) + (G_{ave,k} * R_{vertical}/2) * (n+k) * (n+1-k)] \tag{4.96}$$

This adjustment matrix can be easily found once all the initial weights have been calculated.

Minimizing inaccuracy due to practical effects The analysis for the effects of the parasitics strongly suggest that mapping the weights to the physical array should put the highest weights in the first columns and the last rows of the array when possible, with the lowest weights in the last columns and the first rows (see Figure 4.13). For a given neuron, high positive and negative weights should be grouped in nearby rows if possible. This approach will minimize the weight adjustments needed, and will also minimize the likelihood of an incorrect neuron output due to any inaccuracies in the weight adjustment (either through the use of the above approximate adjustments or imprecision in the memristor programming). This also shows the potential value of on-chip learning: the weights will automatically be adjusted correctly for various effects as part of the process.

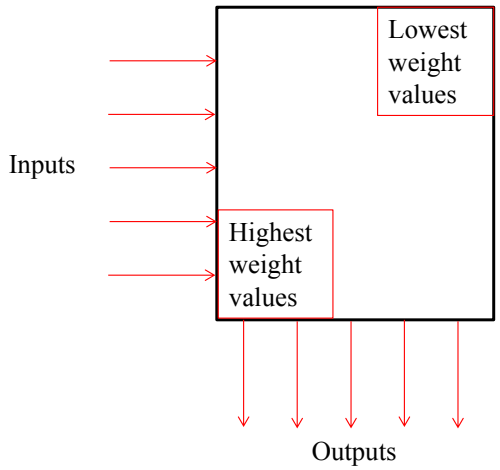

Figure 4.13: Mapping of weights into the array to minimize errors due to parasitic effects.

4.5 Comparator design

The comparator is an important element of the architecture [22]. The speed of this circuit is one of the main factors in estimating the neural network throughput (the router network is another important factor). Since the comparator is required to sink the currents from the array, it has to be large enough to handle the total current while still being able to discriminate a minimum difference ($\Delta G_{min} = 1$ μS). This can have a major impact on the overall area and timing, and can limit the number of inputs allowed into a single neuron. It is also a significant consumer of the overall power. Our comparator architecture is shown in Figure 4.14. The two input currents are transduced into differential voltages, and the difference is amplified to create an output that can be buffered and latched for driving the data onto the communication network. The desired design will be compact, low power, and fast. If this can be achieved, a comparator circuit can be used for each neuron, instead of multiplexing as is often used [22]. We have designed a comparator that meets these requirements; assuming 45 nm processing technology, it is:

Compact (≈ 55 μm^2)

Low power (≈ 15 μW)

Fast (≈ 250 MHz)

4.5.1 Basic architecture

The basic design is easily described; Figure 4.15 has a schematic of the essential elements. The input stage is built using an FET with drain and gate connected to create a diode-connected FET (one each for the positive and negative inputs). The amplifier stage uses simple 5 FET differential voltage amplifiers. Two amplifiers are used so that the output voltage of the amplifier stage is driven to $\approx V_{dd}$ or V_{ss} as needed. The output stage (not shown) uses four inverters to drive the output load, which consists of wire capacitance and the input gate of the router/switch. Data latching is enabled by the use of a controllable transmission gate between inverters 2 and 3.

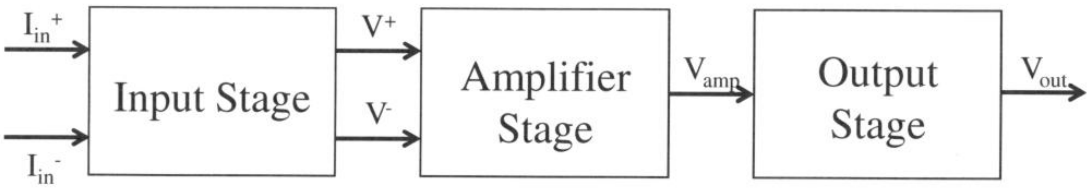

Figure 4.14: Comparator Architecture.

For traditional applications this design would be impractical for at least two reasons:

1. The maximum current input and minimum current difference that can be sensed are inversely related, limiting the operating range of the design;

2. The design is very sensitive to device mismatches (such as small V_t differences).

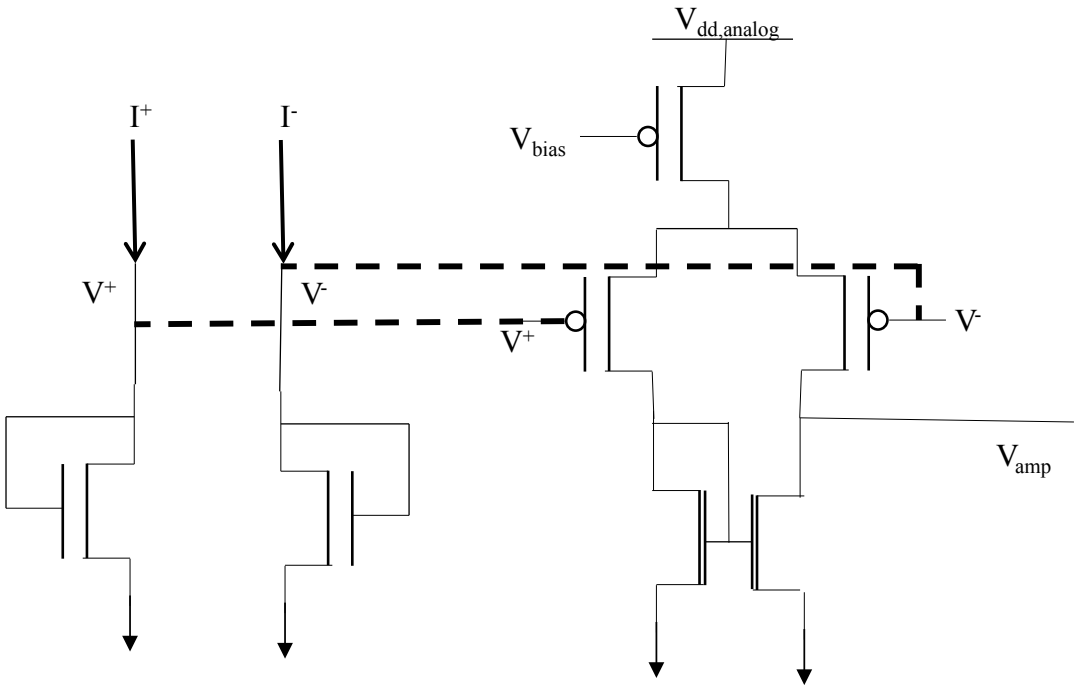

Figure 4.15: Input and amplifier stage of the comparator (simplified schematic).

4.5.2 Design modifications and test procedures to handle mismatches

In using the comparator for neural net applications, we can take advantage of some conditions that are not typically available with other applications.

1. The weights in the neural network must be programmed, and are therefore known in advance; for any given set of weights, the comparator needs to operate correctly in only a subset of the total range required.

2. The memristors are programmable conductance devices that can be used to ensure correct operation even under device and parameter mismatch conditions.

3. Corner case operating conditions that create inaccurate functionality can be tolerated if they rarely occur; the overall neural net accuracy can still be very good (perhaps unaffected).

We take advantage of this knowledge by modifying the simple comparator design (see Figure 4.16). The first modification is to use multiple diode-connected FETs in parallel for each of the two inputs. These parallel FETs have control gates that enable one or more diodes to be active, depending on the total weight for the neuron. The control PFETS have W/L = 270/45 nm; using the analysis from the row driver circuit, we estimate the voltage drop will be < 5 mV. The total weight (maximum possible conductance) is known in advance since the memristors need to be programmed. This enables a proper number of diodes to be used, enabling the design to operate in its desired range under most input conditions. The specifics of

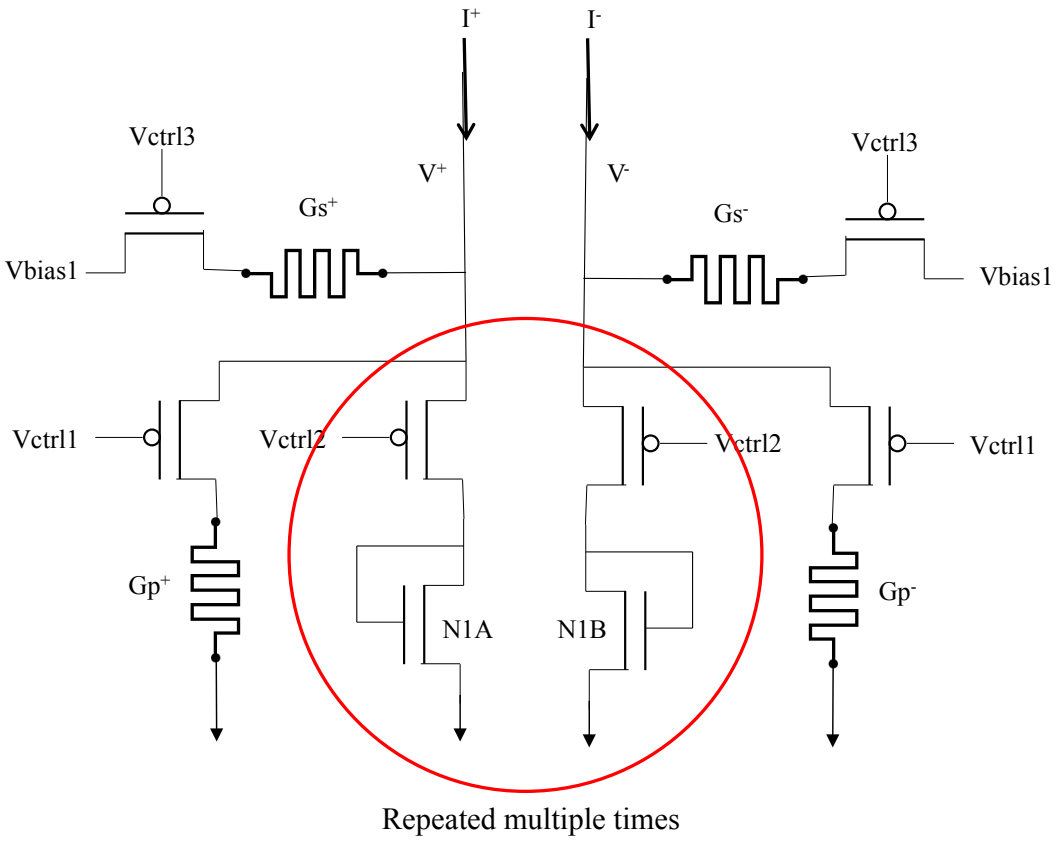

Figure 4.16: Modified input stage design to correct for parameter and device mismatches.

this will be discussed later. The second modification is to include additional memristors in the design. One set of two memristors (G_s^+, G_s^-) is connected to the bit lines (like a bias or data input). The other set of two memristors (G_p^+, G_p^-) is in parallel with the diodes. These memristors can be programmed in a manner similar to the network weights, and enable modification of the differential voltages (V^+, V^-) to compensate for device and parameter mismatches.

The series memristors have greater effect when V^+ (or V^-)is low; the parallel memristors have greater effect for higher V^+ (or V^-). These biasing memristors can be adjusted each time the weights are programmed. The bias values would most likely be found as part of a chip calibration procedure. This procedure would be done before setting the desired programming weights into the array, and uses a majority function for this purpose:

- Set the weights to create equal numbers of +1 and -1 values, and set all inputs high

- At each major clock cycle (10 ns in the simulation shown in Figure 4.17), cycle one +1 weight, and then one -1 weight by turning one input off, then on (each for 5 ns)

- After this, reduce both the total positive and negative weights by 1 (or any

other equal decrement)

- Repeat until the "common mode" weight is a minimum

- Based on the outputs, adjust the G_p and G_s devices:

- High common mode weights that create "0" errors require an increase in G_p^+

- Low common mode weights that create "1" errors require an increase in G_s^+

- G_p^- and G_s^- would be adjusted if the opposite conditions exist

This simple procedure assigns a value of 1 to each correct output, -1 to each incorrect output, and adjusts the comparator bias memristors until the total value equals the number of outputs measured (fully correct functionality). This procedure can be modified in many ways; for example, heavier emphasis can be given to getting correct values for high total conductance values and ignoring incorrect values at very low conductance values (or specific biases can be used to ensure extremely low conductance levels are never seen). Other optimizing algorithms can be used as desired. Using 45 nm design technology, LTspice simulations have shown that up to $\pm$ 10 mV (20 mV total) V_t mismatch and up to $\pm$ 5nm (10 nm total) dimensional

mismatch can be tolerated between the critical pairs of devices in this differential design (N1A and N1B in the input stage shown in Figure 4.16, and both sets of input FETs in the amplifier stage). An example LTspice simulation for this level of mismatch is shown in Figure 4.17.

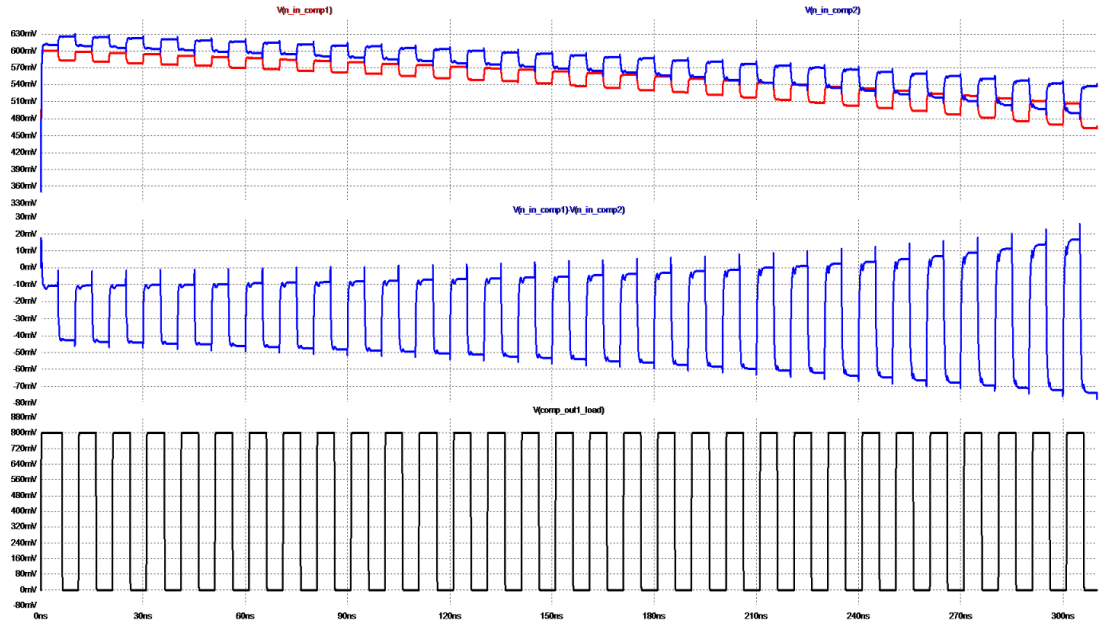

Figure 4.17: Effect of biasing procedure for handling mismatch. Top plot shows V^+ and V^-, the middle plot is the differential voltage ($\Delta V_{diode} = V^+ - V^-$); note that it is almost always negative. The bottom plot is the comparator output, using a 5 ns clock. The output signals are 100% correct and very clean.

4.5.3 Modifications to increase performance

This next design modification is used to improve the comparator performance (see Figure 4.18). Let us consider the basic operation. As the two currents I^+ and I^- flow into the diode, they raise the diode voltages (V^+ and V^-). The small difference in the currents caused by the differing weighted sums creates a small voltage difference; this differential voltage is amplified to create a much large voltage difference between V_{gate1} and V_{amp1} (V_{gate1} stays relatively constant, while V_{amp1} swings over a large range). The second amplifier is used to drive its output (V_{amp2}) nearly to V_{dd} or V_{ss}. The speed of this basic design is mainly dependent on the first amplifier, and is primarily determined by two factors:

1. How *fast* can the bias current swing V_{amp}?

2. How *much* does the bias current need to swing V_{amp}?

This can be expressed using the simple fact that V_{amp} has a node capacitance: therefore

$$I = C * \Delta V / \Delta t \tag{4.97}$$

or

$$\Delta t = C * \Delta V / I \tag{4.98}$$

To reduce Δt, you have to increase I or reduce ΔV. The first factor is essentially a design optimization: higher bias currents can swing the output faster, but take more power and create a larger design (increased capacitance) that will slow down the amplifier. Larger amplifier inputs will also slow down the rate at which the diodes

can swing the input voltages, but that is a smaller, secondary influence. The second factor is input dependent. In a situation where the previous weighted sum is highly negative and the current weighted sum = +1, the ΔV_{amp1} value is very high, but the final V_{amp1} voltage will be very close to V_{gate1}: the bias current needs a relatively long time to switch V_{amp1} past V_{gate1}; only then will the second amplifier switch as well. The second factor however, can be mostly managed by the architecture. Figure 4.18 depicts the changes to the amplifier architecture needed, and Figure 4.19 and Figure 4.20 are timing simulations that display the effect of these changes. By adding a transmission gate between V_{amp1} and V_{gate1}, we can force V_{amp1} to be very close to V_{gate1}; ΔV_{amp1} will be very small (and relatively constant under all input conditions). This is done by using a strobe signal that turns this T-gate on during the early part of the comparison operation, and turning it off during the later part. The diodes are always on. While this T-gate is on, a second T-gate (connected to V_{amp2}) is turned off, and the T-gate in the output driver is turned on, which keeps the previous output valid and avoids V_{out} glitching. The T-gates are turned off (or on, respectively) during the later portion of the comparison operation. The simulations in Figure 4.19 and Figure 4.20 are for the same array and inputs, with equal time scales. The comparator in the first plot does not have the T-gate/T_{strobe} feature; since the previous sum value is very positive ($\Delta V_{diode} \approx 6.0$ mV), $V_{amplifier1}$ must swing very far to cross V_{gate1} (from 900 mV to 400 mV). This takes a long time, making ΔT large (6.865 ns). In the second plot, the comparator with the T-gate/T_{strobe} included enables $V_{amplifier1}$ to drop very close to V_{gate1} almost instantly; this greatly reduces the delay ($\Delta T = 1.559$ ns).

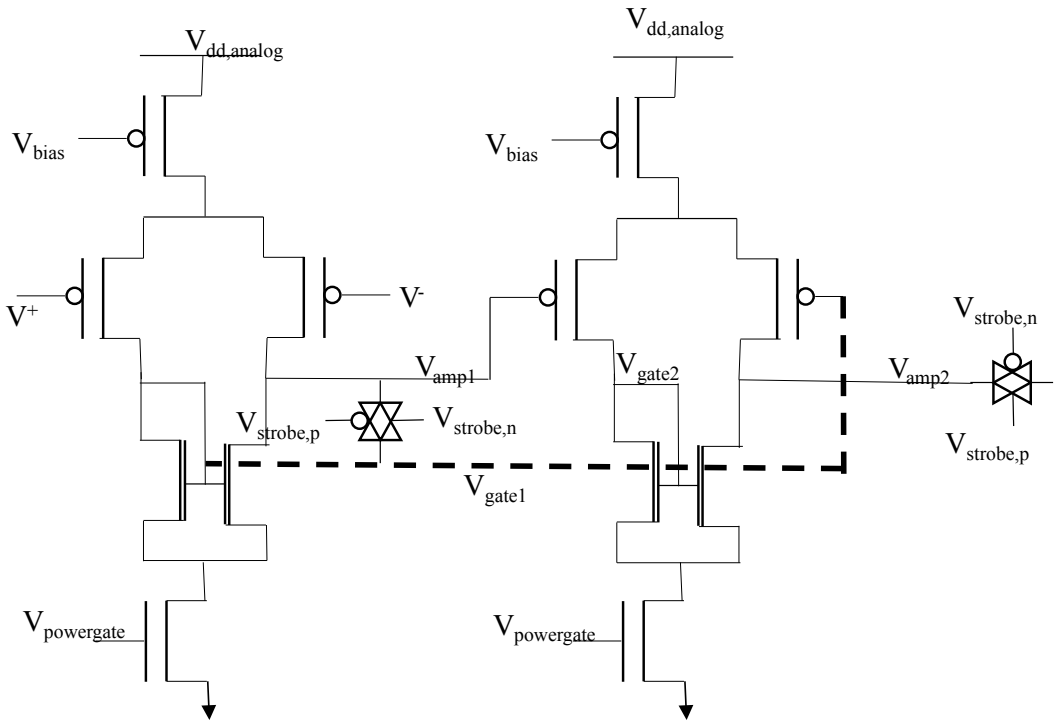

Figure 4.18: Amplifier stage of the comparator with modifications.

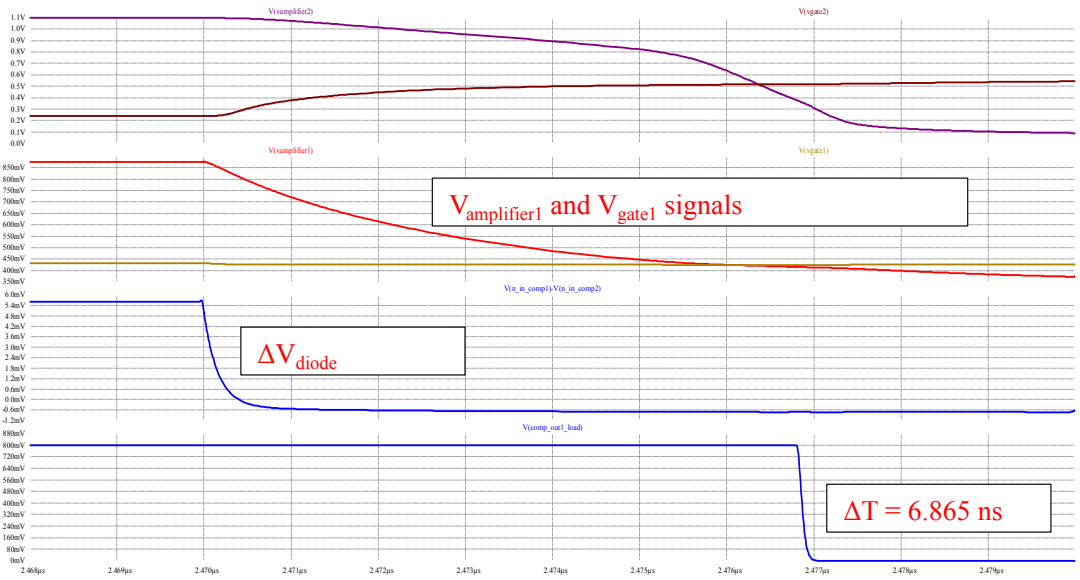

Figure 4.19: Timing plot for comparator with no T_{strobe}.

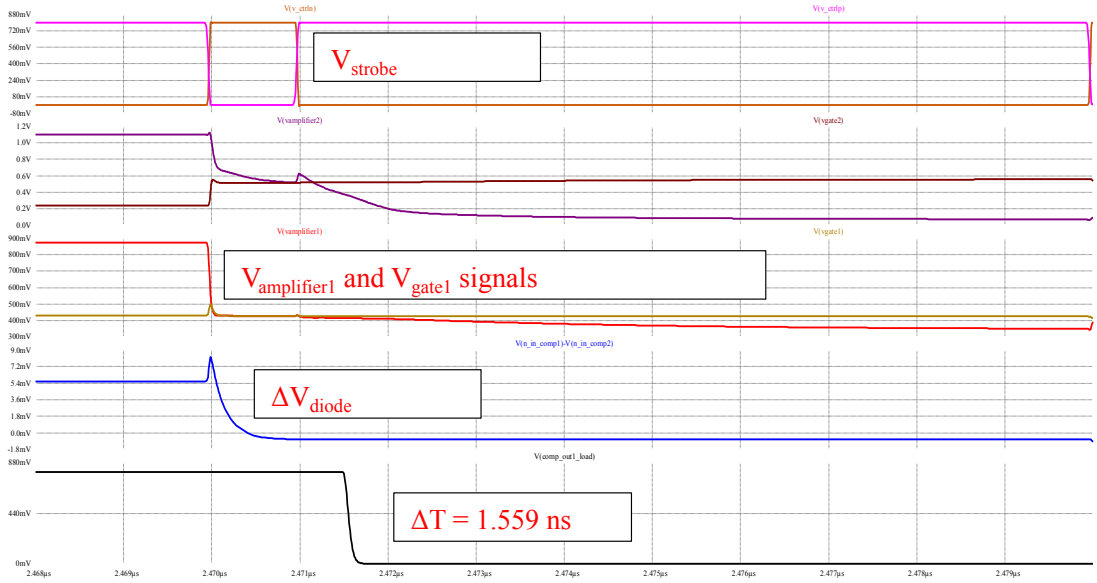

Figure 4.20: Timing plot for comparator with T_{strobe} = 1 ns.

Figure 4.21, Figure 4.22, and Figure 4.23 show how this affects the comparator speed. Without the T-gates, the speed is dependent on both the previous inputs and the current inputs. With the new architecture, V_{amp1} always starts very close to V_{gate1} and the worst case time delay is drastically reduced; the comparator speed is now relatively independent of the inputs. The T_{strobe} time increases the delay for very fast transitions, but these do not define the comparator speed. T_{strobe} = 1ns appears to provide the best balance.

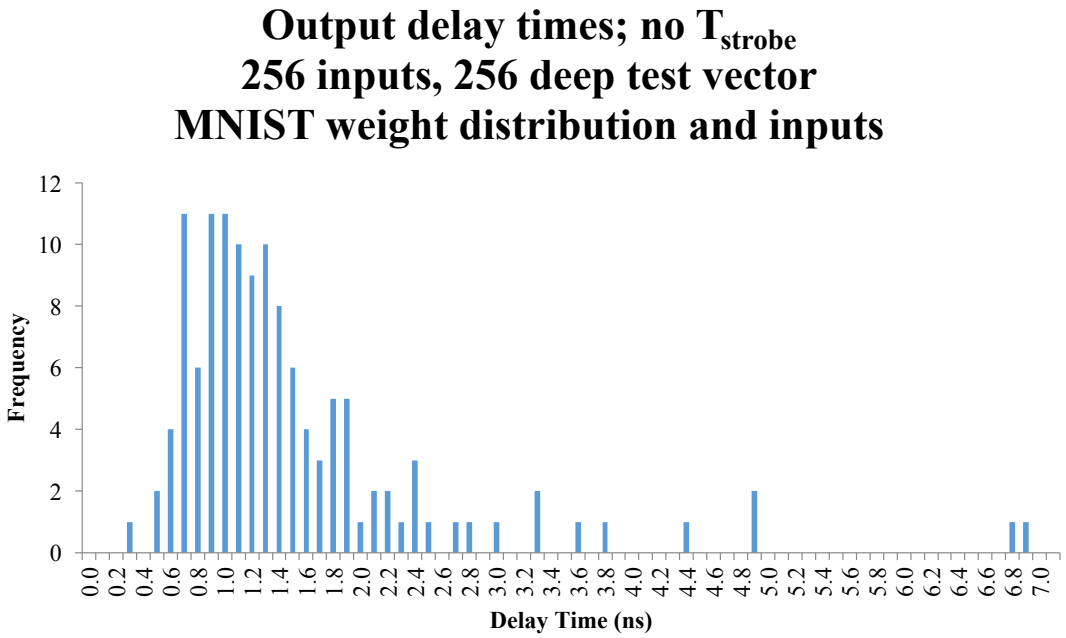

Figure 4.21: Output delay times for no T_{strobe}. The distribution is wide, and includes very long delays.

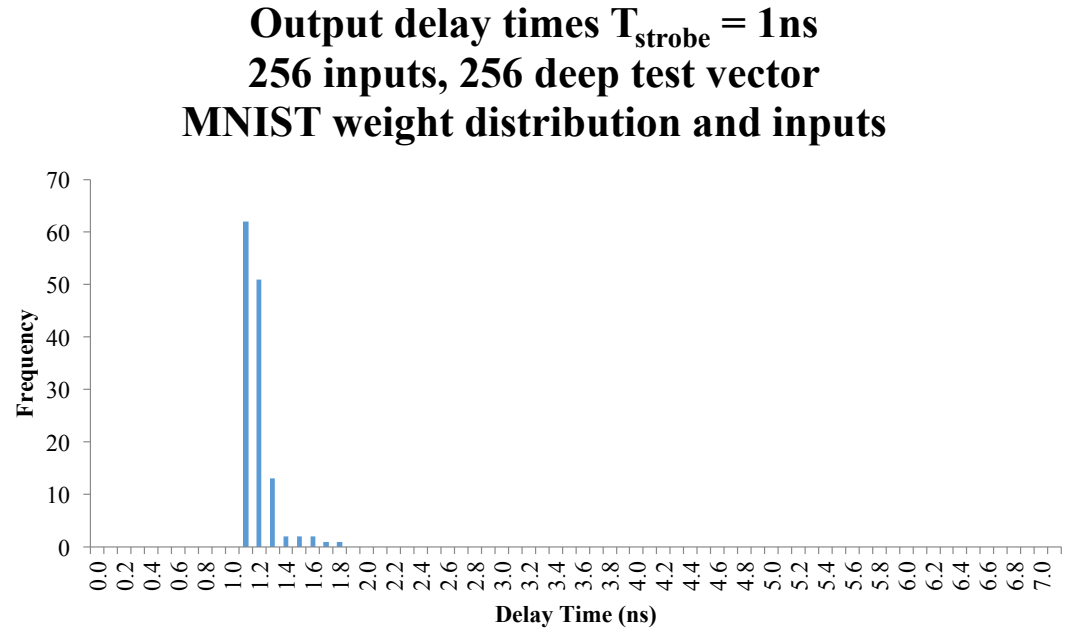

Figure 4.22: Output delay times for $T_{strobe} = 1$ ns. The distribution is tight, and T_{max} is below 2.0 ns.

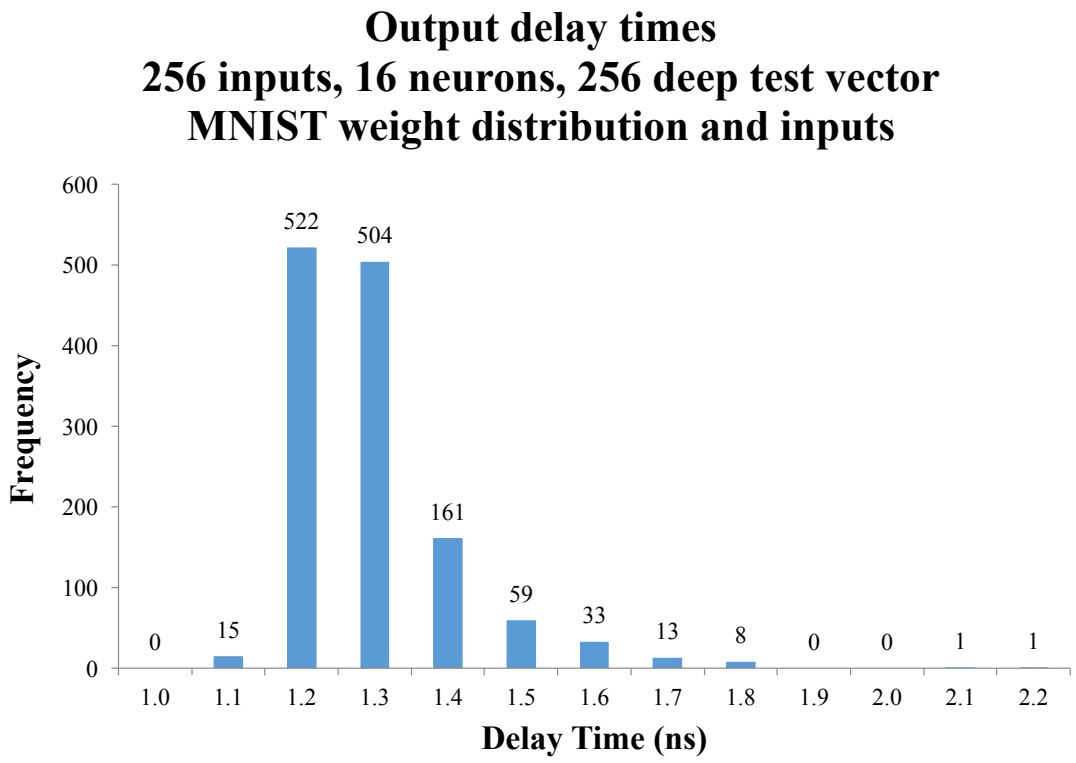

Figure 4.23: Distribution of output delay times for a 256 x 16 array; (the test vector is 256 deep). $T_{strobe} = 1$ ns. 99% of the delays are below 2.0 ns.

4.5.4 Practical operating range of the comparator

Another important consideration is the diode sizing: if they are too small, the total area actually increases due to the need for tolerating dimensional mismatches; if they are too large, the granularity of matching the diode to the programmed weights can be too coarse. Our design uses 4 sets of controllable diodes, and one diode is turned on and becomes active for every 60 μS of conductance programmed into the neuron (the maximum of the positive or negative weighting is used). This assumes nominal 0.8 and 1.1 V supplies for the digital and analog circuits, respectively (see Table 4.7).

Table 4.7: Optimal Conductance Range (G_{min}, G_{max}) for nominal supply voltages

#Diodes active	G_{min} (μS)	G_{max} (μS)
1	16	60
2	32	108
3	48	176
4	64	176
3 (1.0/1.2V)		260

This controllable input architecture enables the diode voltage V_{diode} (V^+ or V^- in Figure 4.16) to stay reasonably constant between 0.55 and 0.65 V, minimizing V_{mem} (and power in the array), while ensuring that ΔI_{diode} (and therefore ΔV_{diode}) when $\Delta G = 1$ is large enough to drive the amplifiers within the clock cycle (T_{clock} = 4 ns). Simulations show that $T_{clock} > 4$ ns for 0.1 mV $< \Delta V_{diode} <$ 0.25 mV; we use $\Delta V_{diode} = 0.5$ mV as a minimum for our analysis. As the total conductance (G_{max}) programmed into the neuron gets larger, more diodes will need to be active. This reduces the overall sensitivity; above a certain $G_{max} \approx 180$ μS it will become impractical to properly sense $\Delta G = 1$. By operating the comparator at elevated voltages (1.0 and 1.2V), we can increase this G_{max} by about 50%. However, the comparator will still continue to function with a higher G_{max} at nominal voltages if $\Delta G_{min} > 1$; the ratio of total conductance allowed is roughly linear with the ΔG_{min} needed (see Table 4.8). $G_{max} > 1000$ μS is possible if $\Delta G_{min} = 5$ μS.

Table 4.8: Maximum Conductance (G_{max}) when $\Delta G \geq 1$, 4 diodes active

ΔG_{min} needed (μS)	G_{max} (μS) for nominal voltages	G_{max} (μS) for elevated voltages
1	176	260
2	310	460
3	440	640
4	570	800
5	600	1020

The maximum number of diode pairs designed into the comparator will need to take into account the operating voltages, total possible array conductance, and the ΔG_{min} expected at these high conductances. For example, our analysis of a neural net using this design and running an MNIST application indicates the input neurons typically have low total conductances (G_{max} is below 100 μS ≈ 90% of the time) but occasionally require $\Delta G = 1$ μS (≈ 3% of the time), while the output neurons have much higher total conductances (between 240 and 390 μS) but rarely/never require $\Delta G < 5$ μS.

4.5.5 Power supply variation and temperature analysis

A simple test neuron, with 256 inputs and a weight distribution representative of an MNIST neural net, was used to examine the performance of the comparator over a range of supply voltages and temperatures. Table 4.9 and Table 4.10 show power and timing results for $V_{dd,digital}$ from 0.8 to 1.0 V, and $V_{dd,analog}$ from 1.0 to 1.2 V. Table 4.11 shows how operating temperature affects the power and speed of the neuron with this comparator; the supply voltages used for these simulations were $V_{dd,digital} = 0.80$ V, and $V_{dd,analog} = 1.00$ V (near worst-case). This results indicate that an operating frequency of 250 MHz for neuron evaluation in the cross-bar is practical, and 300 MHz is possible (only one T_{max} exceeded 3.333 ns in the simulation).

Table 4.9: Neuron power (in μW as a function of changing supply voltages)

$V_{dd,digital}$		0.8	0.9	1.0
$V_{dd,analog}$	1.0	43.7	59.2	81.4
	1.1	51.0	65.5	85.0
	1.2	62.3	75.9	94.8

Table 4.10: Neuron timing (worst case delay in ns as a function of changing supply voltages)

$V_{dd,digital}$		0.8	0.9	1.0
$V_{dd,analog}$	1.0	1.685	1.602	2.045
	1.1	1.555	1.375	1.408
	1.2	1.485	1.345	1.245

Table 4.11: Neuron power and timing as a function of temperature

Temperature (C)	Power (μW)	T_{max} (ns)
0	47.6	1.435
25	43.9	1.665
50	41.9	2.065
75	40.8	2.685
100	40.8	3.535

4.5.6 Final comparator results

A breakdown of the power components shows that the comparator consumes 15 μW, and the operating frequency is expected to be at least 250 MHz. These values are from extensive simulation of the comparator design presented. The area of the comparator is estimated at 55.1 μm^2, calculated using the following methodology:

- Calculate the gate area (W x L) of each transistor in the design
- Sum up the total gate area of the transistors and multiply this amount by 35X

The 35X factor was determined by reviewing actual SRAM (Static Random Access Memory) sizing as a function of technology nodes [70]. It was validated by looking at the physical size of layouts of comparators similar to the one described here [71], and from design information in [26]. This methodology will be used for estimating areas as part of the array mapping and evaluations later, when specific design area information is not available.

4.6 Tile concept as enabled by 1/High Z and comparator components

The use of the 1/High Z differential current architecture, and the comparator design described above, enables an important architectural option, which we call a tile. One of the major difficulties in trying to design a general purpose neural processor, based on the applications in this study, is that the desired array sizes

span a wide range. Just to use a few examples:

- The MNIST application maps well to 256 x 64 arrays
- The CSlite decoder stage naturally fits into an 8 x 256 array
- The CSlite detector stage has one layer in the network that requires 512 x 32 arrays; less than 512 inputs could not be mapped
- The AES-256 State Machine would prefer a 16 x 16 array mapping

Finding a single array size that can efficiently map *all* of these is a daunting task; the availability of tiles makes it more practical. The DZ architecture and our comparator allow for the use of control FETs to divert the differential current to specific diodes at the input stage of the amplifier. This means that we can add one set of two additional control FETs per comparator (per neuron) that enable the current to be passed to a comparator in a different array. Keep in mind that the current being passed represents the weighted sum of the inputs; therefore the function of the neural net is maintained. The second array is now evaluating its inputs *plus* the inputs from the first array; the two arrays are now combined into a single neuron (or set of neurons). Figure 4.24 shows the basic concept; this feature means that smaller arrays (tiles) can be connected to form much larger arrays. For example, four 64 x 16 tiles can be connected together to make a 128 x 32 array (see Figure 4.25). Here the inputs go into Tile 1 and Tile 3; they are also sent across to Tile 2 and Tile 4. The comparators in Tile 1 and Tile 2 are shut down, and the current (the sum of the weighted inputs) is passed to the comparators in Tile 3 and Tile 4, which now

sum up all of the weighted inputs to create the final outputs.

The design optimization is now to find the optimum tile size, not the optimum array size. This new architectural option also greatly expands the set of possible solutions. Without this, no array smaller than 512 inputs could have been used for the general purpose neural processor design; now tiles that are very small (8 x 2 or 16 x 1, for example) are possible solutions.

The tile concept is further enhanced by the ability to control the current (and therefore the power) in the unused portions of the tile (unit cells, comparators). Simulations of a 256 x 32 array show that the active power can be completely eliminated; the leakage power is an extremely small fraction (much less than 1%) of the total. There are drawbacks to using the tiles; the input circuits may need to send the input value across a tile to a neighboring tile, through another driver/latch circuit. This adds a small delay ($\approx$ 30 ps per tile); as long as the number of horizontal tiles connected is reasonable (10 or less), the effect on performance is small. Another point to be made is that the control PFETs added to the comparator design need to pass a large amount of current, and are therefore large (W/L = 1200/45 nm); this keeps the ΔV_{ds} below 3 mV. This adds about 3.8 μm^2 in area to the comparator. A more important issue is that all the tiles need to have comparators, which are relatively large; this is because a tile needs to be an array itself, not just part of a larger array. This issue will be explored more completely in the array mapping and evaluation chapters.

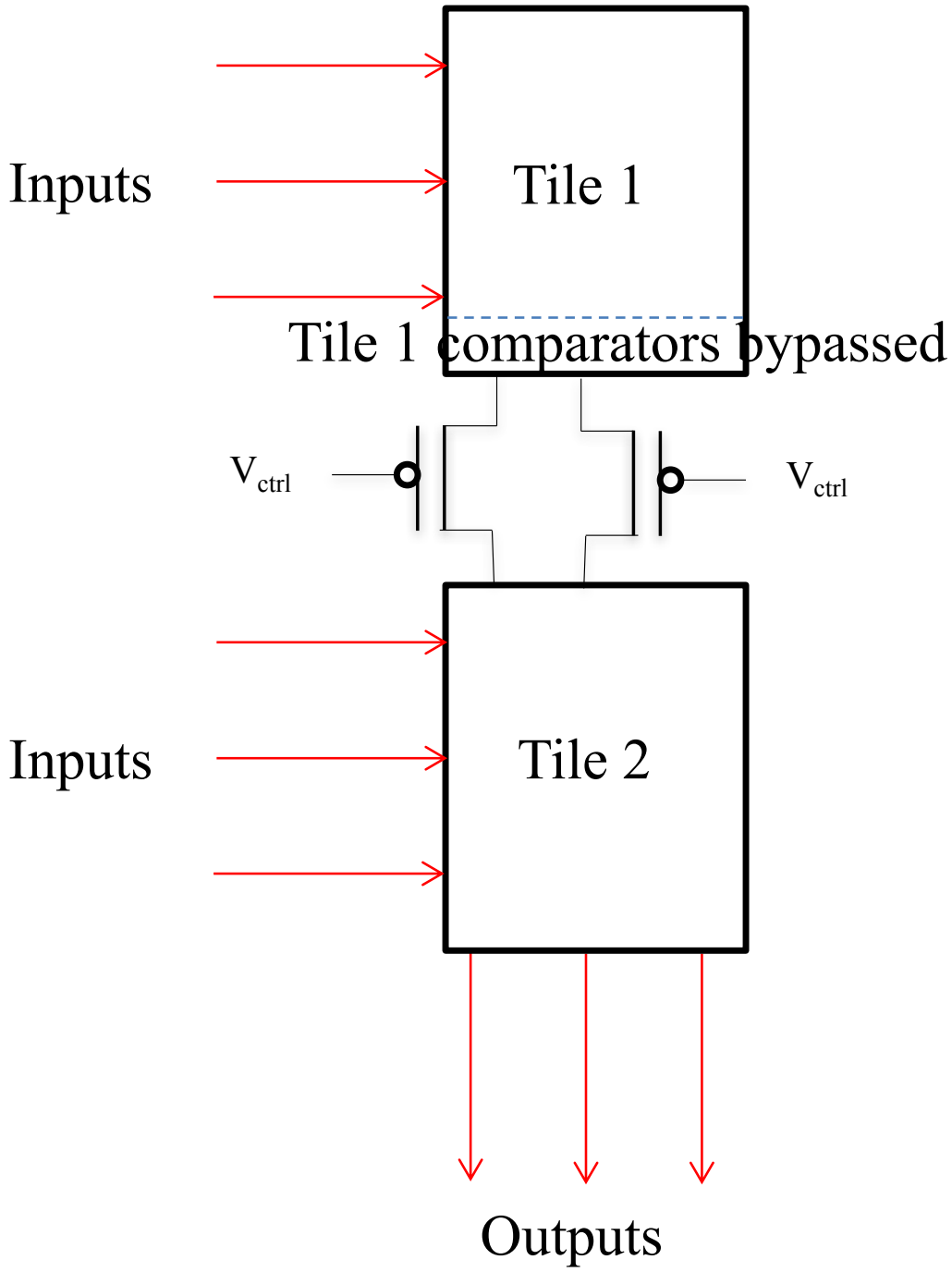

Figure 4.24: Use of control FETs that enable two smaller arrays (tiles) to be combined into a single array.

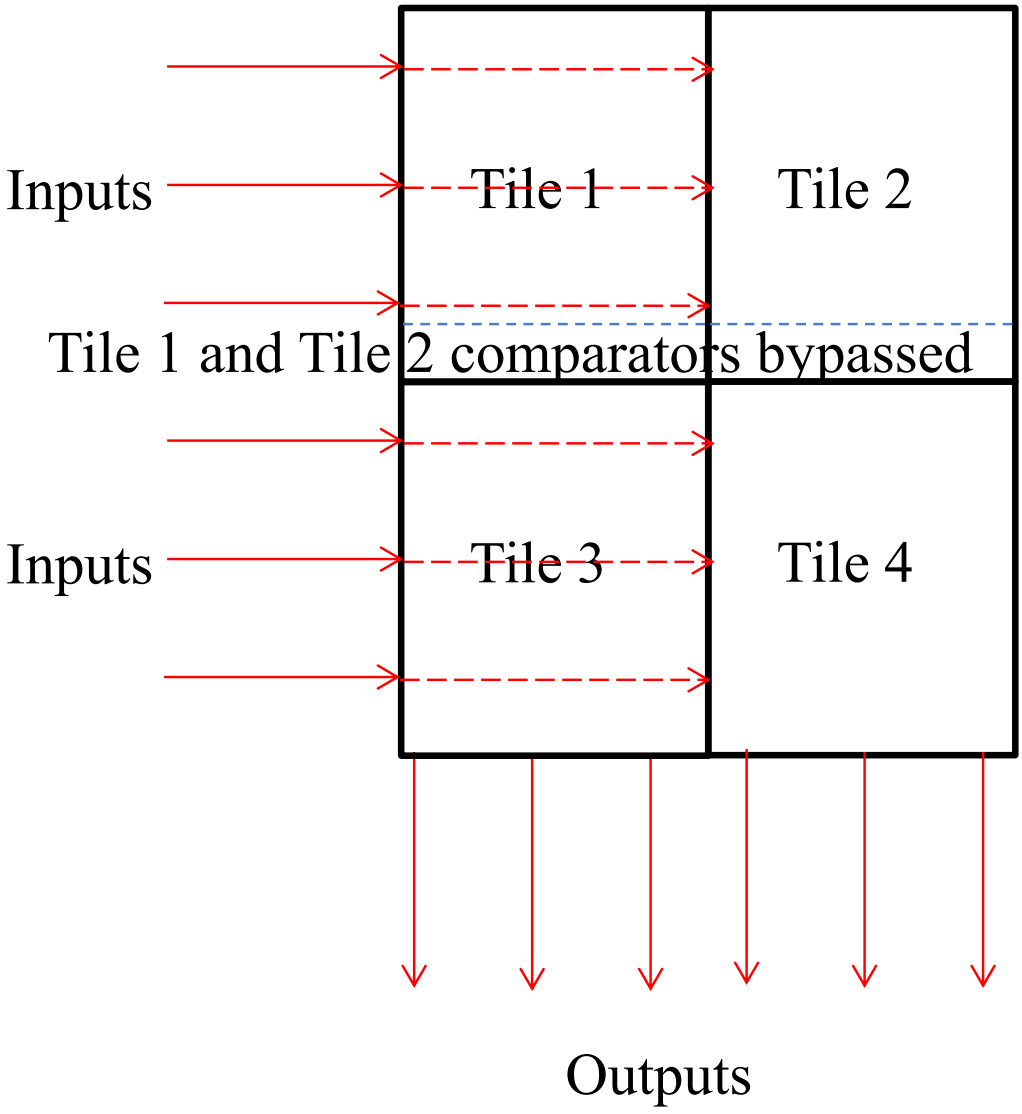

Figure 4.25: Four 64 x 16 tiles combined to form a 128 x 32 array.

4.7 Communication network

The network-on-chip (NOC) is used to send neuron outputs from one array to another; this assumes that direct connections are impractical. As will be seen, the NOC capability will have a noticeable impact on the overall performance of the neural network [72]. This is an area where the brain has better "technology"; each neuron is directly connected to about 10,000 other neurons [73]. For neural networks, researchers must rely on the far greater speed of electronics (MHz/GHz compared to kHZ). There is one additional factor to be considered and leveraged in designing the NOC, sparsity of communication. As has been shown [73, 74], the output of a neuron changes very infrequently, so multiple neurons can be connected together in a bus structure; [74] connects 64 neurons together with only a very small loss in accuracy. For our analysis, we will assume 16 neurons can be connected together in a bus without causing collisions. Adding a small FIFO (First In First Out) or other mechanism would be a possible approach to managing collisions if it proved necessary. Leveraging the sparsity of communication inherent in neural networks greatly reduces the NOC capability required. Since the output of a neuron for a TGN is only a single bit, and data is only sent when the output changes, all that needs to be sent over the NOC is the address. We will assume that 20 bits will be sufficient for this purpose.

4.7.1 2D mesh

The first NOC to be considered is a 2D mesh. This is a common approach for neural nets [26, 58, 73, 75]. Figure 4.26 is a generic example. There is a router cell located at the intersection of each row and column of arrays; the router typically can send data in 5 directions: North, South, West, East, and into the array. It can also recive data from those 5 directions. The basic architecture of the router is very simple (see Figure 4.27), so it is compact, energy efficient, and fast. A router of this design was implemented in an FPGA, with a neural net running MNIST, to characterize its capabilities. Data was also used from [26, 58, 73, 75] to elucidate additional aspects of building a NOC with this approach. The important characteristics of this router are:

- The area is 4000 μm^2 in 45 nm technology
- The speed is 1 GHz, or 1 ns latency per hop
- The number of hops on average has been estimated previously as 5 for a 256 x 256 array [73]; this will be analyzed more carefully later
- Power is dominated by wire capacitance, at 20 nW/wire/Ghz/μm of length: or 400 nW per μm for our design

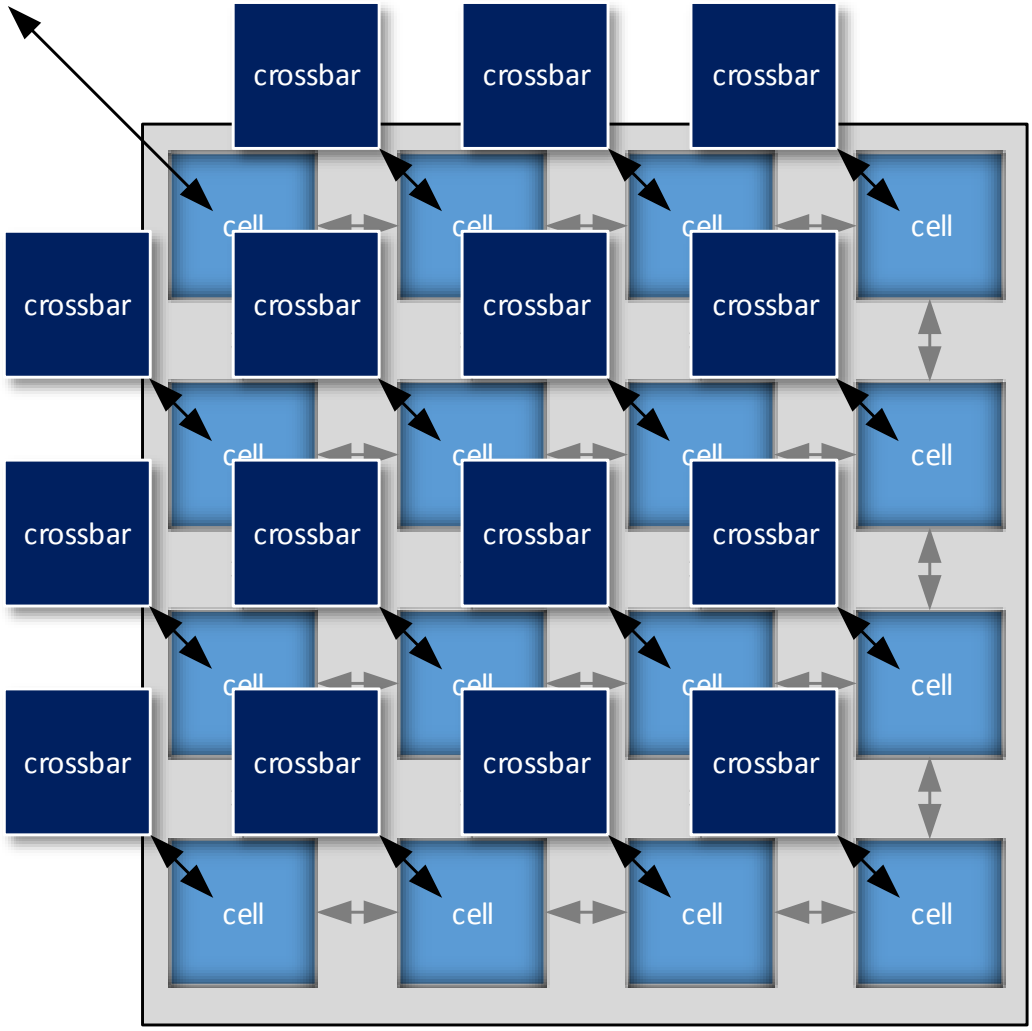

Figure 4.26: NOC using a 2D mesh design; the cell implements the 5-way routing function.

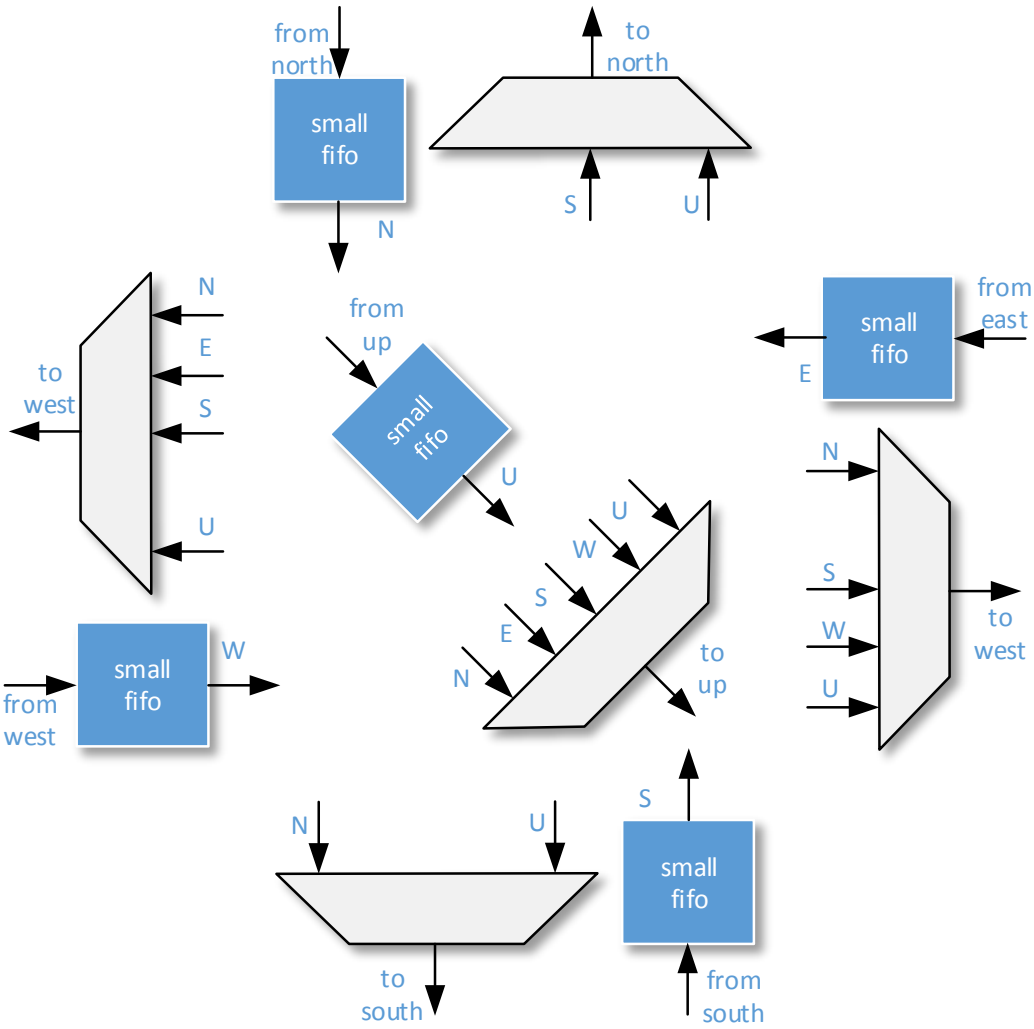

Figure 4.27: Details of the router cell.

4.7.2 All-to-all switch

The second NOC that will be analyzed for use is an all-to-all switch (A2A), configured into a hierarchical tree; it is somewhat similar to the SpiNNaker NOC design [76]. The A2A is based on a switch designed and used in a high performance computing ASIC [77]. That switch was large (96 ports with 72 bit packets) and very fast given its size and technology node (500 MHz, 90 nm). The power, area, and timing of the A2A in our design will be scaled appropriately. Our A2A switches will be connected hierarchically in a fat tree structure (see Figure 4.28). At every level, each switch will require 25 bi-directional ports (input + output):

- 16 ports to connect to the bus/A2A from the next level down
- 8 ports to connect to other switches at the same level
- 1 port to connect to the switch one level up

This means the first level switch connects to 256 neurons directly (16 x 16; one hop), and 2304 neurons (256 x 9) are accessible within two hops. If longer range communication is required, only two more hops (four total) increases the reach to 36,684 neurons, and up to 589,824 neurons are within 6 hops of each other. While the A2A switch is significantly larger and more power hungry than a single router in the 2D mesh, there are fewer needed (one A2A is amortized over many arrays vs. one router per array for the 2D mesh architecture) and they provide much better performance for long range communication. The key characteristics are:

- The area is 43,164 μm^2 in 45 nm technology; 10X the size of the router in the 2D mesh
- The speed is 1 GHz, or 1 ns latency per hop
- Power is 250.8 μW for the 25-port design running at 1 GHz; this is roughly 6-7X the 2D mesh router power, assuming a 100 μm wire length

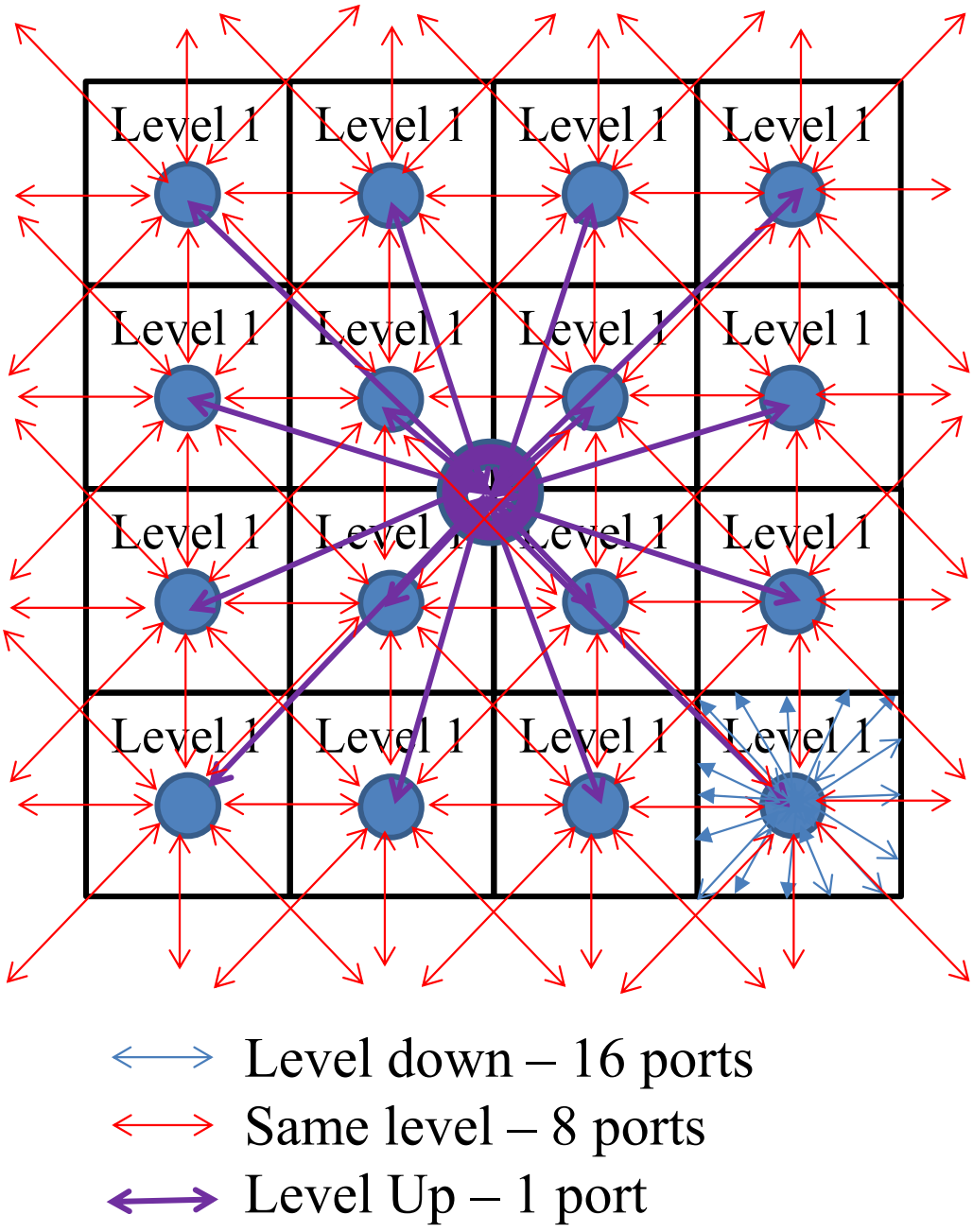

Figure 4.28: Notional tree architecture for the A2A switch.

4.8 Programming Circuits

High precision DAC and ADC circuits [58] will be used for programming the memristors, since precision conductance values will be needed in order to enable large arrays. Static or variable pulse height and widths are both possible, with static pulse heights easier to implement. While our design exploration assumes off-chip learning, the nominal requirements for the programming circuits have been defined. The power requirements of the programming circuits will not impact the overall analysis, since programming is not part of the inference operation; the area is assumed to be negligible, since one set of programming circuits can be used for a large number of arrays.

4.8.1 DAC and ADC

This analysis assumes a programming sequence similar to the one in [29] is used. Here a DAC is used to provide a precise voltage on the access PFET gate in the unit cell, and the current is read using an ADC. A feedback loop is used to adjust the FET gate voltage until the desired current is read in the ADC. This approach can be used to program the conductance to within 1% of the deisred value. Figure 4.29 shows how this can be used to program the G_{eff} of a memristor to a range of integer weight values. In order to achieve the 1% precision desired, both the DAC and ADC need to have 9-bit accuracy.

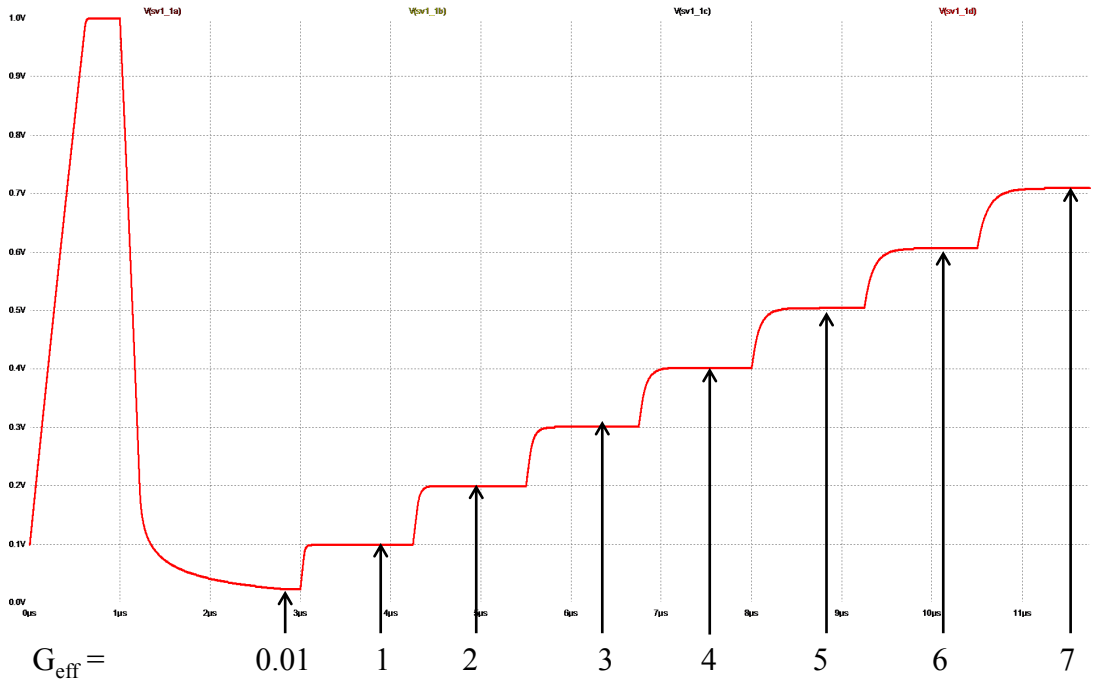

Figure 4.29: Example of precision programming of a memristor.

4.8.2 Programming time

Using data from [78], we can estimate programming times for an entire chip. With precision voltage inputs, we can program in 5 microseconds (50 pulses at 100 ns), and with precision timing inputs we can program in 2 ms (2000 pulses at 10 microseconds).

We will assume the following:

256 x 128 array = 64k memristors per array;

64 arrays per core = 4M memristors per core;

256 cores per chip = 16k arrays per chip.

This gives us 1 billion (1×10^9) memristors to program.

In complete serial fashion, this takes

$(1 \times 10^9) \times (5 \times 10^{-6}) = 5 \times 10^3$ seconds (1.4 hours) with precision voltages.

$(1 \times 10^9) \times (2 \times 10^{-3}) = 2 \times 10^6$ seconds (555 hours) with precision timing.

The authors note that a 1T1M design enables a whole row to be programmed in parallel, which will reduce these times by 256x; this is one reason a 1T1M design is useful.

$5 \times 10^3 / 256 = 20$ seconds

$2 \times 10^6 / 256 = 7800$ seconds = 2 hours

Another option might be to program an array serially, but program all the arrays on the chip in parallel, which will reduce the serial time by 16,384x

5 x 10^3/16384 = 0.3 seconds

2 x 10^6/16384 = 120 seconds

Combining these two options (1 row in each array in parallel, all arrays in parallel) reduces the time by

256 x 16,384 = 4 x 10^6 time reduction

5 x 10^3/4 x 10^6 = 1 ms

2 x 10^6/4 x 10^6 = 0.5 seconds

So precision voltage programming with a 1T1M design (1 row programmed in parallel) is clearly practical. Precision timing programming will need to utilize some additional parallelism. Our design assume a precision voltage approach is used.

4.9 Simulation info

With our design, we can use a standard workstation to simulate a single MNIST-relevant neuron (256 inputs, 1 neuron) using a test vector with a depth of 256 in about 15 minutes, which generates 1-2 GB of data. This allows us to carefully examine many aspects of the design in great detail (such as the factors affecting the comparator speed). This simulation condition was used extensively to provide many of the results above. An array that is close to MNIST-sized (256

inputs, 32 neurons) takes about 3 days to run, and generates about 115 GB of data. This can be done occasionally to investigate certain aspects of large array designs (such as the power gating analysis that verified the tile concept validity). These simulations are only possible because the memristor model we used is computationally efficient; more complex models [63] would limit practical simulations to arrays with perhaps 8 neurons. An MNIST-sized array (256 inputs, 64 neurons) using the model shown in Figure 4.12 requires 3 weeks to simulate and would generate 1 TB of data; it would be essentially impractical to simulate this size array on a single workstation. These larger arrays require XYCE [79], a parallel SPICE simulator to run, or need to be simulated in a partitioned fashion. These simulation results will be published as part of a comparison of a wide variety of neural net implementations of MNIST [80].

Chapter 5: Mapping the neural network onto potential architectures

In this chapter we will describe the methodology and tools used to map the three applications (MNIST, CSlite, and AES-256) onto a variety of tile sizes, in order to determine the best possible tile size for a general purpose neural processor. We we will also provide the rationale for using application area as the primary criteria to reduce a large number of possible alternatives down to the most promising few.

5.1 Integrated simulation environment

To facilitate co-design of applications, architectures and technologies for neural processors, it would be highly desirable to have a workflow to:

- Create and train a neural network for a specific application;
- Map this neural network onto an architecture with defined attributes such as array size, weight precision and range, and allowed connectivity;
- Verify application accuracy for the NN architecture chosen;
- Perform detailed functional and circuit simulations to validate the application and characterize important metrics such as power, performance, and area.

This would enable faster co-design iterations, and would lead to trade-offs that optimize important application and system level attributes. There has been progress in developing such tools; a workflow and toolchain for mapping spiking neural nets (SNN) to a specific architecture that minimizes communication costs is described in [81].

Loom [24] is a set of tools to facilitate the implementation of algorithms as Threshold Gate Networks. Neural nets can be treated as TGNs. The implementation process typically starts by breaking the application into steps or distinct subfunctions. These pieces are then implemented in turn. Finally, the partial TGNs are combined into a whole TGN and simulated. Loom has tools for each of these steps. Loom provides two methods for generating the weights and network connectivity needed to define a TGN. The first method trains a pre-defined network; the designer creates the network topology, including nodes and interconnectivity. It supports completely connected networks in which inputs connect to nodes in the hidden and output layers, and it can assign weights drawn from a restricted set of values and with limited precision. This makes it a very effective tool for creating and analyzing memristor based crossbar architectures.

The second method builds up a network using a linear separation algorithm [82]. This technique starts with a single neuron and then adds additional neurons to further divide the input space until all outputs are correct. The weights for each neuron are found using linear programming. The advantage of this method is that the TGN size and connectivity do not have to be provided a priori. This capability is presently under development.

Loom also contains tools for merging interconnected TGNs into a single TGN and for discarding redundant or unconnected neurons. TGNs can also be created out of designs from a library of commonly used circuits. Loom combines all the TGNs while respecting data and ordering dependencies, including recurrent feedback.

Loom can also simulate the behavior of a TGN. It converts the TGN into a weight matrix and state vector and performs the multiply-accumulate-threshold operations layer by layer. Loom can execute the simulations using multicore CPUs or GPUs. Simulations allow those porting an algorithm to a TGN to test and debug the implementation with complete visibility into the network state. This approach also helps attribute errors to either the TGN itself or to a particular physical implementation.

One current limitation of Loom is that it assumes the smallest array is the best; based on the neural network created, it will define the smallest array potentially possible, and then determine the minimum number of arrays of that size that the neural network can fit into. It will then gradually increase the number of arrays until it reaches a user defined maximum. If the network has not been mapped correctly, it will increase the array size and re-start the process. Loom was not developed with the tile concept in mind, which enables arrays *smaller* than the theoretical minimum size to be used. This capability can be incorporated by including an additional loop in the mapping that starts with a (user-defined) tile size instead of an assumed minimum; this loop can iterate over many tile sizes if desired, using estimated area as an optimization goal. Once a set of possible tile sizes has been generated, the smallest area alternative would be selected.

Currently, the methodology is to use Loom to map the applications onto a variety

of arrays with different aspect ratios (ratio of inputs to outputs) and sizes. Each of these array sizes will require a certain number of arrays of that size to fully implement the entire neural network for the application. Since Loom can partition and decompose the neural network into its functional pieces, different sized arrays will be optimal for the various pieces. The next step is to determine the minimum number of tiles of various sizes that would be needed to create both the pieces and the whole neural network; for the general purpose neural network, we are assuming a single tile size will be used. For each tile size explored, the total area for the neural network can be estimated. The most suitable tiles will be identified by ranking them according to area.

Let us use the CSlite application to work through the process. The first step is to use Loom to map each of the four modules (ByteDecoder, Signature, SetHold, and Detector) to a wide variety of array sizes, and then identify the subset that are the best fit. Best fit is defined by looking at the synapse utilization; the higher the utilization, the better the fit. Table 5.1 and Table 5.2 show how ByteDecoder maps to 8x16 and 8x256 arrays; the utilization is 100% for each, but 8x16 requires 16 arrays and 8x256 only 1 array (for a single byte: there will need to be 6 total, one for each byte). Therefore 8x256 is selected as the better fit, and that array size will be used for the special purpose design to be discussed later. Now we map the potential tile sizes to all the possible arrays; Table 5.3 and Table 5.4 show this mapping to the previous two arrays using a 64x16 tile. For either array size, you will need 16 tiles (the actual total will be 6 x 16 = 96); looking at all possible tile-to-array mappings identifies this as the minimum. This tells us we need 96 64x16 tiles to implement

the ByteDecoder neural net for CSlite.

Table 5.1: Mapping ByteDecoder to 8x16 arrays

Array #	Inputs	Outputs	Utilization
0	8	16	1.000
1	8	16	1.000
2	8	16	1.000
3	8	16	1.000
4	8	16	1.000
5	8	16	1.000
6	8	16	1.000
7	8	16	1.000
8	8	16	1.000
9	8	16	1.000
10	8	16	1.000
11	8	16	1.000
12	8	16	1.000
13	8	16	1.000
14	8	16	1.000
15	8	16	1.000

Table 5.2: Mapping ByteDecoder to 8x256 arrays

Array #	Inputs	Outputs	Utilization
0	8	256	1.000

Table 5.3: Mapping 64x16 tiles to 8x16 arrays for ByteDecoder

Array #	Inputs	Outputs	Utilization	Vertical	Horizontal	Total Tiles
0	8	16	1.000	1	1	1
1	8	16	1.000	1	1	1
2	8	16	1.000	1	1	1
3	8	16	1.000	1	1	1
4	8	16	1.000	1	1	1
5	8	16	1.000	1	1	1
6	8	16	1.000	1	1	1
7	8	16	1.000	1	1	1
8	8	16	1.000	1	1	1
9	8	16	1.000	1	1	1
10	8	16	1.000	1	1	1
11	8	16	1.000	1	1	1
12	8	16	1.000	1	1	1
13	8	16	1.000	1	1	1
14	8	16	1.000	1	1	1
15	8	16	1.000	1	1	1
Total Tiles						16

Table 5.4: Mapping 64x16 tiles to 8x256 arrays for ByteDecoder

Array #	Inputs	Outputs	Utilization	Vertical	Horizontal	Total Tiles
0	8	256	1.000	1	16	16
Total Tiles						16

We then repeat this process for the other modules; the results of this are in Table 5.5 and Table 5.6. Looking at all the mapping data, we can also identify the single array size (not tile size) that would be most suitable for CSlite, which is 512x32. We will use this array size for what we call a limited purpose design (LPD). While the special purpose design (SPD) will use a variety of array sizes, each selected specifically for one module in the application, the LPD will be a single array size selected for the entire application. This will give us a mechanism for quantifying the value of the tile concept; if tiles were not possible, the LPD array size would be the best fit to the application. It is straightforward to get a sense of the mapping efficiency by looking at the total number of synapses needed (inputs x outputs). Comparing the total needed for the best array to that for a given tile, the 64x16 tile size is a poor fit for the ByteDecoder, OK for the SetHold, and excellent for the Signature and Detector modules. In fact, the 64x16 tile is 10% more efficient than the "Best Array" for the Detector module! How can this be? This is a direct result of the tile concept; without the tile feature, the minimum size array is 512x32; this size is a good fit for most of the Detector neural net, but a very poor fit for two of the arrays (as shown in Table 5.7). The 64x16 tile maps perfectly into the 512x32 arrays, and then nearly perfectly into the two smaller arrays also needed.

Table 5.5: Finding Best Array Sizes for the CSlite SPD

Module	Best Array	#BestArrays
ByteDecoder	8x256	6
Signature	64x16	140
SetHold	33x16	125
Detector	512x32	12

Table 5.6: Mapping 64x16 tiles and 512x32 arrays to CSlite

Module	64x16 tiles	64x16 efficiency	512x32 arrays	512x32 efficiency
ByteDecoder	96	0.078	48	0.016
Signature	140	1.000	63	0.139
SetHold	125	0.516	63	0.064
Detector	174	1.100	12	1.000

Table 5.7: Mapping 64x16 tiles to 512x32 arrays for the Detector

Array #	Inputs	Outputs	Utilization	Vertical	Horizontal	Total Tiles
0	507	32	0.990	8	2	16
1	507	31	0.959	8	2	16
2	507	26	0.805	8	2	16
3	507	26	0.805	8	2	16
4	507	26	0.805	8	2	16
5	507	26	0.805	8	2	16
6	507	26	0.805	8	2	16
7	508	26	0.806	8	2	16
8	507	26	0.805	8	2	16
9	507	24	0.743	8	2	16
10	196	32	0.383	4	2	8
11	189	20	0.231	3	2	6
Total Tiles						174

Table 5.8 and Table 5.9 provide an example of the complete output of this methodology for the CSlite application and every tile size considered. These tables rank tiles by the area required to implement the CSlite application, assuming no communication network is needed; outputs are assumed to be directly connected to the appropriate inputs. The tile sizes are ordered so that the most efficient area mappings are in the first table, and the least efficient are in the second table. Ratio refers to how much larger the calculated area is compared to the minimum; there are a reasonable number of tile sizes that are fairly close to the minimum.

Table 5.8: Ranking tiles for CSlite mapping using area (mm^2); first table

Inputs	Neurons	Byte Decoder	Signature	SetHold	Detector	CSlite	Ratio
64	16	0.117	0.171	0.153	0.213	0.655	1.00
32	8	0.093	0.144	0.121	0.324	0.681	1.04
32	16	0.078	0.226	0.101	0.280	0.686	1.05
64	32	0.103	0.274	0.135	0.186	0.698	1.07
128	32	0.156	0.208	0.204	0.143	0.710	1.08
64	8	0.147	0.191	0.191	0.258	0.788	1.20
32	4	0.123	0.160	0.160	0.391	0.834	1.27
128	16	0.185	0.241	0.241	0.169	0.836	1.28
128	64	0.141	0.370	0.188	0.182	0.882	1.35
16	8	0.071	0.221	0.093	0.497	0.883	1.35
32	32	0.070	0.375	0.275	0.253	0.974	1.49
256	64	0.221	0.294	0.294	0.166	0.975	1.49
16	4	0.087	0.226	0.113	0.551	0.976	1.49
256	32	0.250	0.328	0.328	0.114	1.020	1.56
64	64	0.095	0.500	0.250	0.226	1.072	1.64
64	4	0.206	0.268	0.268	0.344	1.087	1.66
128	8	0.243	0.317	0.317	0.229	1.106	1.69
16	16	0.064	0.371	0.249	0.458	1.142	1.74
32	2	0.183	0.239	0.239	0.557	1.217	1.86
256	128	0.206	0.550	0.275	0.223	1.255	1.92
256	16	0.308	0.400	0.400	0.160	1.269	1.94
16	2	0.118	0.306	0.153	0.712	1.288	1.97
8	4	0.068	0.267	0.089	0.867	1.291	1.97
512	128	0.328	0.437	0.437	0.191	1.393	2.13
128	128	0.134	0.635	0.357	0.279	1.405	2.15
8	2	0.084	0.219	0.110	1.017	1.430	2.18
8	8	0.061	0.375	0.158	0.842	1.435	2.19
512	64	0.357	0.475	0.475	0.149	1.456	2.22

Table 5.9: Ranking tiles for CSlite mapping using area (mm^2); second table

Inputs	Neurons	Byte Decoder	Signature	SetHold	Detector	CSlite	Ratio
32	64	0.067	0.690	0.435	0.316	1.508	2.30
16	32	0.060	0.638	0.391	0.431	1.519	2.32
512	32	0.414	0.543	0.543	0.104	1.605	2.45
256	8	0.423	0.551	0.551	0.187	1.713	2.62
16	1	0.180	0.234	0.234	1.085	1.733	2.65
8	1	0.116	0.151	0.151	1.397	1.815	2.77
8	16	0.057	0.659	0.295	0.813	1.824	2.79
512	256	0.313	0.836	0.418	0.366	1.932	2.95
256	256	0.199	0.796	0.531	0.431	1.957	2.99
512	16	0.529	0.689	0.689	0.132	2.039	3.12
1024	128	0.571	0.761	0.761	0.190	2.283	3.49
1024	256	0.542	0.723	0.723	0.361	2.349	3.59
1024	64	0.628	0.837	0.837	0.157	2.460	3.76
8	32	0.055	1.165	0.570	0.785	2.574	3.93
1024	32	0.743	0.975	0.975	0.170	2.862	4.37
1024	512	0.528	1.407	0.704	0.704	3.342	5.10
512	512	0.306	1.633	0.816	0.714	3.470	5.30
1024	16	0.972	1.266	1.266	0.213	3.716	5.68
2048	128	1.056	1.409	1.409	0.264	4.138	6.32
2048	256	0.999	1.332	1.332	0.500	4.163	6.36
8	64	0.054	2.129	1.050	0.994	4.226	6.45
2048	64	1.171	1.561	1.561	0.293	4.586	7.00
2048	512	0.971	2.588	1.294	0.324	5.176	7.91
2048	32	1.400	1.837	1.837	0.321	5.395	8.24
1024	1024	0.694	2.776	1.388	1.388	6.246	9.54
2048	16	1.857	2.419	2.419	0.406	7.101	10.85
8	128	0.053	3.460	2.009	1.624	7.146	10.91
8	256	0.053	5.272	3.939	3.232	12.496	19.09

5.2 Methodology for estimating area

The area calculations for the tiles (except for the unit cell, which has a layout), use the nominal FET design dimensions (W x L) and scale this area by 35X; this scaling factor is based on SRAM cell sizes as a function of technology nodes [70]. We will use a 256x64 array to demonstration our methodology.

The first component block is the input: Address Decoder, Multiplexer, DriverLatch, and Row Driver. The Address Decoder is scaled to match the number of inputs, and the Row Driver is scaled to match the number of neurons; the Multiplexer and DriverLatch areas are independent of the tile size. For a 256x64 tile, the input block is estimated to need 1689 μm^2. The next component block is the array itself; this is simply Inputs x Outputs x 2 x Unit Cell area; the estimate for a 256x64 tile is 3981 μm^2. The final tile component is the comparator; the only scaling done here is based on the number of inputs. Since small tiles are unlikely to need the full conductance range that 4 diodes can provide, the number of diodes is scaled based on the tile input size:

- $\leq$ 32 inputs, use 1 diode
- 64 inputs, use 2 diodes
- 128 inputs, use 3 diodes
- $\geq$ 256 inputs, use 4 diodes

For our example, we require 4 diodes, so the comparator area for the tile is (64 x

55.1 = 3527 μm^2). The total area needed can be found by multiplying this total (9198 μm^2) by the number of tiles required. For direct connections, there will not be any area associated with moving addresses (register and NOC components). The complete general purpose processor analysis will need to include the areas for these functions. The address register (20 bits) is small (6.9 μm^2 per neuron); the NOC components are large (4000 μm^2 and 43,164 μm^2, respectively for the 2D mesh and A2A switch). The NOC area will need to be amortized over the appropriate number of tiles, which depends on the number of neurons in each tile.

If a $4F^2$ unit cell is used (0T1M), and the other circuits are placed underneath, the area is reduced by <2X to 5217 μm^2; this indicates the cost of the 1T1M unit cell is reasonably small, given its clear benefits.

5.3 Use of area as a ranking mechanism

Total area can be used as the primary evaluation mechanism at this stage because the other two major variables (power, timing) are dependent only on how much of the tile is used, not the size of the tile itself. This can be illustrated with a simple example; use the MNIST application, and assume a single tile size of 256 x 64. From simulations and the area estimations just described, we can estimate the power, area, and timing. Simulations indicate a single neuron will require $\approx$ 38 μW (2.46 mW for the whole array); this is split 60/40 between the array itself and the comparator (the output changes infrequently, so the buffer consumes only $\approx$ 1μW

per neuron). Our simulations also show that ≈ 300 MHz operation is possible.

The input layers of MNIST map "perfectly" into 196 x 64 arrays, so for this layer a 256x64 array has too many inputs; the output layer maps into a 256 x 10 array; now there are too many neurons.

The cost in timing is negligible, since it is almost completely driven by the comparator (T = 3.333 ns). The only additional timing cost is for the output layer, where the circuit has to drive a larger array (in neurons); using results from the component analysis, the cost is about 30 ps. The unused synapse weights can be set to $G_0 = 0.01$ μS, and the diodes and amplifiers in the comparator can be shut down. This means the power cost is also very small, related to the fact that the row driver has to be sized larger than necessary (about 7 μW for the output layer neurons, essentially zero for the input layer neurons). The area cost is substantial, since the output layer is 6.4X larger than necessary, and the input layers are 1.2X larger. This means there are the following penalties for inefficient array sizing:

- Timing penalty is 1.01X
- Power penalty is 1.001X
- Area penalty is 1.63X

Given the large differences in these penalties, the use of area as the primary factor is justified. This will not be the case when we evaluate the general purpose and special purpose designs, since they will have different communication networks, which do impose timing and power penalties that are larger than shown here. The area cost of the different NOC options can be estimated fairly easily.

5.4 Results of mapping the applications onto the architectural options

We explore three applications (MNIST, CSlite, and AES-256), three networks (direct connect, switch, mesh), and a large number of tile sizes (the two tables above have 56 tile sizes in total). The following tables provide the area estimates for the top 25 tile variations, grouped in three ways:

Table 5.10, Table 5.11, and Table 5.12 each rank the tiles for one network (across all 3 applications);

Table 5.13, Table 5.14, and Table 5.15 each rank the tiles for one application (across all 3 networks);

Table 5.16 is a combined overall ranking for the tiles (across all 3 applications and all 3 networks).

The differing alternatives are combined into a single ranking in each table by taking the geometric mean of the groupings. The geometric mean is an appropriate metric when the application workloads are known, but the relative usage of each application is not [33].

$$Geo_{\text{mean}} = [A * B * C * ... * N]^{1/\text{N}} \tag{5.1}$$

The results indicate that there are a small number of tile sizes that are near the top across all the rankings (256 x 32, 256 x 64, 128 x 32, 256 x 16, 128 x 16). It is interesting to note that the aspect ratios (inputs/outputs) of the arrays that provide

the best mappings include only 8:1, 4:1, and 16:1; square arrays are not optimal. Additionally, only two input sizes (128 and 256) are in the top group, although n=64 and n=512 tile sizes are near the top for some of the rankings. The tables also show that the A2A switch designs consistently have smaller areas than the 2D mesh designs (see Table 5.13, Table 5.14, Table 5.15, and Table 5.16). We will focus on the few tile sizes that are most efficient in comparing our general purpose neural processor to the special purpose and limited purpose designs (SPD, LPD).

Table 5.10: Ranking tiles for direct connect network using area (mm^2)

Inputs	Neurons	CSlite	AES-256	MNIST	GeoMean	Ratio
128	32	0.710	1.404	0.058	0.388	1.00
64	32	0.698	1.406	0.060	0.389	1.00
64	16	0.655	1.509	0.064	0.398	1.03
256	64	0.975	1.554	0.046	0.412	1.06
128	64	0.882	1.417	0.059	0.419	1.08
256	32	1.020	1.628	0.047	0.427	1.10
64	64	1.072	1.497	0.052	0.436	1.13
128	16	0.836	1.573	0.065	0.441	1.14
32	16	0.686	1.700	0.084	0.461	1.19
64	8	0.788	1.827	0.080	0.486	1.25
32	32	0.974	1.653	0.073	0.490	1.27
32	8	0.681	1.943	0.101	0.511	1.32
256	16	1.269	1.951	0.054	0.513	1.32
256	128	1.255	1.736	0.086	0.572	1.48
128	8	1.106	2.029	0.086	0.578	1.49
32	64	1.508	1.822	0.072	0.583	1.50
128	128	1.405	1.795	0.100	0.632	1.63
512	128	1.393	2.322	0.082	0.642	1.66
512	64	1.456	2.511	0.074	0.648	1.67
32	4	0.834	2.561	0.131	0.653	1.69
64	4	1.087	2.561	0.109	0.673	1.74
512	32	1.605	2.700	0.078	0.695	1.79
256	8	1.713	2.650	0.075	0.698	1.80
16	8	0.883	2.709	0.155	0.718	1.85
16	16	1.142	2.519	0.130	0.721	1.86

Table 5.11: Ranking tiles for A2A switch network using area (mm^2)

Inputs	Neurons	CSlite	AES-256	MNIST	GeoMean	Ratio
256	32	2.120	3.424	0.097	0.841	1.000
256	64	2.166	3.485	0.102	0.878	1.044
256	16	2.381	3.702	0.102	0.949	1.129
128	32	1.940	3.874	0.159	1.095	1.302
128	16	2.055	3.908	0.161	1.137	1.352
512	64	2.557	4.442	0.130	1.217	1.448
256	8	2.804	4.378	0.123	1.227	1.460
512	32	2.649	4.496	0.128	1.236	1.470
512	128	2.539	4.253	0.149	1.270	1.510
128	64	2.567	4.156	0.171	1.351	1.607
128	8	2.332	4.319	0.182	1.352	1.609
512	16	3.079	5.108	0.141	1.491	1.774
64	16	2.157	5.012	0.210	1.506	1.791
256	128	2.895	4.027	0.198	1.521	1.808
64	8	2.233	5.218	0.226	1.622	1.929
64	32	2.529	5.134	0.217	1.679	1.997
64	4	2.509	5.952	0.253	1.943	2.310
1024	64	3.516	6.353	0.187	2.044	2.431
1024	128	3.361	5.974	0.210	2.054	2.443
512	256	3.595	4.178	0.291	2.092	2.489
64	64	4.106	5.765	0.198	2.163	2.573
1024	32	3.902	6.639	0.190	2.217	2.637
1024	256	3.517	5.817	0.271	2.352	2.798
128	128	4.236	5.434	0.303	2.639	3.138
1024	512	5.049	5.847	0.266	2.801	3.331

Table 5.12: Ranking tiles for 2D mesh network using area (mm^2)

Inputs	Neurons	CSlite	AES-256	MNIST	GeoMean	Ratio
256	32	2.631	4.201	0.121	1.101	1.000
256	64	2.718	4.333	0.128	1.147	1.042
256	16	2.896	4.454	0.124	1.171	1.063
256	8	3.310	5.120	0.145	1.349	1.225
512	32	3.134	5.273	0.152	1.358	1.233
512	64	3.067	5.289	0.156	1.364	1.239
128	32	2.511	4.964	0.206	1.370	1.244
128	16	2.620	4.932	0.205	1.384	1.257
512	128	3.070	5.117	0.181	1.416	1.286
512	16	3.560	5.860	0.164	1.506	1.367
128	8	2.900	5.322	0.226	1.516	1.377
128	64	3.348	5.379	0.223	1.590	1.444
256	128	3.656	5.058	0.250	1.667	1.513
64	16	2.854	6.577	0.277	1.733	1.574
64	8	2.903	6.731	0.293	1.790	1.625
1024	64	4.005	7.201	0.213	1.832	1.663
1024	128	3.861	6.837	0.241	1.854	1.683
64	32	3.378	6.807	0.290	1.882	1.709
1024	32	4.383	7.416	0.213	1.907	1.731
64	4	3.168	7.465	0.319	1.962	1.781
512	256	4.366	5.074	0.354	1.987	1.804
1024	256	4.059	6.712	0.312	2.041	1.854
1024	16	5.225	8.670	0.242	2.221	2.017
64	64	5.512	7.696	0.265	2.241	2.035
1024	512	5.841	6.763	0.307	2.299	2.087

Table 5.13: Ranking tiles for CSlite application using area (mm^2)

Inputs	Neurons	Direct	A2A Switch	2D Mesh	Geomean	Ratio
128	32	0.710	1.940	2.511	1.512	1.000
64	16	0.655	2.157	2.854	1.591	1.052
128	16	0.836	2.055	2.620	1.651	1.091
64	8	0.788	2.233	2.903	1.722	1.139
256	32	1.020	2.120	2.631	1.785	1.180
256	64	0.975	2.166	2.718	1.790	1.184
64	32	0.698	2.529	3.378	1.813	1.199
32	8	0.681	2.660	3.577	1.865	1.233
128	8	1.106	2.332	2.900	1.955	1.293
128	64	0.882	2.567	3.348	1.964	1.299
32	4	0.834	2.664	3.512	1.983	1.311
64	4	1.087	2.509	3.168	2.052	1.357
32	16	0.686	3.062	4.163	2.060	1.362
256	16	1.269	2.381	2.896	2.061	1.362
512	128	1.393	2.539	3.070	2.214	1.464
512	64	1.456	2.557	3.067	2.252	1.489
256	128	1.255	2.895	3.656	2.368	1.566
512	32	1.605	2.649	3.134	2.371	1.567
32	2	1.217	3.008	3.838	2.413	1.595
256	8	1.713	2.804	3.310	2.514	1.662
16	4	0.976	4.008	5.413	2.766	1.829
16	8	0.883	4.213	5.758	2.777	1.836
512	16	2.039	3.079	3.560	2.817	1.863
64	64	1.072	4.106	5.512	2.895	1.914
32	32	0.974	4.709	6.441	3.091	2.044

Table 5.14: Ranking tiles for AES-256 application using area (mm^2)

Inputs	Neurons	Direct	A2A Switch	2D Mesh	Geomean	Ratio
256	32	1.628	3.424	4.201	2.861	1.000
256	64	1.554	3.485	4.333	2.863	1.001
128	32	1.404	3.874	4.964	3.000	1.049
128	16	1.573	3.908	4.932	3.118	1.090
128	64	1.417	4.156	5.379	3.164	1.106
256	16	1.951	3.702	4.454	3.180	1.112
256	128	1.736	4.027	5.058	3.282	1.147
128	8	2.029	4.319	5.322	3.600	1.258
512	256	2.246	4.178	5.074	3.624	1.267
64	32	1.406	5.134	6.807	3.663	1.280
64	16	1.509	5.012	6.577	3.678	1.285
512	128	2.322	4.253	5.117	3.697	1.292
256	256	1.991	4.687	5.937	3.812	1.332
512	64	2.511	4.442	5.289	3.893	1.360
256	8	2.650	4.378	5.120	3.902	1.364
512	32	2.700	4.496	5.273	4.000	1.398
64	8	1.827	5.218	6.731	4.004	1.399
64	64	1.497	5.765	7.696	4.050	1.416
128	128	1.795	5.434	7.090	4.104	1.435
512	16	3.357	5.108	5.860	4.649	1.625
64	4	2.561	5.952	7.465	4.846	1.694
32	16	1.700	7.630	10.320	5.115	1.788
32	32	1.653	8.033	10.936	5.257	1.837
32	8	1.943	7.626	10.201	5.327	1.862
1024	256	3.884	5.817	6.712	5.333	1.864

Table 5.15: Ranking tiles for MNIST application using area (mm^2)

Inputs	Neurons	Direct	A2A Switch	2D Mesh	Geomean	Ratio
256	32	0.047	0.097	0.121	0.082	1.000
256	64	0.046	0.102	0.128	0.084	1.030
256	16	0.054	0.102	0.124	0.088	1.079
256	8	0.075	0.123	0.145	0.110	1.342
512	32	0.078	0.128	0.152	0.115	1.399
512	64	0.074	0.130	0.156	0.115	1.402
128	32	0.058	0.159	0.206	0.124	1.517
128	16	0.065	0.161	0.205	0.129	1.578
512	16	0.094	0.141	0.164	0.129	1.579
512	128	0.082	0.149	0.181	0.130	1.589
128	64	0.059	0.171	0.223	0.131	1.598
64	64	0.052	0.198	0.265	0.139	1.701
128	8	0.086	0.182	0.226	0.152	1.858
64	16	0.064	0.210	0.277	0.155	1.887
64	32	0.060	0.217	0.290	0.156	1.900
256	128	0.086	0.198	0.250	0.162	1.979
1024	64	0.131	0.187	0.213	0.173	2.115
64	8	0.080	0.226	0.293	0.174	2.123
1024	32	0.139	0.190	0.213	0.178	2.171
1024	128	0.143	0.210	0.241	0.193	2.359
64	4	0.109	0.253	0.319	0.207	2.522
1024	16	0.172	0.220	0.242	0.209	2.553
128	128	0.100	0.303	0.396	0.229	2.795
32	32	0.073	0.354	0.484	0.232	2.836
32	64	0.072	0.364	0.500	0.236	2.877

Table 5.16: Ranking tiles across applications and networks using area (mm^2)

Inputs	Neurons	Direct	A2A Switch	2D Mesh	Geomean	Ratio
256	32	0.427	0.841	1.101	0.734	1.000
256	64	0.412	0.878	1.147	0.746	1.016
256	16	0.513	0.949	1.171	0.829	1.130
128	32	0.388	1.095	1.370	0.835	1.137
128	16	0.441	1.137	1.384	0.886	1.207
128	64	0.419	1.351	1.590	0.965	1.316
64	16	0.398	1.506	1.733	1.012	1.380
512	64	0.648	1.217	1.364	1.024	1.396
512	128	0.642	1.270	1.416	1.049	1.430
256	8	0.698	1.227	1.349	1.049	1.430
512	32	0.695	1.236	1.358	1.053	1.435
128	8	0.578	1.352	1.516	1.058	1.442
64	32	0.389	1.679	1.882	1.071	1.460
64	8	0.486	1.622	1.790	1.121	1.528
256	128	0.572	1.521	1.667	1.132	1.542
512	16	0.862	1.491	1.506	1.246	1.699
64	64	0.436	2.163	2.241	1.283	1.749
64	4	0.673	1.943	1.962	1.369	1.865
512	256	0.879	2.092	1.987	1.541	2.099
32	16	0.461	2.965	2.802	1.565	2.133
32	8	0.511	2.822	2.681	1.569	2.138
128	128	0.632	2.639	2.498	1.609	2.193
1024	128	1.096	2.054	1.854	1.610	2.194
1024	64	1.125	2.044	1.832	1.615	2.201
1024	32	1.245	2.217	1.907	1.740	2.371

Chapter 6: Evaluation

The primary evaluation is to compare the general purpose neural processors (GP), utilizing a single tile mapping and network (NOC), to the special purpose designs (SPD). In the SPD, each component of the application uses the most efficient array size, and direct connections are used (no NOC). For each application, we will also develop a limited purpose design (LPD); this design will assume a single array size for the entire application, but will still assume direct connections. Finally, we will compare our GP designs that use tiles, to GP designs that assume the tile concept had not been invented. This will provide a quantification of the value of the tile concept.

6.1 Criteria

The primary criteria for any design is performance, but it is usually better to evaluate options on a normalized basis, in terms of Performance/Watt or Performance/Cost [22, 77]. Since neural nets typically are used for streaming applications or as accelerators, the proper performance metric is application throughput, defined as number of new input bits per second. While costs for a design can be estimated,

using chip area is a reasonable proxy [77]. This provides us with two primary metrics:

- Throughput/Watt (T/W)
- Throughput/Area (T/A)

with Watts/Area (W/A) as a constraint due to cooling.

The most important comparison will be based on the relative capabilities of the SPD and GP neural processing architectures. Current architectures are also evaluated in this way, and will serve as a guide for our conclusion [21, 83]. To justify highly specialized architectures, typical improvements in the 100X range in Ops/Watt and Ops/Area are needed, absent a compelling need for ultimate performance or energy efficiency. If the GP neural processor is within 10X of the SPD options in both T/W and T/A, that will be a strong argument for the practicality of a general purpose neural processor. If the GP capabilities are $\geq$100X worse in both T/W and T/A, this would indicate a general purpose neural processor, based on our design, is not practical.

6.2 Methodology

For each design point, the area will be calculated as described previously. For directly connected designs, the timing will be straightforward; it will be the speed of the comparator (3.333 ns/300 MHz). For the networked designs, a worst-case tim-

ing analysis will be done based on the actual size of the design (number of neurons) needed for the application. Worst-case latency is especially important for neural nets in time critical scenarios, such as on-line translation services [33] and some types of cybersecurity applications. Keep in mind that the initial analysis of the hierarchical A2A suggested that approximately 500,000 neurons are reachable in only 6 cycles (or 6 ns); this means the time delay for this design will at most be a small multiple of the comparator delay (assumed to be 4 ns to account for the possibility of horizontal tiling). The 2D mesh NOC has a much larger time delay for worst-case timing.

The power data from LTspice simulations and the actual NOC implementations have been parameterized in terms of inputs, neurons, clock frequency, and tile size. Once the compute + communicate timing is known, it will be straightforward to estimate the power. Finally the T/W and T/A metrics (and W/A) will be calculated, assuming a chip size of 10 mm^2. This will enable the capabilities of the designs to be compared.

Let us work through an example calculation, using the AES-256 application and one of the general purpose tile sizes (in this case, a 128x16 tile), and assuming the A2A switch is used for the NOC. This will require 817 tiles in total, which is found by determining the number of tiles needed for each of the components (MixAB, MixC, State Machine, SubBytes) and adding them together. The first part of the analysis is to find the compute time and the communication time; we assume they do not overlap. The compute time for tiles will always be assumed

to be 4 ns, allowing for the potential additional delay when inputs need to cross multiple tiles. The communication time is found by determining how many switch levels are needed. At level 1, a switch has 16 ports for connection to the tiles, and each port is connected to 16 neurons (sparsity of communication leveraging). So we can calculate the number of level 1 switches using:

Number of switches at Level 1 is

NS1 = #neurons/(16 * 16) = #neurons/256 = (817 * 16)/256 = 51.06, or 52 switches

This same ratio of switch ports continues at every successive level, so

NS2 = NS1/16 = 52/16 = 3.25, or 4 switches

NS3 = 0, since at any level up to 9 switches can directly connected (NS3 = 0 since NS2 $\leq$ 9).

So we have 52 + 4 = 56 switches in total. For all of the applications we are looking at, no NS3 switches will be needed. Since we will need two levels of switches, a communication could pass through as many as 4 switches (2 up, 2 down); a switch is assumed to have a latency of 1 ns, so the worst-case communication time is 4 ns. This provides us enough information to calculate power, area, and timing.

Timing = Compute time + Communication time = 4 + 4 = 8 ns;

Frequency = 125 MHz = 0.125 GHz

The area is the previous total tile area (3.868 mm^2), which already includes any level 1 switches, plus the area of the level 2 switches (4 * .043 mm^2).

Area (mm^2) = 3.868 + (4 * 0.043) = 4.040 mm^2

The total power includes the following. The values come from the neuron simulations, except for the NOC power, which is estimated from a scaled down version of the A2A switch (or from the measurement of the FPGA for the 2D mesh).

Input circuits (excluding the row driver) = 0.11 μW/GHz/input/tile

Row driver = 0.01 μW/GHz/input/neuron/tile

Output buffer = 6 μW/GHz/neuron/tile

Switch = 250.8 μW/GHz/switch

The unit cell and the comparator need to have their power calculated slightly differently, since these circuits are always using power while active. We will scale this power based on the ratio of compute time to total time (or activity factor).

α = compute/(compute + communicate) = 4/ (4 + 4) = 0.5

Unit cell = α * 0.0825 μW/input/neuron/tile

Comparator = α * 15 μW/neuron/tile

So now we can calculate the total power for our example

$$Input = 0.11 * 0.125 * 128 * 817 = 1437.9\mu W \tag{6.1}$$

$$Rowdriver = 0.01 * 0.125 * 128 * 16 * 817 = 2091.5\mu W \tag{6.2}$$

$$Outputbuffer = 6.0 * 0.125 * 16 * 817 = 9804.0\mu W \tag{6.3}$$

$$Switch = 250.8 * 0.125 * 56 = 1755.6\mu W \tag{6.4}$$

$$UnitCell = 0.5 * 0.0825 * 128 * 16 * 817 = 69020.2\mu W \tag{6.5}$$

$$Comparator = 0.5 * 15 * 16 * 817 = 98040.0\mu W \tag{6.6}$$

$$Total = 1437.9{+}2091.5{+}9804.0{+}1755.6{+}69020.2{+}98040.0 = 182149.2\mu W = 182.1mW \tag{6.7}$$

The dominant factors in the power are the unit cells and the comparator; for this example, they comprise 37.9% and 53.8% of the power, respectively (combined, 91.7% of the total).

With this information, we can estimate the key metrics:

$$Throughput = Bits/cycle * cycles/sec = 128 * 125 * 10^6 = 16Gbps \tag{6.8}$$

$$T/W = 16 * 10^9/182.1 * 10^{-3} = 87.9Gbps/W \tag{6.9}$$

$$T/A = 16 * 10^9/4.040 = 3.96Gbps/mm^2 \tag{6.10}$$

$$W/A = 182.1 * 10^{-3}/4.040 = 0.045W/mm^2 \tag{6.11}$$

As long as W/mm^2 stays below 0.5, power density is not an issue (air cooling will be possible).

6.3 Special purpose designs and capabilities for the three applications

For each of the three applications (MNIST, CSlite, and AES-256) it is possible to construct a highly specialized or special purpose design (SPD) by identifying the optimal array size for each component of the application. Optimal array size can be found by using array utilization, as described in the previous chapter, or by inspection of the mapping. We will use the following array sizes for our 3 SPDs:

MNIST:

Input layer: 4 (192x64) arrays

Output layer: 1 (256x10) array

CSlite:

Byte Decoder: 6 (8x256) arrays

Signature: 140 (64x16) arrays

SetHold: 125 (33x16) arrays

Detector 12 (512x32) arrays

AES-256:

MixAB: 8 (256x256) arrays

MixC: 16 (64x32) arrays

State Machine: 1 (16x16) array

SubBytes 1: 256 (16x16) arrays

SubBytes 2: 208 (256x16) arrays

There are two different array sizes for the SubBytes component, because the individual parts require very different array sizes. As you can see, the SPD array sizes being used vary significantly (8x256) to (16x16) to (512x32). Using these array sizes, and assuming direct connections are used (no NOC), we can estimate the T/W and T/A for the three applications (Table 6.1). The MNIST T/W and T/A values are so much higher than the other applications because there are 768 bits of input on each cycle (vs. 8 bits for CSlite and 128 for AES-256).

Table 6.1: Calculation of metrics for the 3 SPD architectures

Application	T/W (Gbps/W)	T/A (Gbps/mm^2)	W/A (W/mm^2)
SPD.MNIST	25818.1	6859.6	0.27
SPD.CSlite	17.2	5.6	0.33
SPD.AES256	129.7	33.8	0.26

6.4 Limited purpose designs and capabilities for the three applications

The SPD are extremely specialized; we can also look at designs that are slightly less specialized, or limited purpose designs (LPD) for the applications. For the LPD we assume that each application uses a single array size for the whole application. This will give us some insight into how much of the SPD T/W and T/A advantages are due to the extreme specialization. The array size is chosen to use the maximum input size and output size (or an integer fraction of the maximum output size). We

will analyze the following designs:

MNIST:

LPD.MNIST.256X64: 5 (256x64) arrays

LPD.MNIST.256X16: 17 (256x16) arrays

CSlite:

LPD.CSlite.512x32: 186 (512x32) arrays

LPD.CSlite.512x16: 370 (512x16) arrays

AES-256:

LPD.AES256.256X16: 609 (256x16) arrays

We find the following: (Table 6.2)

Table 6.2: Calculation of metrics for the application-centric LPDs

Application	T/W (Gbps/W)	T/A (Gbps/mm^2)	W/A (W/mm^2)
LPD.MNIST.256X64	18547.1	5009.7	0.27
LPD.MNIST.256X16	21599.8	4230.5	0.20
LPD.CSlite.512x32	6.6	1.5	0.23
LPD.CSlite.512x16	6.6	1.2	0.18
LPD.AES256.256X16	100.5	18.5	0.26

The CSlite SPD is much better (3X to 4X higher in T/W and T/A) than the LPD; for the other applications, the difference is more modest (1.5X to 2X). The highly specialized array selection clearly provides benefits.

6.5 General purpose designs and capabilities

Finally, we can look at our general purpose designs. We will look at two different tile sizes (128x16 and 256x64), which represent the smallest and largest of the tile sizes that were at the top of the Geo_{mean} tile rankings. We will also look at two array sizes (512x32 and 512x16); these represent the best array sizes for the applications assuming tiles had not been invented. Including this in our exploration will help quantify the value of the tile feature. The Detector component in CSlite is the limiting factor in selecting an array, and is best matched to a 512x32 array. Since many of the other components are reasonably well-matched to x16 array sizes, we also include this. We will also look at both the A2A switch and the 2D mesh NOC options.

Our estimates for these general purpose designs using the A2A switch are (Table 6.3):

and for the 2D Mesh (Table 6.4):

The T/W values are not much different for the A2A and 2D Mesh NOC options, but there is a significant difference in the T/A values (3X to 8X). The tile feature shows a clear advantage when compared to the array, in both T/W (1.5X to 2X) and T/A (1.2X to 1.5X). The two tile options (128x16, 256x64) are fairly close in T/W and T/A (within about 1.2X).

Table 6.3: Calculation of metrics for the GP designs with the A2A switch

Application	T/W (Gbps/W)	T/A (Gbps/mm^2)	W/A (W/mm^2)
GP.MNIST.128x16Tile	12629.5	795.3	0.06
GP.MNIST.256x64Tile	15479.1	1252.9	0.08
GP.MNIST.512x32Array	10947.7	998.5	0.09
GP.MNIST.512x16Array	11503.8	904.9	0.08
GP.CSlite.128x16Tile	10.3	0.47	0.05
GP.CSlite.256x64Tile	7.6	0.44	0.06
GP.CSlite.512x32Array	5.5	0.37	0.07
GP.CSlite.512x16Array	5.5	0.32	0.06
GP.AES256.128x16Tile	87.8	3.96	0.05
GP.AES256.256x64Tile	76.6	4.47	0.06
GP.AES256.512x32Array	52.6	3.49	0.07
GP.AES256.512x16Array	53.7	3.08	0.06

Table 6.4: Calculation of metrics for the GP designs with the 2D mesh router

Application	T/W (Gbps/W)	T/A (Gbps/mm^2)	W/A (W/mm^2)
GP.MNIST.128x16Tile	12566.6	249.48	0.02
GP.MNIST.256x64Tile	15239.7	460.80	0.03
GP.MNIST.512x32Array	10880.6	389.61	0.04
GP.MNIST.512x16Array	11475.3	361.13	0.03
GP.CSlite.128x16Tile	10.3	0.07	0.01
GP.CSlite.256x64Tile	7.5	0.07	0.01
GP.CSlite.512x32Array	5.5	0.06	0.01
GP.CSlite.512x16Array	5.5	0.05	0.01
GP.AES256.128x16Tile	87.2	0.43	0.00
GP.AES256.256x64Tile	75.1	0.54	0.01
GP.AES256.512x32Array	52.1	0.44	0.01
GP.AES256.512x16Array	53.4	0.41	0.01

6.6 Analysis

We will use the Geo_{mean} calculation, combining all 3 applications, to compare the SPD and GP designs. The results are shown in Table 6.5.

Table 6.5: Combined Results using Geo_{mean} for the SPD (no NOC) and GP (A2A and 2D mesh) designs

Design type	T/W (Gbps/W)	T/A (Gbps/mm^2)	W/A (W/mm^2)
SPD	385.9	108.9	0.28
GP.128x16Tile.A2A	225.5	11.4	0.05
GP.256x64Tile.A2A	208.3	13.5	0.07
GP.512x32Array.A2A	147.2	10.8	0.07
GP.512x16Array.A2A	150.5	9.6	0.06
GP.128x16Tile.Mesh	224.0	1.9	0.01
GP.256x64Tile.Mesh	204.7	2.5	0.01
GP.512x32Array.Mesh	146.0	2.2	0.01
GP.512x16Array.Mesh	149.8	2.0	0.01

The SPD is 1.8X better in T/W than the GP designs using tiles, and 8X to 10X better in T/A than the GP tile design with the A2A switch (40 - 50X better than the GP tile design with the 2D mesh). The improvement in T/W for the SPD is largely a result of having direct connections between neurons; a recent study shows that designing convolutional neural networks for specific data movement characteristics optimizes energy efficiency [84]. The T/W and T/A for the SPD are clearly better than for the GP design, but the improvement is not overwhelming when considering that the SPD is suitable for neural net inferencing on one application only. The GP tile designs are useful across a range of applications. These results strongly indicate that a GP tile design with an A2A switch is practical and worth building. The GP

tile design with the 2D mesh is much more dependent on the specific communication patterns; for the worst-case analysis used here, it is clearly inferior to the A2A switch option. These results are consistent with other studies that have explored NOC designs [85–87]. The use of the tile feature provides clear benefits (1.5X in T/W and 1.2X in T/A for the GP tile design versus the GP array design using the A2A switch).

Chapter 7: Conclusions

We have carefully explored a variety of details needed to build a general purpose neural processor. Using a wider variety of applications than typically explored (MNIST, CSlite, and AES-256), we developed a general purpose neural processor that was shown to be very capable, even when compared to highly specialized designs (application specific SPD). The GP processor has a T/W within 2X of the SPD, and T/A within 10X; given the highly flexible application space of our GP design, it is clearly worth building. Two communication networks (NOC) were explored: a traditional 2D mesh and a hierarchical all-to-all (A2A) high performance switch. The A2A had clearly superior worst-case latency timing when compared to the 2D mesh, which results in a 5X improvement in T/A. Leveraging sparsity of communication helps to reduce the overall impact of either NOC on power and area.

This design incorporated a new feature, which we call a tile, to improve its capabilities. The tile concept enables smaller arrays to be used by themselves for an application, or they can be combined both horizontally (to increase the neurons) or vertically (to increase the inputs). This makes the GP design much more flexible; clear improvements in both T/W (1.5X) and T/A (1.2X) were identified when

compared to GP designs without the tile feature incorporated.

The tile design is enabled by the use of a compact (55 μm^2), low power (15 μW), and fast (250 MHz) differential current comparator. This reduces the penalties with using the tiles, and takes advantage of the 1/High Z differential architecture, which is shown to reduce array power by 3-7X over other array architectures. The comparator design included programmable memristors to minimize the effect of parameter and device mismatches, multiple controllable diodes to minimize power and provide optimal operating range, T_{strobe} controlled gates to increase performance, and special bypass FETs to enable the tile concept.

We made a detailed examination of the potential limits on memristor crossbar array sizes, including parasitic effects on timing, voltage drops through devices and parasitic resistances, and memristor programming precision. We identified processes for adjusting the programmed conductance to account for these effects when necessary. We provided general guidelines for mapping weights onto the array to minimize these effects.

Finally, we suggested modest enhancements to the memristor model, and identified mechanisms for automating the inclusion of tiles in the Loom neural compiler.

- es ist vollbracht

References

[1] Agarwal et al. Energy scaling advantages of resistive memory crossbar based computation and its application to sparse coding. *Frontiers of Neuroscience*, 9(484):1–9, January 2016.

[2] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[3] R. Dennard et al. Design of ion-implanted mosfets with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, SC-9(5):256–268, October 1974.

[4] White House Office of Science and Technology Policy. A nanotechnology-inspired grand challenge for future computing. `https://www.whitehouse.gov/blog/2015/10/15/nanotechnology-inspired-grand-challenge-future-computing`, first accessed October 20, 2015.

[5] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1936.

[6] D. Mountain. Neuromorphic computing: Our approach to developing applications using a new model of computing. *IEEE Rebooting Computing Summit 4*, December 2015.

[7] Krizhevsky et al. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, December 2012.

[8] Q.V. Le. Building high-level features using large scale unsupervised learning. *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8595–8598, May 2013.

[9] Hannun et al. Deepspeech: Scaling up end-to-end speech recognition. *arXiv:1412.5567v2*, December 2014.

[10] Hinton and Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[11] G. Hinton. *A practical guide to training restricted boltzmann machines*. Neural Networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012.

[12] Y. LeCun et al. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[13] Yangjie et al. Fpga design of a multicore neuromorphic processing system. *IEEE Aerospace and Electronics Conference*, pages 255–258, June 2014.

[14] J. von Neumann. First draft of a report on the edvac. *Contract No. W-670-ORD-4926*, June 1945.

[15] Wulf and McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH*, 23(1):20–24, March 1995.

[16] M. Gokhale et al. Processing in memory: The terasys massively parallel pim array. *IEEE Computer*, 28(4):23–31, April 1995.

[17] T. Sterling and P. Kogge. An advanced pim architecture for spaceborne computing. *Proceedings of the IEEE Aerospace Conference*, 5, March 1998.

[18] K. Iniewski. *CMOS Processors and Memories*. Springer Science and Business Media, 2010.

[19] Byeong-Gyu Nam and Hoi-Jun Yoo. *Graphics Processing Unit: Algorithm, Architecture, and Power*. Taylor and Francis, November 2014.

[20] A. George et al. Novo-g: At the forefront of scalable reconfigurable supercomputing. *IEEE Computing in Science and Engineering*, 13(1):82–86, January 2011.

[21] D. Shaw et al. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 51(7):91–97, July 2008.

[22] A. Shafiee et al. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, pages 14–26, June 2016.

[23] C. Yakopcic et al. Memristor spice model and crossbar simulation based on devices with nanosecond switching time. *International Joint Conference on Neural Networks*, pages 1–7, August 2013.

[24] D. Mountain et al. Ohmic weave: Memristor-based threshold gate networks. *IEEE Computer*, 48(12):65–71, December 2015.

[25] R. Hasan and T. Taha. Enabling back propagation training of memristor crossbar neuromorphic processors. *International Joint Conference on Neural Networks*, pages 21–28, 2014.

[26] P. Merolla et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, August 2014.

[27] Akopyan et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, October 2015.

[28] Sawada et al. Truenorth ecosystem for brain-inspired computing: Scalable systems, software and applications. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analytics*, pages 130–141, November 2016.

[29] M. Hu et al. Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication. *Design Automation Conference*, June 2016.

[30] J. Schmidhuber. Deep learning in neural networks: an overview. *Neural Networks*, 61:85–117, January 2015.

[31] Y. LeCun et al. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[32] E. Raff et al. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, September 2016.

[33] Jouppi et al. In-datacenter performance analysis of a tensor processor unit. *ACM/IEEE International Symposium on Computer Architecture*, June 2017.

[34] Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[35] Rumelhart et al. Learning representations by backpropagating errors. *Nature*, 323(6088):533–536, October 1986.

[36] Y. LeCun. A theoretical framework for back-propagation. *Proceedings of the 1988 Connectionist Models Summer School*, pages 21–28, 1988.

[37] L. Chua. Memristor: The missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–529, September 1971.

[38] L. Chua and S. Kang. Memristive devices and systems. *Proceedings of the IEEE*, 64(2):209–223, February 1976.

[39] D. Strukov et al. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

[40] L. Chua. Resistance switching memories are memristors. *Applied Physics A*, 102(4):765–783, March 2011.

[41] Adhikari et al. Three fingerprints of a memristor. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 60(11):3008–3021, November 2013.

[42] Yakopcic et al. Memristor-based neuron circuit and method for applying learning algorithm in spice. *Electronic Letters*, 50(7):492–494, March 2014.

[43] Yang et al. Memristive devices for computing. *Nature Nanotechnology*, 8(1):13–24, January 2012.

[44] Waser et al. Redox-based resistive switching memories: Nanoioinic mechanisms, prospects, and challenges. *Advanced Materials*, 21(25-26):2632–2663, 2009.

[45] J. Hasler and Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in Neuroscience*, 7(118):1–29, September 2013.

[46] W. Wilcke. The ibm cortical learning center project. *Neuro-Inspired Computational Elements (NICE)*, 2015.

[47] Indiveri et al. Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, 5(73):1–23, May 2011.

[48] M. Hu et al. Memristor crossbar-based neuromorphic computing system: A case study. *IEEE Transactions on Neural Networks and Learning Systems*, 25(10):1864–1878, October 2014.

[49] T. Taha et al. Memristor crossbar based multicore neuromorphic processors. *IEEE International System-on-Chip Conference (SOCC)*, pages 383–388, September 2014.

[50] Chenchen et al. A spiking neuromorphic design with resistive crossbar. *Design Automation Conference (DAC)*, pages 14–1 to 14–6, 2015.

[51] D. Khudithipudi. *Design of Neuromorphic Architectures with Memristors*. Network Science and Cybersecurity. Springer, 20143.

[52] J. Yang and S. Williams. Memristive devices in computing system: Promises and challenges. *ACM Journal on Emerging Technologies in Computing Systems*, 9(2):11:1 – 11:20, May 2013.

[53] Chi et al. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM/IEEE International Symposium on Computer Architecture*, pages 27–39, June 2016.

[54] M. Heroux et al. Asc co-design proxy app strategy. *Technical report LA-UR-13-20460/LLNL-TR-592878*, January 2013.

[55] Hochreiter et al. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*. A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press, 2001.

[56] E. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242, June 1990.

[57] M. McLean. *Concurrent Learning Algorithm and the Importance Map*. Network Science and Cybersecurity. Springer, 2014.

[58] R. Hasan et al. High throughput neural network based embedded streaming multicore processors. *IEEE International Conference on Rebooting Computing*, October 2016.

[59] NIST. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication 197*, 2001.

[60] Linear Technologies. Ltspice iv. `http://www.linear.com/designtools/software/`, first accessed June 6, 2014.

[61] D. Mountain. Technology considerations for neuromorphic computing. *IEEE International Conference on Rebooting Computing*, October 2016.

[62] Arizona State University. High performance models incorporating high-k/metal gate and stress effects; 45nm ptm hp model:v2.1. `http://ptm.asu.edu/latest.html`, first accessed June 6, 2014.

[63] P. Sheridan et al. Device and spice modeling of rram devices. *Nanoscale*, 3(9):3833–3840, September 2011.

[64] B. Jacob et al. *Memory Systems: Cache, DRAM, Disk*. Elsevier, 2008.

[65] A. Chen. Nonlinearity and asymmetry for device selection in cross-bar memory arrays. *IEEE Transactions on Electron Devices*, 62(9):2857–2864, September 2015.

[66] F. Alibart et al. High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm. *Nanotechnology*, 23:1–7, January 2012.

[67] T. Gokmen and Y. Vlasoz. Acceleration of deep neural network training with resistive cross-point devices: Design considerations. *Frontiers in Neuroscience*, 10:1–13, July 2016.

[68] C. Yakopicic et al. Memristor model optimization based on parameter extraction from device characterization data. *to be submitted to the IEEE Transactions on Nanotechnology*, 2017.

[69] D. Mahalanabis et al. Impedance measurement and characterization of agge30se70-based programmable metallization cells. *IEEE Transactions on Electron Devices*, 61(11):3723–3730, November 2014.

[70] K. Zhang. Memory trends. *International Solid State Circuits Conference (ISSCC)*, 2013-2015.

[71] P. Belemjian and N. Sfida. Project nemo. *Internal NSA technical report version 1.1*, August 2016.

[72] J. Partzsch and R. Schuffny. Analyzing the scaling of connectivity in neuromorphic hardware and in models of neural networks. *IEEE Transactions on Neural Networks*, 22(6):919–935, June 2011.

[73] Rajendran et al. Specifications of nanoscale devices and circuits for neuromorphic computational systems. *IEEE Transactions on Electron Devices*, 60(1):246–253, January 2013.

[74] Knag et al. A sparse coding neural network asic with on-chip learning for feature extraction and encoding. *EEE Journal of Solid-State Circuits*, 50(4):1070–1079, April 2015.

[75] Chen et al. Dadiannao: A machine-learning supercomputer. *Proceedings of MICRO-47*, pages 609–622, December 2014.

[76] Furber et al. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–664, May 2014.

[77] D. Mountain. Analyzing the value of using three-dimensional electronics for a high-performance computational system. *IEEE Transactions on Advanced Packaging*, 31(1):107–117, February 2008.

[78] Gao et al. Programming protocol optimization for analog weight tuning in resistive memories. *IEEE Electron Device Letters*, 36(11):1157–1159, November 2015.

[79] E. Keiter et al. Xyce parallel electronic simulator, users guide version 5.1. *Sandia Report SAND2009-7572*, November 2009.

[80] C. Krieger et al. A comparison of multiple computer architectures for neuromorphic computing. *to be submitted*, 2017.

[81] Ji et al. Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints. *49th Annual IEEE Symposium on Microarchitecture (MICRO 2016)*, pages 1–13, October 2016.

[82] S. Muroga. *Threshold Logic and Its Applications*. John Wiley and Sons, 1972.

[83] Cullinan et al. Computing performance benchmarks among cpu, gpu, and fpga. *Www.Wpi EduPubsE-Proj.-Proj.-030212- 123508unrestrictedBenchmarking Final*, 2013.

[84] Chen et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, January 2017.

[85] Pande et al. Fixed latency on-chip interconnect for hardware spiking neural network architectures. *Parallel Computing*, 39:357–371, April 2013.

[86] Carrillo et al. Scalable hierarchical network-on-chip architecture for spiking neural network hardware implementations. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2451–2461, December 2013.

[87] P. Merolla et al. A multicast tree router for multichip neuromorphic systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(3):820–832, March 2014.