A Scalable Flash-Based Hardware Architecture for the Hierarchical Temporal Memory Spatial Pooler

Lennard G. Streat
lgs8331@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
A Scalable Flash-Based Hardware Architecture for the Hierarchical Temporal Memory Spatial Pooler

by

Lennard G. Streat

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Dhireesha Kudithipudi
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2016

Approved by:

Dr. Dhireesha Kudithipudi, Associate Professor
Thesis Advisor, Department of Computer Engineering

Dr. Marcin Łukowiak, Associate Professor
Committee Member, Department of Computer Engineering

Dr. Andreas Savakis, Professor
Committee Member, Department of Computer Engineering
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

A Scalable Flash-Based Hardware Architecture for the Hierarchical Temporal Memory Spatial Pooler

I, Lennard G. Streat, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

________________________________________________________________________
Lennard G. Streat

________________________________________________________________________
Date
© Copyright 2016 by Lennard G. Streat
All Rights Reserved
To my supportive and loving family and girlfriend, may God bless you—I hope to see you all in Heaven someday...
Acknowledgments

I would like to thank and acknowledge everyone that helped make this work possible. The continual patience and support of Dr. Dhireesha Kudithipudi will be remembered as a valuable asset. I am also grateful for the advice and feedback from James Mnatzaganian, regarding HTM.

To my committee members, I must also give a great deal of thanks and acknowledgment for the timely support and for reviewing my work. Thank you Cibele Eller Rodrigues and Neil Wong Hon Chan for grammar checking this document. I owe numerous thanks to the Hanuch and McDonald families for treating me like a part of their family—giving me a home away from home. MCAS, Ms. Laurie Clayton, Lorrie Hendron, Tracy Gatewood, Tomicka Wagstaff and the McNair scholars program have also been invaluable resources to me over the course of my college career. Finally, I would like to acknowledge all the other members of the Nanocomputing Research Group.
Abstract

A Scalable Flash-Based Hardware Architecture for the Hierarchical Temporal Memory Spatial Pooler

Lennard G. Streat

Supervising Professor: Dr. Dhireesha Kudithipudi

Hierarchical temporal memory (HTM) is a biomimetic machine learning algorithm focused upon modeling the structural and algorithmic properties of the neocortex. It is comprised of two components, realizing pattern recognition of spatial and temporal data, respectively. HTM research has gained momentum in recent years, leading to both hardware and software exploration of its algorithmic formulation. Previous work on HTM has centered on addressing performance concerns; however, the memory-bound operation of HTM presents significant challenges to scalability.

In this work, a scalable flash-based storage processor unit, Flash-HTM (FHTM), is presented along with a detailed analysis of its potential scalability. FHTM leverages SSD flash technology to implement the HTM cortical learning algorithm spatial pooler. The ability for FHTM to scale with increasing model complexity is addressed with respect to design footprint, memory organization, and power efficiency. Additionally, a mathematical model of the hardware is evaluated against the MNIST dataset, yielding 91.98% classification accuracy. A fully custom layout is developed to validate the design in a TSMC 180nm process. The area and power footprints of the spatial pooler are 30.538mm$^2$ and 5.171mW, respectively. Storage processor units have the potential to be viable platforms to support implementations of HTM at scale.
# Contents

Dedication ................................................................. iv

Acknowledgments ....................................................... v

Abstract ........................................................................ vi

1 Introduction ............................................................... 1
   1.1 Motivation .......................................................... 1
   1.2 Contributions ...................................................... 3
   1.3 Outline .................................................................. 4

2 Background & Related Work ........................................ 5
   2.1 Flash Memory Technology ...................................... 5
       2.1.1 Industry Implementations ................................. 7
       2.1.2 Flash Cell Scaling .......................................... 10
       2.1.3 Large-Scale Memory Systems ......................... 12
       2.1.4 In-Memory Compute ...................................... 16
   2.2 Neuromorphic Hardware Architectures .................... 19
       2.2.1 SpiNNaker .................................................... 19
       2.2.2 HICANN Analog Network Core ....................... 21
       2.2.3 IBM TrueNorth ............................................. 22
       2.2.4 Neurogrid Hardware Accelerator ...................... 24
       2.2.5 HRL Neuromorphic Chip ................................. 24
       2.2.6 Hierarchical Temporal Memory Hardware ............ 25
### 2.3 Hierarchical Temporal Memory
- 2.3.1 Generation I: Zeta
- 2.3.2 Generation II: CLA
- 2.3.3 Spatial Pooler

### 3 Proposed Flash-Based HTM Design
- 3.1 Spatial Pooler Math Model
  - 3.1.1 Phase I: Overlap
  - 3.1.2 Phase II: Inhibition
  - 3.1.3 Phase III: Learning
- 3.2 Transaction-level Model
  - 3.2.1 Simulation Controller
  - 3.2.2 Host Processor Module
  - 3.2.3 SPU Module
- 3.3 Register Transfer Level Model
  - 3.3.1 Overlap Pipeline
  - 3.3.2 Channel Arbiter
  - 3.3.3 Inhibition Engine
  - 3.3.4 Content-Addressable Memory Channel Arbiter
  - 3.3.5 Write-back Content-Addressable Memory
  - 3.3.6 Content-Addressable Memory Hit Notifier
  - 3.3.7 Write-back Pipeline

### 4 Results & Analysis
- 4.1 Math Model Classification Results
- 4.2 Transaction-level Model
  - 4.2.1 Timing Estimates
  - 4.2.2 Logical Memory Mapping
- 4.3 Synthesized Hardware
### 4.3.1 Full Custom Layout & Area Estimates

86

### 4.3.2 Power Estimates

96

## 5 Conclusions & Future Work

98

### 5.1 Conclusions

98

### 5.2 Future Work

100

## A Temporal Memory

109

### A.1 Temporal Memory Algorithm

109

### A.2 Mathematical Model

111

#### A.2.1 Phase I: Active State

112

#### A.2.2 Phase II: Predictive State

114

#### A.2.3 Phase III: Synapse Update

115

### A.3 Hardware Model

118

### A.4 Temporal Memory Prediction

123
# List of Tables

2.1 2010-2014 ITRS technical road map for flash technology compared against the commercially available implementations produced ............................................. 8

2.2 Summary of neuromorphic computing architectures developed and made available through the literature. .............................. 27

3.1 Configuration information stored in the zeroth logical sector .... 53

4.1 The constant parameters selected for the SP model ............... 71

4.2 Per-epoch SP-to-SVM classification results for training and test data .......................................................... 78

A.1 The constant parameters selected for the TM model .............. 125
## List of Figures

2.1 Cross-sectional view of the standard FGMOS transistor . . . 6  
2.2 NAND flash memory trends . . . . . . . . . . . . . . . . . 11  
2.3 PCIe x2 link, comprised of two LVDS pairs . . . . . . . . . 12  
2.4 The architecture of a basic SSD system-on-chip . . . . . . . 13  
2.5 A TSOP package for an asynchronous flash memory package 15  
2.6 The organization of a flash memory array . . . . . . . . . . 16  
2.7 An array of NAND memory is composed of columns of ver-  
   tically aligned strings of FGMOS cells . . . . . . . . . . . . 17  
2.8 SpiNNaker inter-processor communication architecture . . . 20  
2.9 HICANN is comprised of multiple ANC nodes that encap-  
   sulate neural functionality . . . . . . . . . . . . . . . . . . 21  
2.10 TrueNorth is one of the largest scale custom neuromorphic  
    hardware systems currently published . . . . . . . . . . . 22  
2.11 Neurogrid is comprised of a software and hardware layer . . 23  
2.12 HRL Laboratories LLC has been funded by DARPA to de-  
    velop a neuromorphic hardware design . . . . . . . . . . . . 24  
2.13 A parallel multi-core HTM architecture, proposed by Zyarah 26  
2.14 The HTM Zeta node structure, which utilizes Bayesian be-  
    lief propagation . . . . . . . . . . . . . . . . . . . . . . . . 30  
2.15 HTM depicted having 3 hierarchically organized layers, form-  
    ing a tree-shaped structure of nodes . . . . . . . . . . . . . 32  
2.16 An HTM region, consisting of 36 columns, each composed  
    of 4 cells . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.17 The overlap for each column is computed through a three-step process ........................................ 35
2.18 The second phase of the spatial pooling process, namely inhibition .................................................. 37
2.19 The third and final phase of the spatial pooling algorithm—learning .................................................... 38
2.20 The update of SP network parameters ................................. 39

3.1 FHTM high-level architecture concept ................................. 41
3.2 The FHTM TLM, implemented using the SystemC C++ extension ..................................................... 47
3.3 UML sequence diagram for the FHTM TLM design .............. 49
3.4 The logical memory organization for the FHTM architecture 52
3.5 Microarchitecture for FHTM SP ........................................ 56
3.6 Overlap pipeline for FHTM architecture ............................ 57
3.7 Logic used to manage the handshake between overlap engine and the channel arbitration logic ........... 59
3.8 The channel arbiter manages transmitting overlap values from multiple overlap engines to the downstream inhibition engine 60
3.9 The inhibition engine for the FHTM architecture .................. 61
3.10 The inhibition engine control logic, dictating when and how to swap data ...................................... 62
3.11 The content-addressable memory channel arbiter ............... 63
3.12 The fundamental unit that is chained together to form the 
   *WBCam* .................................................................. 64
3.13 The *Camhit* component is comprised of comparators used to determine whether the ID for the $i^{th}$ channel has been matched in the CAM .................................................. 66
3.14 High-level view for the write-back pipeline, *WBPipe* ........ 67
3.15 The duty cycle and boost update pipeline is a three phase ALU governed by the SSD controller to select arguments for the multiplication and accumulation phases.

3.16 The segment update data path portion of WBPipe.

4.1 Change in network parameters over time for SP with number of columns, $K$, equal to 49.

4.2 Change in network parameters over time for SP with number of columns, $K$, equal to 98.

4.3 Change in network parameters over time for SP with number of columns, $K$, equal to 196.

4.4 Change in network parameters over time for SP with number of columns, $K$, equal to 392.

4.5 Change in network parameters over time for SP with number of columns, $K$, equal to 784.

4.6 Classification results for the SP.

4.7 $L_T$ and $L_{LDPC}$ are hidden by pipelining page reads and utilizing a pipelined LDPC unit to 100% availability of the LDPC decode functionality.

4.8 Demonstration of a read operation for an SPU featuring 8 channels, 16-bit wide data bus, and a 5 element deep inhibition queue.

4.9 Maximum number of synapses possible for HTM network, as a function of segment size.

4.10 Synthesis process followed to generate hardware results for the FHTM design.

4.11 Synthesized hardware implementation of the overlap pipeline with an 8-bit wide data bus.
4.12 Synthesized hardware implementation of an 8-channel fixed priority arbiter with an 8-bit wide data bus and integrated multiplier ......................................................... 88

4.13 Synthesized hardware implementation of a 5-element deep inhibition engine with an 8-bit wide data bus ......................... 90

4.14 Synthesized hardware implementation of an 8-channel with 8-bit wide data bus ................................................................. 91

4.15 Synthesized hardware implementation of an 5-element WB-Cam with 8-bit wide data bus ................................................. 92

4.16 Synthesized hardware implementation of a 5-element, 8-channel Camhit component with 8-bit wide data bus ............. 93

4.17 Synthesized hardware implementation of a WBPipe with 8-bit wide data bus ................................................................. 94

4.18 Synthesized hardware implementation of a WBCntl, comprised of Camharb (bottom-right), WBCam (left), and Camhit (top-right), an with 8-bit wide data bus ................................. 95

4.19 Area and dimensions for each of the synthesized RTL components .......................................................................................... 96

4.20 Average power consumed for each of the synthesized RTL components .................................................................................. 97

A.1 Temporal memory phase 1—infrence and learning ................. 110

A.2 The second phase of the temporal learning process, namely the predictive state calculation .................................................... 111

A.3 The third and final phase of the temporal learning process—the commit phase for the updates staged in the prior two phases ................................................................. 112

A.4 The first task of TM is to ascertain the subset of cells that will enter the active predictive state ............................................. 119
A.5 Phase one of the segment activation calculation ............... 120
A.6 The predictive state update and final step of the gamma calc-
culation ............................................................. 121
A.7 After all of the segment activation are calculated, the best
candidate cells for learning are selected ............................. 122
A.8 Learning is concluded by writing the distal segments back
to memory in a fashion similar to that of SP ...................... 123
Chapter 1

Introduction

1.1 Motivation

Information is being created in quantities orders of magnitude larger than it has been in the past [1, 2]. These data sources are being mined to extract information that may guide decision making to support monetization; thus, insight is made into a product. This is possible, because large systems of data are assumed to be characterized by complex, often unknown, underlying statistical models. As engineering design problems are becoming more difficult to solve, finding more robust methods to characterize these systems becomes increasingly necessary. In many cases, approximating these models leads to insights that provide solutions to the most challenging problems in modernity. State-of-the-art research acknowledges the utility of data and the implications of being able to understand it.

Machine intelligence provides a promising solution to these problems by creating complex models of large sets of unstructured data. Learning systems are able to solve general pattern recognition with state-of-the-art accuracies. Many varieties of algorithms have been explored in the literature, providing varying results. Scientists in the field of machine intelligence have stated that performance in modern learning systems is significantly influenced by the scale of the model. For neural networks—a class of machine learning systems inspired by cortical functionality—larger network sizes significantly improve classification performance. However, training large networks is non-trivial due to issues such as the vanishing gradient problem—a
reality that is especially challenging for both deep and recurrent neural networks (DNN and RNN) [3, 4, 5]. This makes back-propagation, a popular algorithm in the prior art, an impractical method for training large networks. Alternate techniques such as deep learning, and Hierarchical Temporal Memory (HTM) seek to utilize different learning methodologies to make the training procedure for large-scale networks feasible.

Modern neural networks have been scaled to be comprised of billions of parameters, requiring high performance computing clusters and weeks to solve large problems [6]. Another challenge to scaling is that the memory infrastructure serves as a profound bottleneck. A common trend has been to design hardware architectures that emulate the functionality of these algorithms to improve the scale of the solution. Neuromorphic computing, a subset of these hardware architectures, seeks to utilize abstractions of cortical operating principles to achieve the scalability of the brain; albeit, all computing finds its original inspiration in abstractions of cortical functionality.

Considered as the last frontier of biology, the brain has perplexed researchers for millenia [7]. Over the ages, contributions to the concept of how the brain functions were proposed by researchers from various disciplines, from philosophy to chemistry, biology and even physics. To normalize language and concepts in this field, many research disciplines were compacted into the term neuroscience. This consolidation made it possible for the common goal of understanding the structure and function of the normal and abnormal brain more feasible. Neuromorphic computing focuses on modeling the underlying functional capabilities of the normal mammalian brain and subsequently apply them to real-world applications.

The human brain is comprised of billions of neurons and trillions of synapses—each neuron being connected to thousands of other neurons through synapses. High-level neural realities such as cognition, attention and memory are believed to exist due to low-level neural computations. These computations altogether consume power on the order of tens of watts—several orders of magnitude less than current hardware systems designed to abstract these underlying principles. Significant research effort has been directed
toward understanding the brain and subsequently designing systems to abstract these principles with the aim of achieving similar scalability. The thesis of modern neuromorphic computing is that if key principles of brain functionality can be emulated in a hardware platform, we will be capable of solving a subset of the challenging future engineering problems by extracting non-intuitive patterns in complex data. Acquiring solutions to this problem has the potential to yield electronic systems that autonomously extract useful meaning from large repositories of data, at a low power profile.

It is the aim of this thesis to present a scalable hardware implementation of the HTM cortical learning algorithm (CLA) spatial pooler (SP). This architecture, called Flash HTM (FHTM), leverages large-scale solid state flash memory architecture to target pattern recognition applications. FHTM was tested on the MNIST dataset and subsequently evaluated in terms of potential scalability.

1.2 Contributions

This thesis investigates the implementation of a scalable flash-based HTM CLA storage processor unit (SPU). In particular, the following contributions are made:

1. A Transaction-level model (TLM) of the SPU is proposed, implemented in SystemC, capable of being customized to model system-level functionality

2. A robust configurable register transfer level (RTL) vector processor is implemented to estimate area, and power

3. The proposed architecture was evaluated against MNIST using a linear support vector machine (SVM) classifier

4. An analysis that describes the benefits and challenges of implementing a custom PCIe flash-based SPU
1.3 Outline

The remainder of this thesis is organized as follows:

Chapter 2 provides background information concerning flash technology and the HTM CLA. A summary of the two versions of the algorithm in addition to the literature containing hardware implementations thereof. Related work regarding other neuromorphic hardware designs are also discussed.

In Chapter 3, a new hardware architecture is presented—i.e. FHTM. Microarchitectural details, such as the memory organization, implementation of the channel vector processor are described. FHTM was implemented as a mathematical model, TLM and synthesized RTL.

In Chapter 4, simulation results for the mathematical model are presented. FHTM was synthesized and laid out to acquire requisite area; power data was extracted from the layout-based parasitic netlist.

Chapter 5 discusses the conclusions and potential areas for future work. Emphasis is placed upon implications to future flash-based SPU architectures. Preliminary work regarding the temporal pooler model have been captured in the Appendix.
Chapter 2

Background & Related Work

2.1 Flash Memory Technology

The invention of flash technology has led to many innovations in consumer and enterprise electronics applications. Originally invented by Fujio Masuoka in 1984, the flash cell (Fig. 2.1) has undergone many technological improvements, resulting in what today is considered a very robust device [8, 9]. NAND topology flash has risen as the design of choice in large scale memory systems for the foreseeable future. Flash is a non-volatile memory (NVM) that exploits quantum mechanical and thermodynamic properties of semiconductors to store charge without the need of a relatively short periodic refresh cycle. Carriers are injected or removed from the floating node by utilizing Fowler Nordheim Tunneling (FNT) or Channel-Hot Carrier Injection (CHCI) [10, 11]. FNT, also known as cold field emission, is defined as

\[
J_{\text{field}} = \beta_{\phi} \mathcal{E}^2 e^{\frac{E}{\alpha}}
\] (2.1.1)

where \( J_{\text{field}} \) is the probabilistic charge density that flows through the oxide barrier into the floating node as a result of the presence of a large magnitude electric field; \( \beta_{\phi} \) is a variable dependent upon the charge of the carrier, its mass, the plank’s constant, the mass of the oxide, and the barrier height; \( \mathcal{E} \) is the uniform electric field; and \( \alpha \) depends upon a different relationship between the same parameters that influence \( \beta_{\phi} \). This equation assumes that thermionic effects are negligible in comparison to the impact of the induced field on the charge density—an assumption that requires that
Figure 2.1: (a) Cross-sectional view of the standard FGMOS transistor. (b) The capacitive coupling model is described as a MOSFET with capacitively coupled inputs. A high-valued resistance, on the order of Giga-Ohms, is connected to a sum of DC voltages to represent a steady state voltage for DC analysis.

The semiconductor be at lower temperatures, around room level [12]. FNT is derived by solving the time-independent Schrödinger equation (TISE).

CHCI, unlike FNT, is more directly a kinetic effect. As charged particles inside of the semiconductor gain energy, they probabilistically collide with other particles that are within the channel lattice. Some impacted carriers are deflected upwards into the oxide. If these particles have enough energy, they will either be injected into the oxide, or tunnel all the way into the conducting floating node. A more complete discussion and explanation of injection mechanisms as well as impact ionization are provided in [11]. For circuit modeling purposes, the natural equations for CHCI are not intuitive, as they are not defined in terms of simple voltages, and currents. Consequently, fitting models have been presented in the literature that fit the form

\[ I_{CHE} = I_{sub} \alpha_{ox} e^{\frac{b_{ox}}{E_{ox}}} \]  
(2.2.1)

\[ I_{sub} = I_{DS} \frac{a_i}{b_i} V_{sat} e^{-\frac{b_i}{V_{sat}}} \]  
(2.2.2)
where $I_{sub}$ is the substrate current; $\alpha_{ox}$ and $b_{ox}$ are fitting parameters; $E_{ox}$ is the electric field across the tunneling dielectric; $I_{DS}$ is the drain-to-source current; $a_i$ and $b_i$ are the impact ionization coefficients; and $V_{sat}$ is the saturation potential [13].

These mechanisms that make flash technology feasible also happen to present the most profound challenges to future scaling. The injection of charge into the oxide lattice has a degenerate effect upon the structure thereof. In the most common case, this results in threshold voltage shift, but in the worst case it may result in the formation of a conductive path between the charge storage layer and the drain-to-source channel.

### 2.1.1 Industry Implementations

Recent road maps, as shown in Table 2.1, indicate that manufacturers of cellular flash have improved its capabilities by: (1) reducing the feature sizes down to the 1xnm scale; (2) increasing the number of bits per cell; (3) utilizing more effective oxide materials (high-K dielectrics); (4) employing novel cell programming methodologies; (5) using more robust semiconductor materials, namely charge-trapping flash (CTF); (6) and expanding the memory array to the third dimension to overcome two-dimensional scaling limitations.

There are four organizations that produce the majority of commercially available flash cells, namely Samsung, Micron (partnered with Intel), Toshiba (Sandisk), and Hynix. Many more companies purchase these flash chips and integrate them into higher-level consumer products. Mobile devices and solid state memory are common applications for flash produced by the aforementioned companies. Semiconductor memory manufacturers implement innovative flash memories by attempting to address key design concerns. The primary concern that drives innovation in this space is the cost-per-bit for a memory module. Other latent parameters such as the robustness of the device and power dissipation are indirectly influenced by the desired cost-per-bit. As the ability for flash cells to withstand more program cycles increases, the maintenance cost of using them also decreases,
Table 2.1: 2010-2014 ITRS technical road map for flash technology compared against the commercially available implementations produced, separated by company. Despite the brevity of this list, more companies exist that are directly or indirectly invested in the semiconductor flash memory market such as: AMD, Atmel, Fujitsu Microelectronics, ISSI, Macronix International Co. Ltd, Microsemi, On Semiconductor, Sharp Electronic Corp, SST, Spansion, and STMicroelectronics. The organizations included in this table are industry leaders in generic flash storage as presented in [14, 15].

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>2010</th>
<th>2011</th>
<th>2012</th>
<th>2013</th>
<th>2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITRS Flash Road map</td>
<td>32 nm</td>
<td>22 nm</td>
<td>20 nm</td>
<td>18 nm</td>
<td>16 nm</td>
</tr>
<tr>
<td>Samsung</td>
<td>35 nm</td>
<td>27 nm</td>
<td>21 nm (MLC, TLC)</td>
<td>16 nm (eMMC 4.5)</td>
<td>16 nm (3D: TCAT)</td>
</tr>
<tr>
<td>Micron Intel</td>
<td>34 nm</td>
<td>20 nm (MLC, HKMG)</td>
<td>20 nm (TLC)</td>
<td>18 nm</td>
<td>16 nm</td>
</tr>
<tr>
<td>Toshiba Sandisk</td>
<td>43 nm</td>
<td>24 nm</td>
<td>19 nm (MLC, TLC)</td>
<td>2D: 1Y nm</td>
<td>2D: 1Z nm</td>
</tr>
<tr>
<td>SK Hynix</td>
<td>46 nm</td>
<td>20 nm (MLC)</td>
<td>20 nm (MLC)</td>
<td>16 nm</td>
<td></td>
</tr>
</tbody>
</table>

because devices will be replaced less frequently. The feasibility of implementing flash in large scale systems–server farms and high performance computing–is tremendously impacted by the cost-per-bit. A specification has been created by IEEE that provides recommendations for flash memory systems [16].

One of the largest monolithic planar flash memory modules was fabricated by Micron engineers–namely, a 128Gb MLC floating gate (FG) NAND flash-based chip [17]. CTF, an alternative to FGMOS, has further extended out the potential scalability of NAND by enabling 3D technology. Spansion is an example of a company that has produced CTF in multiple feature sizes, such as in the 45 nm and 32 nm technology nodes [18]. Three-dimensional flash topologies have been proposed as early as the beginning of the 21st century. As feature sizes scale down to the 1 x nm scale, quantum mechanical effects impact the device behavior in more profound ways. Inter-cellular interference, a major design limitation, still presents a profound challenge to scaling; disturbance and interference are less significant in CTF than in FGMOS. Photolithographic patterning techniques are also
prohibitive at these scales [19, 20].

Today, 3D devices are considered necessary to continue flash scaling trends. In [21], Park et al. fabricated a two-layer 45nm three-dimensional memory integrated chip. Design concerns due to scaling up were addressed, namely variations in the threshold voltage ($V_t$) distributions of the two layers, program mechanisms, and power saving methodologies. This work was a part of research efforts by Samsung, which eventually lead to the creation of V-NAND™ [19, 20, 22, 23]. Second generation V-NAND structures at Samsung have been scaled to be composed of 48 layers, with better performance and endurance characteristics when compared to planar NAND. Some other notable 3D NAND implementations are as follows:

1. **Pipe-shaped bit cost scalable (P-BiCS)** is a 60nm SONOS-type memory with 16 vertically-stacked layers developed by the Toshiba Corporation. It features a u-shaped pipe structure used to electrically connect two vertical NAND strings. Write and erase voltage magnitudes were on the order of 20V [24]

2. **Dual control-gate with surrounding floating gate (DC-SF) NAND** utilizes relatively lower voltage per cell for read and write operations; 15V for program and -11V for erasure [25]

3. **Virtual-gate (VG) architecture**, developed by researchers from Macronix International, utilizes a TANOS cell structure and requires a $5F^2$ cell size. The creators asserted that the cell has potential to reach $2X nm$ minimum feature size [26]

Another interesting development in the semiconductor memory industry is the introduction of 3D XPoint™ memory through the joint efforts of Intel and Micron; although 3D XPoint is not flash-based. It was recently made generally available as a novel memory architecture that enables individual bit-accessibility across a two-layer array of resistive cells. Currently, the physical details of the technology have not been made public. All that has been made public is that this technology cannot be characterized as a flash device and is expected to yield performance several orders of magnitude above that of flash technology. Furthermore, companies such as Crossbar
Inc seek to develop practical implementations of resistive RAM (ReRAM). Despite these developments, the more than 15 billion dollar flash marker is still expected to coexist alongside these technologies for the foreseeable future.

2.1.2 Flash Cell Scaling

Flash-based solid state memories continue to yield some of the largest densities for NVMs, falling shy of magnetic technologies [27]. Memory systems scale based upon several parameters–namely, area, performance, bits/cell, power consumption, and endurance. Reduction in the total area consumed per cell has been a primary source of improvement in the memory industry and the semiconductor space as a whole. Footprint shrinking has led to the major achievement of a 128Gb MLC [17] at the 16nm technology node. Further scaling of devices will continue to be challenging due to degenerate quantum mechanical effects, which have led to a whole host of issues. Some important concerns include:

1. Higher rate of device mismatch due to increased variations
2. Growing static power dissipation from increasing leakage currents
3. Lower oxide endurance leading to quicker breakdown of the oxide lattice due to read/write stress
4. Limited threshold voltage scalability
5. Inter-device program and erase disturbances

Operational speeds for flash are several orders of magnitude less than the next best performing technology (DRAM), but with the benefit of lower cost-per-bit and reduced power consumption. Most types of memory are expected to coexist in the market for the foreseeable future, because of the balanced trade-off between strengths and limitations of each technology. In light of this, flash technology is expected to grow drastically in the wake of present data growth predictions released by primary industry leaders. Performance has improved significantly for flash cells from FGMOS to CTF
Figure 2.2: (a) NAND flash memory trends, indicating that flash technology has scaled well in the past decade, surpassing Moore’s law, due to the introduction of multi-level cell technology up to 4bits/cell [27, 28]. (b) The memory capacity trend for emerging NVMs indicate that NAND flash, being quite mature, is ahead of the other technologies. This information was compiled from ISSCC, JEDEC, and TechInsights [27].

and now to 3D technologies, which boast increased endurance and density without trading off performance (when compared to the planar cell technologies).

The original single-level cells (SLC) are no longer as cost-effective now that the multi-level cell (MLC) technology has matured. Scaling bits-per-cell beyond MLC has proven challenging as it reduces device reliability and overall endurance. In addition to this, bit-per-cell scaling requires more complex programming mechanisms and longer read latencies. Despite these challenges, there are still TLC (3bits/cell) and 16LC (4bits/cell) implementations, albeit with lower endurance than MLC realizations. Trends for NAND flash from recent years are depicted in Fig. 2.2. Currently, both planar and 3D devices are on the road maps of most semiconductor flash manufacturers, because the planar device manufacturing processes are more mature than that of the 3D varieties and cost less to manufacture.

NAND technologies are more prevalent in the market due to their fast sequential read and write speeds. However, implementations of NOR-type flash boast improvements in power consumption for random accesses due to greater bit access granularity. This variety in topology makes flash amenable
Figure 2.3: PCIe is the interface of choice for the state-of-the-art high-speed storage technology. At the physical layer, there are multi-directional lanes comprised of two low voltage differential signaling pairs [29, 30].

to a multitude of application domains from mobile to enterprise. Power profiles for flash have been limited by quantum effects; as dynamic power is reduced by scaling other parameters, static power tends to increase. In light of this, power dissipation has improved through novel cell topologies as well as creative programming and erasure schemes. Future performance scaling of flash is expected to be significantly dependent upon novel programming and erasure methodologies.

2.1.3 Large-Scale Memory Systems

Flash-based solid-state drive (SSD) technology will significantly enhance the performance of enterprise storage systems, because it has orders of magnitude shorter access-latency and larger bandwidth compared to mechanical disk technology. NAND technology has matured to the point where cost-per-bit is significantly lower than in prior years, making it more feasible for large-scale integration. Prior SSDs have also saturated bandwidth of the host interfaces that they have been connected to, namely SATA. This provides a future trajectory for storage performance scaling. To utilize the improved throughput, alternate host interfaces on the SSD ASIC have been explored, i.e. PCI Express (PCIe), depicted in Fig. 2.3. Drives utilizing
Figure 2.4: The architecture of a basic SSD system. Modern devices have on the order of 10s of channels. Each channel is shared by up to eight flash packages that have unique enable/disable signals to facilitate higher read and write bandwidths. In modern embodiments, DRAM is used to cache data, improving memory lifetime by reducing write stress on the NAND array.

This interface are capable of transferring data on the order of gigabytes-per-second; older generation SATA SSDs operated on the order of hundreds of megabytes-per-second.

SSDs store information inside of individual NAND-type packages. These packages are developed by the companies described in the prior sections. An SoC is placed on the SSD printed circuit board (PCB) and manages read and write access to these industry-standard flash packages (Fig. 2.4). NAND packages today are characterized by one of three standard NAND interfaces: (1) classic asynchronous interface; (2) Open NAND Flash Interface (ONFI); and (3) Toggle-Mode interface. The asynchronous interface is depicted in Fig. 2.5. These interfaces all share much in common, the details of which are presented in great detail in the literature [31, 32].

Non-volatile memory express (NVMe) is a modern register-level PCIe-supported standard created to enhance interoperability and device utilization.
over the host interface in future PCIe SSDs. Prior to this, the Advanced Host Controller Interface (AHCI) was used, but it was designed for much slower memory devices and resulted in host-to-SSD intercommunication inefficiency. Capacitors are used as a low-cost emergency battery on the SSD printed circuit board to mitigate data loss due to power instability; the DRAM is allotted time to write pending data back to the NAND array before full power-down [32].

Modern flash controllers are quite complex, being composed of the host interface, flash file system (FFS), error-correcting code (ECC), and the flash interface (Fig. 2.4). Typically, the memory controller is comprised of approximately eight individual processors—typically ARM-based. A subset of these cores implement the host interface protocol (e.g. PCIe), guaranteeing interoperability at the electrical and logical levels between the host and the flash components. Through this interface, the inner workings of the NAND array are effectively hidden from the host processor.

For instance, if the host must interact with the NAND array over a PCIe medium to read one block of memory, it will first transmit a command via the aforementioned protocol implemented by the host interface. Thereafter, the memory controller will decode this command into the proper control signals needed by an off-the-shelf NAND component (Fig. 2.5). Furthermore, a specific flash package will be selected to be read from. These controllers will also hide read and write latencies by utilizing in-channel pipelining and across-channel parallelism.

Media processing concerns are also handled by the memory controller—namely, wear leveling, garbage collection, and bad block management. Because the endurance of flash technology is a primary design limitation, the FFS evenly distributes the stress of program and erasure across the NAND array. In practice, this requires address translation between the physical and logical memory spaces, the latter of which is seen by the host. Translation tables used by the SSD are backed up in the NAND array, which results in a nominal reduction of the available storage capacity visible to the host. The translation process and the other operations that create a logical file system optimized for flash are a part of the flash translation layer (FTL).

NAND flash arrays operate in the row-column memory space, requiring a
Figure 2.5: A TSOP package for an asynchronous flash memory package. NAND flash packages are constrained to a simplified shared read/write interface to reduce I/O complexity. The data bits for commands, addressing, and data (read and write) are all passed via a bi-directional port ($D_Q$). Control signals determine the significance of data bits that are currently on the flash data port. To facilitate the ability for multiple memory elements to be connected onto a shared channel interface, an enable signal is used ($CE#$) as well as an open-drain status signal ($R/B#$). This interface is clocked via the $WE#$ signal, but the other, more modern, interfaces utilize an explicit clock to obtain greater performance [31, 32].

mapping from the physical page to a row and column address. The memory organization of an off-the-shelf flash package is depicted in Fig 2.6; a more detailed look at the structure is provided in Fig. 2.7. For brevity, additional detail concerning the device is not depicted here, such as the sense amplifier, inner-workings of the decoder, and logical-to-physical translation.

An SSD acts as a slave component to the host, receiving commands through the host interface. Input data from the host are used to generate command sequences to one or multiple flash channels by way of a channel controller. This controller is used to pipeline and interleave individual memory accesses. Each channel is shared by around 8 flash packages, and tends to come in $D_Q$ widths of 8x, 16x or 32x bit-widths. Individual chip enables qualify commands and data to facilitate the pipelining process and parallelization [33].

For instance, to mask the majority of read latency, each flash package on a channel, save for the first, would be disabled. A command sequence is transmitted to the first flash package that is enabled, then the controller disables the element’s interface. Subsequent cycles are used to transmit
Figure 2.6: The organization of a flash memory array. Each NAND array is subdivided into blocks, and these in-turn are composed of pages. A page is comprised of all cells that are in the same row in the array (i.e. connected to the same word line). Programming and reading of the memory occurs at the page level. Blocks are a logical division of the memory that represents the smallest amount of cells that may be erased. Spare storage space is hidden from the host and is used by the FTL/FFS.

commands to the second package–this process continues until the last flash element connected to the channel receives a command from the memory controller. After the latency, $t_R$, for the first flash package is completed, data should begin to become available at each cycle. After the first page of data is received the remaining data will become available after each cycle–instead of after the significantly longer access latency, $t_R$. The channel controller also implements ECC using LDPC codes [34]. Translation from encoded to corrected data is handled by this dedicated processing element.

2.1.4 In-Memory Compute

Process technology has continually scaled down to the $1xnm$ scale, resulting in a larger number of available transistors. These excess transistors may be utilized to incorporate processing elements into the SSD controller that
operate on data being read from or written to the NAND array. This has the potential to reduce memory access latencies—the major system bottleneck. Specialized memory architectures that take this into account may be sought after by large enterprises. Key applications that are being targeted through these innovations include data analytics and pattern recognition. Netezza, now a subsidiary of IBM, is a prime example of this—their primary product line includes tools to handle the processing of queried data close to the memory. Currently, their model is to utilize FPGAs paired with CPUs to manage a cluster of memory elements.

It is understood that there exists a memory wall; the host processor is
capable of saturating the bandwidth of all attached memory components. Close-to-core storage, found in the register file and the cache, is fast, but volatile and cost prohibitive to implement in large quantities. Hard disk drives are significantly more cost-effective, but are orders of magnitude slower with a large power footprint. In the former case, the host processor is able to saturate the memory units simply because they have a limited amount of data to be processed due to size limitations (limited volume); the latter case is constrained in that it is unable to supply information to an attached processor fast enough (limited bandwidth). A consequence of under supplying the CPU with data is that a large number of clock cycles are spent in an idle state, waiting for memory to be delivered.

Although flash technology offers significant bandwidth improvements over HDDs, it still is not capable of supplying the demand of a multi-core processor. For applications in which the workload is well understood, handling a portion of the processing on the SSD ASIC may be a more cost-effective solution to the memory bandwidth problem in the long-term. Placing common compute workload functions close to the memory is expected to reduce the required link bandwidth between the processor and the storage element [35]. Significant power savings are also obtained through this paradigm, because fewer data transmissions are made over the long communication interconnects.

To make such computations more feasible, they may operate at the DRAM level. The notion of an SPU was presented in [36]. This system embeds an additional co-processor into each channel along with the ECC. A summary of the known implementations of in-memory compute were summarized in the literature [37]. ActiveFlash was an implementation of a data analytics SPU on an SSD using ARM cores [38]. Access to the computing resources within the controller would be shared using a scheduling algorithm that divided time between commands received from the host, garbage collector, and in-memory compute operations.
2.2 Neuromorphic Hardware Architectures

The term “Neuromorphic Computing” was originally coined by Caver Mead in 1990 [39]. Since then, profound design effort has been directed toward developing a novel neuromorphic hardware architecture with a low power profile and small area footprint. These efforts have been motivated and funded in large part by the DARPA Systems of Neuromorphic Adaptive Plastic Scalable Electronics (SyNAPSE) project and the Human Brain Project (HBP). The SyNAPSE project has provided over $100 million in funding to IBM, HRL Laboratories, and smaller subcontracted groups. HBP, initiated in 2013 and funded by the European Union, is a collaboration between over one-hundred research groups with the aim of achieving a multi-level, integrated understanding of brain structure and function.

Aside from the major sources of funding, academia as well as other industry partners have placed large emphasis on neuromorphic computing. A complex challenge, emulating brain functionality continues to be a multi-disciplinary design effort. The neuroscience community is expanding the knowledge base of cortical operating principles; machine intelligence specialists are developing the algorithms that model that functionality; neuromorphic hardware engineers design the architectures that recreate the capabilities of these algorithms. The remainder of this chapter focuses on highlighting existing neuromorphic hardware architectures.

2.2.1 SpiNNaker

In 2010, researchers at the Manchester University presented a hardware neural network simulator called SpiNNaker (Fig. 2.8)–the Spiking Neural Network architecture. It is an 18-core System-on-Chip (SoC) capable of simulating 18k spiking neurons, yielding 3.96GIPs (Giga Instructions per Second) [40]. Multicast communication is used to transmit source-routed packet information between each neural processor through six physical links–namely, North, South, East, West, Northeast, and Southwest. ARM968 chips serve as the processing elements, which transmit data in an asynchronous fashion–creating a low-power chip multiprocessor (CMP).
Figure 2.8: SpiNNaker inter-processor communication architecture. As with most CMPs, a network-layer interconnect element, i.e. a router, is used to enable interprocess communication. A network interface module (NIM) abstracts the networking protocols from the processing, so the processor does not have to specifically handle intercommunication details. The receiver and transmitter links are physical connections to the remainder of the network [40]. Each individual core is locally synchronous and the network layer is asynchronous, which improves power efficiency.

Because general purpose processors (GPP) are used, neuron models of varied complexities may be implemented—the thalamocortical model developed by Eugene Izhikevich is typically utilized. To address the issue of scalability, the SpiNNaker CMP can be connected to other CMP via an ethernet interface—enabling scaling up to 65,536 chips. In the full-size embodiment, SpiNNaker is expected to be comprised of 57,600 chips, with 1,036,800 processors, operating at 90kW. A data sheet was provided to accompany the architecture [41].
2.2.2 HICANN Analog Network Core

The High Input Count Analog Neural Network (HICANN) is the mixed-signal primary building block for the Fast Analog Computing with Emerging Transient States (FACETS) wafer-scale system (Fig. 2.9) [42]. Each chip consists of 8 HICANNs, comprised of an analog neural core (ANC) and supporting circuitry. A chip can implement 128,000 synapses with a controllable pre-synaptic input count. This was developed as a part of the BrainScaleS project—a collaborative effort between 13 research groups. HICANN is believed to be a promising platform for studying Hebbian learning, spike-timing dependent plasticity (STDP), and cortical dynamics. HICANN is unique from most other neuromorphic hardware in that its application is primarily geared toward understanding neurobiology instead of utilizing it for applications. Unlike the other models it is also capable of representing 10,000 synaptic connections per neuron, which reaches biological levels. Both SpiNNaker and HICANN fall under the Human Brain Project.
Figure 2.10: TrueNorth is one of the largest scale custom neuromorphic hardware systems currently published. This diagram depicts a rough overview of the layout of a single TrueNorth core and its associated local memory. Each core is capable of modeling 256 neurons, 256 axons and also 256x256 synaptic connections. Neuronal elements communicate to axons using 32-bit-wide spike packets, which traverse the network using XY routing. To allow for the system to map to the same functionality as the software model, namely Compass, the hardware architecture has been modified to operate in a globally synchronized fashion via a synchronization clock [43].

(HBP).

2.2.3 IBM TrueNorth

TrueNorth is a 4,096-core non-von Neumann neuromorphic network on chip (NoC) designed as a part of the IBM SyNAPSE project [44]. One chip can support 1 million programmable neurons and 256 million programmable synapses. Manufactured in 28nm technology, it is the largest neuromorphic chip architecture created to date, utilizing an integrate-and-fire spiking neuron model. The TrueNorth project is a collaborative work between IBM Research and Cornell University; it is funded by the DARPA SyNAPSE project. A simulation environment and programming language were designed to facilitate hierarchical architecture design—namely, the Compass simulation environment [45] and the Corelet programming language [46].
Figure 2.11: Neurogrid is comprised of a software and hardware layer. A given design is configurable and capable of being modified to suit various applications. Each node contains one Neurocore, which is constituted of a set of neurons and synapses. Interaction between Neurocores is realized by intercommunication on a packet-switched network.

The current system acts more as a reconfigurable spiking network simulator, because it does not feature on-chip learning. Several applications have been tested on the architecture—namely, speaker recognition, composer recognition, digit recognition, hidden markov modeling (HMM), and collision avoidance [43, 47].

Although TrueNorth was presented as recently as 2014 (the first iteration was made public in 2011), the IBM SyNAPSE team worked on this engineering design problem for nearly the past decade. A major criticism of TrueNorth is that software implementations of integrate-and-fire neural networks have not been proven to provide state-of-the-art performance in the literature. Regarding the structure of the architecture, each core is composed of a memory block, controller, scheduler, neuron parameter block, and router (Fig. 2.10).
Figure 2.12: HRL Laboratories LLC has been funded by DARPA to develop a neuromorphic hardware design with the intent to eventually fabricate it. The architecture is discussed in the work of Srinivasa [48]. Processing cores are connected in a mesh-like topology with minimal switching overhead. The wires connected between the neuron and the synapses are actually split into two isolated channels—one for the input spikes and the other for the feedback and outputs.

2.2.4 Neurogrid Hardware Accelerator

Stanford engineers have developed a reconfigurable neuromorphic hardware architecture called Neurogrid, depicted in Fig. 2.11. Capable of modeling 1 million neurons in real-time, it is targeted at real-time neuronal system modeling for neuroscience research, and real-world applications. Neurogrid utilizes a dimensionless spiking neuron model for the various aspects of the cortical-inspired hardware design. This architecture is comprised of neuron, transmitter, receiver and router modules. Information is routed throughout a tree-shaped network to minimize transmission latencies. To reduce design footprint, various neural elements share some logic in common with each other—a decision that increased the amount of possible synaptic connections. Neurogrid is a mixed-signal design [49].

2.2.5 HRL Neuromorphic Chip

Hughes Research Laboratories (HRL) has been in the process of developing a neuromorphic chip as a part of the DARPA SyNAPSE initiative [48]. This
chip (Fig. 2.12) has been designed to use spike-timing-dependent plasticity (STDP) and was physically manufactured in 2013. Being constrained by the same requirements as IBM, HRL seeks to create a complete ecosystem to enable brain-scale neuromorphic hardware architectures. The proposed architecture is composed of three components—a reconfigurable front end, an analog core, and analog memory.

2.2.6 Hierarchical Temporal Memory Hardware

A novel model for cortical computation was proposed by Numenta in [50], called Hierarchical Temporal Memory (HTM). Several efforts to model the algorithm in hardware have been presented. Most of the prior art in neuromorphic computing utilizes a spiking neuron model to describe neuronal behavior. However, HTM presents a methodology for abstracting synaptic computation into a higher-level framework, with the goal of optimizing the algorithm for real-world applications. Literature containing hardware implementations of the HTM CLA is very limited—a reality likely caused by volatility in the algorithmic formulation, which now being open source, is subject to regular changes.

A notable example of HTM research was presented by Price [52]. In his work he developed a software implementation of CLA in C++ for multi-core architectures using OpenMP. A speedup factor of 3 was achieved over a baseline, leading to the conclusion that algorithmic adjustments as well as more aggressive optimization would be necessary to improve the scalability of the algorithm. Phases I and II of the temporal memory process was determined to be a great candidate for acceleration. Price only attempted to speedup a portion of the algorithm, which led to these realizations in addition to nominal speedup results.

A C-based many-core implementation of HTM was also developed by Zhou and Luo [53]. The Adapteva Epiphany hardware platform was employed as the architecture of choice for accelerating the algorithm. Using a single-core CPU as the baseline, a case was made for the usefulness of the selected hardware platform for HTM modeling. Adapteva Epiphany is a 2D mesh network of 16 cores, each with 32kB of memory. Zhou and Lou
Figure 2.13: A parallel multi-core HTM architecture, proposed by Zyarah [51]. This design utilized a mesh topology with centralized control. All interaction between the cores is managed by the MCU.

...added that future exploration of multilayer HTM would also be of interest. Furthermore, parallelization was only implemented at the columnar level, which could be improved upon in future designs.

Vyas and Zaveri presented a verilog implementation of a portion of the design, namely the spatial pooler [54]. This design was presented as a small-scale example, implemented on the Xilinx Spartan-3e FPGA platform. In the thesis of Zyarah, a hardware architecture that modeled both the spatial pooler and temporal memory was presented [51]. This is one of the more comprehensive efforts in the literature to model the HTM CLA. An analog model of HTM was created by a team at Purdue that utilized spin-neurons for synaptic connections [55]. However, this employed the first generation of the HTM algorithm, namely Zeta. Each of the aforementioned hardware architectures discussed in this chapter are summarized in Table 2.2.
Table 2.2: Summary of neuromorphic computing architectures developed and made available through the literature.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Made</th>
<th>Neurons</th>
<th>Synapses</th>
<th>Model</th>
<th>Power (mW/cm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpiNNaker</td>
<td>2010</td>
<td>16k</td>
<td>16M</td>
<td>STDP</td>
<td>1,000</td>
</tr>
<tr>
<td>HiCANN</td>
<td>2010</td>
<td>512</td>
<td>144k</td>
<td>AdEXP</td>
<td>3,000</td>
</tr>
<tr>
<td>TrueNorth</td>
<td>2014</td>
<td>1M</td>
<td>256M</td>
<td>LIF</td>
<td>20</td>
</tr>
<tr>
<td>Neurogrid</td>
<td>2014</td>
<td>65k</td>
<td>375k</td>
<td>LIF</td>
<td>50</td>
</tr>
<tr>
<td>HRL NC</td>
<td>2012</td>
<td>576</td>
<td>73k</td>
<td>STDP</td>
<td>120</td>
</tr>
<tr>
<td>HTM</td>
<td>2015</td>
<td>100</td>
<td>≈1600</td>
<td>CLA</td>
<td>N/A</td>
</tr>
</tbody>
</table>

2.3 Hierarchical Temporal Memory

Hierarchical Temporal Memory (HTM) is a biomimetic machine learning algorithm, constructed with the aim of capturing key functional properties of the mammalian brain to solve pattern recognition problems. Created through the joint efforts of Jeffrey Hawkins and Dileep George, HTM was intended to address key challenges within the state-of-the-art in the field of machine learning [56, 57]. They sought to address the lack of a truly biologically inspired framework that also modeled temporal associations in an invariant, efficient manner. Furthermore, they desired to use a connectionist approach to machine intelligence to develop a common algorithm that models the functionality of cortical microcircuits.

Jeffrey Hawkins, in *On Intelligence* [58], discussed the necessity for a new framework upon which learning systems should be defined. Key challenges of the prior art were presented, specifically in the fields of artificial intelligence and neural networks. The work was introduced with seven specific questions—the most important of which being, “what is intelligence if it isn’t defined by behavior?” and more fundamentally, “how does the brain work?” His claims are summarized as follows:

1. The prior art in neural networks and artificial intelligence did not live up to expectations, because the assumption was made that intelligence was defined by behavior—Hawkins states that the cornerstone of intelligent systems is the ability to make predictions about the future

2. A novel framework that would innovate the field would necessarily be
built upon the operating principles of the neocortex, which does exhibit intelligence

3. Such a framework would model the canonical neural microcircuit

From these claims, Hawkins then presented a brief history of learning systems and described his proposed framework. A year after his publication (2005), Jeffrey Hawkins, Dileep George, and Donna Dubinsky founded Numenta, Inc; the goal was to develop a theory that was informed by the operating principles of the brain as well as an algorithmic framework for the computational capabilities of the neocortex. At a glance, the intellectual framework for HTM was created by Jeffrey Hawkins; the original implementation, Zeta, was designed by Dileep George; around the time that George separated from Numenta to pursue an independent venture—Vicarious—the company modified the HTM theory to be in its current state today, i.e. the HTM CLA; and several applications for CLA have been presented by the company, albeit in limited release. Today, Numenta is comprised of a small team of fewer than 50 developers that focus on refining the theoretical framework for HTM.

Numenta’s primary achievement has been the development of the Numenta Platform for Intelligent Computing (NuPIC) [59]. It has since been made available as an open source software (OSS) IP [60]. Since its creation, NuPIC has also leveraged the work of another OSS initiative, OpenHTM, and incorporated contributions from a community of researchers. Using the HTM technology, Numenta has publicized three applications for HTM—(1) Grok for stocks and IT analytics; (2) rogue behavior detection; and (3) geospatial tracking. The project has been reasonably successful, generating interest within multiple research groups, and large companies such as IBM, which has built a complete research center to investigate HTM (i.e. the Cortical Learning Center). Two major criticisms of HTM CLA are: (1) it has sought to explore a more biologically inspired route at the expense of abandoning its mathematical foundation; (2) it has not yet provided results that have surpassed the state-of-the-art technology on any common dataset. Achieving this in the future will continue to be a challenging feat as the application space of the state-of-the-art and HTM continue to diverge from
2.3.1 Generation I: Zeta

*On Intelligence* provided the intellectual framework for the dissertation of Dileep George [61]. In this work, the foundation for Hierarchical Temporal Memory had been laid—i.e., HTM Zeta, a Bayesian inference network [62]. George’s work presented an unsupervised algorithm that utilized Bayesian inference and Markov graphs to build a mathematical model for a pattern recognition system. Proposed as an unsupervised learning algorithm, it seeks to find invariant representations of data without being explicitly informed of the classification scheme. The Zeta algorithm was summarized and subsequently applied to some small datasets in an official early publication from Numenta [63].

Time-varying data were supplied as input to Zeta, resulting in what George asserted was an invariant input representation, namely a manifold. Temporal slowness served as a core assumption that made the system feasible [56]. Using Zeta, an invariant model may be constructed assuming the input data are observed to be changing over time in a manner dependent upon the object structure and motion of the observing mechanism. An invariant representation of the input was generated by using backwards belief propagation. Originally a generative learning model, it was inherently an offline learning process.

This network was constructed from Zeta nodes (Fig. 2.14). George’s initial implementation of Zeta was tested on a custom dataset called *Pictures*, yielding 99.73% and 57% classification accuracies for the training and testing datasets, respectively. The official dissemination of the principles behind the Zeta algorithm is found in [62] and the most in-depth discussion of it in its original form were the contents of Dileep George’s dissertation work [61]. Details regarding the Zeta source code were not made publicly available, because HTM was private IP at that time. Instead it was originally presented at a high-level [63].

The original HTM Zeta algorithm was built upon two core principles—hierarchy and temporal association. Hierarchy is an important principle for
Figure 2.14: The HTM Zeta node structure, which utilizes Bayesian belief propagation. Zeta was a modified version of a hierarchical Bayesian network to model both spatial and temporal data associations. \( \lambda^k \) is the vector passed by the \( k \)th node to its parent, and \( y^k \) is the vector that the spatial pooler sends to the temporal memory. These nodes may be chained together hierarchically by connecting the output from one node to the input of another node. The meanings and derivations of the equations describing each node may be found in the 2007 white paper [63].

HTM systems for a variety of reasons: (1) shared representations ultimately lead to improved generalization and storage efficiency; (2) models of the real world must consider both the spatial and temporal hierarchy thereof; (3) belief propagation ensures “fast” stabilization of the network and mutually compatible beliefs across layers; (4) hierarchy enables a mechanism for attention. Through this hierarchal model, the belief network is able to extract larger timescale and higher complexity features using lower level time-varying data. Temporal context is important to Zeta because: (1) It provides a rule to associate patterns in an unsupervised fashion; and (2) it improves invariance in the data representation.
Zeta was reviewed by several other researchers and one notable example was presented as a PhD dissertation [64]. Rozado, evaluated HTM and described several possible applications for Zeta ranging from sign language recognition to eye tracking. To test the HTM algorithm, a toy dataset was used in which a small 32x32 pixel binary image was presented as the input to the HTM’s input nodes. These images were translated and scaled, creating a “movie” as was done in the work of George [56, 61]. In this envisioning of the algorithm, deterministic connectivity was also used. The notion of “quantization points” was also discussed—new representation vectors would only be created for an input if it was significantly unique from other vectors that had previously been observed. Therefore, spatial pooling would remove noise from patterns. Whereas, for temporal memory, groups were formed that associated spatial patterns that occurred at different points in time. An extension of the Zeta algorithm was made so that it would support multivariate data, called the sequence pooler. This was applied to sign language recognition.

In an important study, Melis et al. discussed the Zeta algorithm, acknowledging that it lacked a hardware VLSI model and then presented such a model, albeit as a concept and not as an implementation [65]. Melis concluded that hardware acceleration for Zeta would be requisite, because of its inherently large dependence upon floating point multiplications, additions, and memory-bound behavior. Significant quantities of data were required specifically for the matching probability calculation. A later contribution of the same group of researchers was to discuss the use of Zeta for real world applications [66].

Fan et al. presented the only known complete CMOS of HTM (in any of its two major revisions) [55]. A low-voltage resistive cross-bar network (RCN) was implemented at the 45nm technology node. Because of their inherent noise resilience, it was proposed that some machine learning algorithms are well suited to noisy semiconductor circuits. Results were acquired by creating a simulation pipeline based upon SPICE. A classification accuracy of 95% was acquired for the MNIST dataset.
Figure 2.15: HTM depicted having 3 hierarchically organized layers, forming a tree-shaped structure of nodes. Each layer may consist of multiple regions, where a region is comprised of both spatial pooler and temporal memory. Input data are consumed by the first level of the network and passed up toward the highest level. Output of the system is a result of the temporal learning process of level 3, which is a combination of both the current state of the system as well as the predicted next state.

2.3.2 Generation II: CLA

Since the creation of Zeta, HTM theory has undergone several incremental revisions—resulting in the CLA. The remainder of this section provides an overview of the HTM CLA as well as the relevant prior art in the literature. A description of the algorithm as described in the official white paper is also provided [50].

After Dileep George moved on to create Vicarious, Numenta changed directions to diverging from the Bayesian inference and Markov graph foundation of Zeta and replaced it with CLA. This was a radical move for HTM, because it made it more challenging to characterize the behavior of the algorithm using previously acquired knowledge from probability theory. Another major difference between CLA and Zeta is that the significance of
sparse distributed representations (SDR) was made more pronounced in CLA, whereas before, the necessity for merely distributed representations was more strongly emphasized. A representation of the HTM CLA architecture at a high-level is shown in Fig. 2.15, where each region is structured as depicted in Fig. 2.16–these will be described more as the algorithm is explained.

SDR has been in use for some time now in the field of image processing and have been used, because it was hypothesized that the brain operated in this fashion. However, the nature of SDRs and the intuition for why they should be used are not fully understood; sparse distributed representations find their origin in the neuroscience community. Subutai Ahmad of Numenta, along with Hawkins, published a paper summarizing the aspects of the SDR concept that were borrowed by HTM [67]. In general, using larger SDRs improves the separation capability and noise resilience within SP. In light of this, SDR size is traded-off, because at some point increasing the SDR makes the algorithm too computationally demanding.

CLA was studied by several other researchers in the field, primarily focusing on addressing methods to accelerate the algorithm, and apply it to interesting applications. In addition to this, overviews regarding the technology have also been presented [68]. However, something that is quite rare are hardware implementations of the HTM CLA–the same is true for Zeta.
A key focus within HTM literature is noise resilience.

In the work of Liddiard et al., HTM CLA was implemented and evaluated using pseudo-sensory data [69]. However, the analysis was lacking with regard to specificity in the methodology. Furthermore, a non-standard dataset was presented and not characterized to satisfaction. SP was implemented and little information regarding this implementation was presented.

A multi-core implementation of HTM CLA was implemented in [52]. A C-based many-core implementation of HTM was also developed by Zhou and Luo on the Adapteva hardware platform [53]. Vyas and Zaveri presented a verilog implementation of a portion of the design, namely the spatial pooler in HDL [54]. More recently, Zyarah designed an architecture that modeled many concepts of CLA and resulted in significant speedup over a matlab-based software implementation [51]. These architectures share some commonalities, and primarily focused on performance scaling, but discussions regarding scalability are lacking.

2.3.3 Spatial Pooler

HTM CLA is comprised of two algorithms–namely, the spatial pooler (SP) and temporal memory (TM); TM is referred to as the temporal pooler (TP) in the original white paper. SP, serves two primary purposes, to reduce noise from an input signal, and create a sparse distributed representation (SDR) of an encoded input signal. At a high-level the HTM system is composed of hierarchically-arranged regions (Fig. 2.15). Each region is composed of columns, which themselves are formed of functionally related cells. SP operates at the columnar level. In biology, cells within the human neocortex are roughly structured in a similar manner. Neuronal cells that share a common function are connected through a hierarchical pathway. Hence, a column is a group of cells that share the same functional purpose.

Concerning the hierarchy, regions within lower levels (closer to the input) model more simple patterns within the data. As the hierarchy is traversed upward into higher levels, the information represented by that region increases in complexity. Hierarchy within the human brain helps extract
Figure 2.17: The overlap for each column is computed through a three step process; this figure depicts the first step—namely, the overlap computation. The overlap value of the $k_{th}$ column, $C_k$, is initialized to 0 at each step in the simulation in which a new input representation is presented to the SP. Next, each potential synapse of this column, $C_k[j]$, is checked to see if it is connected to an active bit in the input vector. If the synapse is also connected (above threshold, $P_{th}$), the overlap for the column will be incremented by one. Once all the active, connected synapses for the column have been evaluated, the overlap score is then subject to the minimum overlap. All columns with overlap scores below this value are excluded from the next steps. However, columns that exceed the minimum overlap are subject to a positive boosting factor greater than or equal to one. This process is repeated for the remainder of the columns.

useful patterns from complex input data. In the literature, a common example of this is the idea that a lower level may extract edges, then the next level may, in turn, extrapolate more composite features such as eyes, wheels, hands; at the highest level complete objects would be identified. Despite the emphasis placed on hierarchy, no research has been identified that implements multi-region, multi-layer HTM while also demonstrating the benefits thereof. Instead, a single region, composed of SP and TM are scaled up in complexity.
Another concept utilized by HTM CLA theory is the notion of a two-class taxonomy of synapses; proximal and distal synapses. Simply stated, proximal synapses describe feedforward synaptic connections, while the distal are shared between cells in the region. In a single layer network, proximal synapses are formed between an input vector and a layer of cell-containing columns. HTM, at first glance may appear very similar to a feed-forward neural network (Fig. 2.15). However, this is a misconception—HTM differs in the specifics of the learning rule and also in that the contribution of input data to the output of a column is binary—either 0 or 1. Multi-Layer Perceptrons (MLPs) differ in this sense, because they compute a weighted sum, followed by a thresholding calculation. HTM synapses are either connected, or have the potential to be connected (i.e. potential synapses).

A synapse is determined to be connected if its “weight” (called its permanence) is above a predefined threshold. This feature was included to abstractly model the neuronal firing dynamics observed within the cortex. When an HTM SP network is initialized, each column is connected to some subset of the input space through some connection rule. In the white paper, random proximal synapse connectivity was described, however, more structured connection rules have also been used in the literature; connecting columns to input bits that are “close” to one another. Permanence values are also randomly initialized.

Spatial pooling, a columnar operation, begins by first calculating the overlap for each column (Fig. 2.17). Overlap is a measure of the amount of input bits that the column is connected to, in which two criteria are met:

1. The synapse connecting the column and the input bit has a permanence above the predefined threshold—0.2 in the white paper
2. The connected input bit is active

At the reception of new input, each column starts out with zero overlap and then begins to update its values subject to the feedforward input. This phase ends by pruning out all columns with overlap scores that are below a minimum threshold. To reduce the starvation of columns, a boosting mechanism is implemented that takes into account the frequency of activity for each column. After all overlap calculations are completed, columns begin
the inhibition process (Fig. 2.18). This phase may be implemented in two versions: (1) local inhibition; and (2) global inhibition. The latter of the two, global, being a special case of the other. In the literature, global inhibition is commonly used because it yields higher computational efficiency with no identifiable loss in classification performance.

Inhibition, as described in CLA, is a competitive learning process in which columns compete against others within their inhibition radius and lose if they are below the $k^{th}$ largest in range. For the global case, all columns are assumed to be neighbors of one another, and as a consequence, a static inhibition radius is assumed. Furthermore, the lack of a changing radius of a size below the network dimensions results in the inhibition process simplifying down to a $k$-select operation on the overlap scores ($O(n)$ average time complexity). All columns below the threshold are considered to be inactive, and all those above to be active.

Following the inhibition phase, is the synapse permanence update phase
Figure 2.19: The third and final phase of the spatial pooling algorithm–learning. The permanences connected to an active column are trained up or down depending on the value of the input bit that they were connected to, by either $P_{inc}$ or $P_{dec}$.

(Figs. 2.19, and 2.20)–only calculated when learning is enabled. All columns that were active after inhibition are determined to be candidates for learning. Synapses on the proximal segment connected to active inputs are positively reinforced, while those that were connected to inactive inputs are negatively reinforced. Reinforcement occurs by incrementing or decrementing permanences by fixed quantities, $P_{inc}$ and $P_{dec}$, respectively. Permanence values are bounded between 0 and 1, inclusive.

After permanence values are updated, the boosting factor is updated. Each column has a unique boost value that is determined by its duty cycle. There are two types of duty cycles; active duty cycle and overlap duty cycle, dependent upon the frequency of columnar activity and the frequency of overlap, respectively. These processes are not adequately described in the white paper, but were instead found by surveying the NuPIC implementation. Upon training all proximal synapses for every column, the inhibition
radius is updated so that it would approach a preferred steady-state. The minimum overlap is also another quantity that is updated in like manner to the inhibition radius.
Chapter 3

Proposed Flash-Based HTM Design

Several hardware implementations of HTM CLA have been presented in the literature. Flash Hierarchical Temporal Memory (FHTM) is a unique hardware implementation of CLA spatial pooler that leverages PCIe SSD architecture (Fig. 3.1). It is a proof of concept that, with minor modifications, a storage device may be augmented to serve as a scalable SPU for machine learning applications.

To furnish a more comprehensive design analysis, the CLA spatial pooler was implemented at three levels of detail. Network parameters have been selected using a matlab mathematical model, as described in Section 3.1. A transaction-level model (TLM) is presented in Section 3.2 to address high-level system architecture concerns. An RTL microarchitecture of the SP is presented in Sections 3.3, respectively. Each level of abstraction provides intuition regarding the benefits and challenges of implementing a scalable HTM SPU.

3.1 Spatial Pooler Math Model

This section presents the mathematical formulation modeled by the FHTM architecture. Each aspect of spatial pooling is progressively discussed, starting with a simple statement of overlap and eventually arriving at the full embodiment of the spatial pooler with a learning rule. Limitations of the model are presented as well to identify where FHTM delineates from CLA.
3.1.1 Phase I: Overlap

Spatial pooling, forming the first partition of an HTM region, seeks to create a mapping from an encoded input to a sparse representation. Columns are represented as a vector of permanence values. Once the magnitude thereof is above a threshold, $P_{th}$, a potential synapse is promoted to the connected state. This relationship is described as

$$\vec{C}_i = \vec{c}_i \geq P_{th}$$  \hspace{1cm} (3.1)

where $\vec{c}_i$ represents the permanence vector for the $i^{th}$ column; $\vec{C}_i$ is the binary vector of columnar synapse state—a value of 1 is indicative of a connection. There is a one-to-one mapping in terms of the size of each column vector and the number of bits within the input space.

$$\alpha'_i = \vec{C}_i \cdot \vec{X}_t$$  \hspace{1cm} (3.2)

$$\vec{\alpha}' = \vec{C}^T \cdot \vec{X}_t$$  \hspace{1cm} (3.3)
The column space of SP is represented as a $K \times N$ matrix, $C$, in which the $i^{th}$ column corresponds to the proximal segment of the $i^{th}$ HTM column. Each of the $N$ columns are capable of being connected to a maximum of $K$ synapses. Overlap for the $i^{th}$ column is a dot product of its proximal segment state vector and the binary input vector (3.2). The entire network may be evaluated in parallel using a dot product between the transpose of $C$ and the current input state vector, $\vec{X}_t$, as shown in (3.3); $\alpha'_i$ and $\vec{\alpha}'_i$ are overlap for the $i^{th}$ column and the entire network, respectively. This relationship is extended to account for the boost factor in addition to minimum overlap, yielding (3.4).

$$\alpha_i = \begin{cases} \alpha'_i \beta_i, & \alpha'_i \geq A_{th} \\ 0, & \text{otherwise} \end{cases}$$

(3.4)

where $\alpha_i$ is the overlap value for the $i^{th}$ column after boosting and minimum overlap thresholding; $A_{th}$ is the minimum overlap threshold; and $\beta_i$ is the boosting factor. The white paper defined the boost factor to always be greater than or equal to unity.

### 3.1.2 Phase II: Inhibition

**Local Inhibition**

Local inhibition, although not implemented, is described here. It begins by first calculating locations (within the column space) for all columns, $C_j$ within the inhibition radius of another column $C_i$ such that

$$j \in \{i \pm (D_{cs}m + n)|m, n \in \mathbb{N}_0 \leq r_i\}$$

(3.5)

where $j$ is the index of a column within the inhibition radius of the $i^{th}$ column, $D_{cs}$ is the number of HTM columns aligned in one dimension (the 2D column-space is assumed to be a square), $m$ and $n$ are used to index through the list of inhibited columns, $r_i$ is the inhibition radius of the $i^{th}$ column.

Inhibition is applied using a k-select operation and a prune step to remove all elements in the set with a value below that of the selected element. This
may be modeled by sorting the set of columns, \( j \) within the inhibition radius of another column, \( i \); hereto referred to as \( \Lambda^j_i \). Pruning is modeled in (3.6) by removing columns below a specified index from the aforementioned set, which now becomes the SP output SDR.

resulting in

\[
A_i = \alpha_i \geq \vartheta[i, d_{th}]
\] (3.6)

\[
\vartheta[i, d_{th}] = \Lambda^i_j[\min(d_{th}, \text{length}(\Lambda^i_j))]
\] (3.7)

where \( A_i \) is the binary inhibited columnar activation; \( d_{th} \) is the desired local activity; \( \vartheta \) is a bounded indexing function–used to index into \( \Lambda^i_j \), subject to \( d_{th} \); \( \vartheta[i, d_{th}] \) calculates the minimum local activity for the \( i^{th} \) column; the \( \min \) function is used to restrict the range of \( \vartheta \) to indexes within \( \Lambda^i_j \).

Global Inhibition

Global inhibition is a special case of the more generalized local case. This case is simplified, as the inhibition radius of each column is known and remains constant. The global case is simply a configuration in which the “local” inhibition radii of all columns are large enough to encapsulate the entire column space. Consequently, the equation governing both global and local cases are, in principle, the same. Most implementations of SP in the literature utilize global inhibition, such as in [51]; both yield comparable results [70]. With regards to hardware implementation, the global case is more practical, requiring fewer parameters to be tracked, such as inhibition radii, implicitly removing the requirement for updating this parameter dynamically.

3.1.3 Phase III: Learning

Synaptic Adaptation

The third, and final, step of spatial pooling is to commit learning updates to the proximal segment state matrix. Columns are updated as a function
of prior input patterns and the current feedforward input vector; the latter is dictated by (3.8).

\[
\tilde{C}^*_i[k] = \begin{cases} 
\tilde{C}_i[k] + (P_{inc} + P_{dec})\tilde{X}_t[k] - P_{dec}, & A_i = 1 \\
\tilde{C}_i[k], & \text{otherwise}
\end{cases} 
\tag{3.8}
\]

where \(\tilde{C}^*_i[k]\) is the new value for the \(k^{th}\) permanence of the \(i^{th}\) proximal segment state vector; bounded between 0 and 1, inclusive. The \(P_{inc}\) and \(P_{dec}\) correspond to the permanence increment and decrement, respectively. Synaptic state is only updated for active columns and is based upon the binary input vector–if the bit was active, the permanence is incremented, otherwise it is decremented. This operation may be vectorized.

**Duty Cycle Update**

The duty cycle update is utilized in the boosting process and is important to its functionality. This step was independently considered from the boosting step for clarity. First the active duty cycle is updated such that

\[
D^*_A[i] = \frac{D_A[i](\tau_D - 1) + A_i}{\tau_D} 
\tag{3.9}
\]

where \(D^*_A[i]\) is the next value for the active duty cycle, \(D_A\), of the \(i^{th}\) column; \(\tau_D\) is the period over which the duty cycle is evaluated and consequently averaged, and \(A_i\) is the current active state for the column (resulting from the inhibited overlap calculation). Active duty cycle is a measure of how often a column passes through inhibition. A similar process is repeated for the overlap duty cycle

\[
D^*_O[i] = \frac{D_O[i](\tau_D - 1) + \alpha_i}{\tau_D} 
\tag{3.10}
\]

where \(D^*_O[i]\) is the next value for the overlap duty cycle, \(D_O\); and \(\alpha_i\) is the new post-inhibition overlap resulting from the current feedforward input. Although active and overlap duty cycles were implemented having
the same period, this is not an inherent limitation of CLA—this was taken as a reasonable hardware-simplifying assumption.

**Weak Column Boosting**

Weak column boosting, described by (3.11), seeks to increase the number of synapses that are in the connected state for potentially starved columns. The permanence values of a column, $\vec{C}_i$, are increased by 10% of the permanence threshold [50]. Other such magic numbers were used in the white paper, but further work should seek to optimize the HTM to ascertain optimal parameters using parametric optimization techniques such as simulated annealing.

$$\vec{C}^*_i[k] = \begin{cases} \vec{C}_i[k] + P_{th}/10, & D_O[i] < \tilde{D}_O[i], \vec{C}_i[k] > 0 \\ \vec{C}_i[k], & \text{otherwise} \end{cases}$$ (3.11)

where $\tilde{D}_O$ is the minimum overlap duty cycle. If the duty cycle of the column in question is below the minimum overlap duty cycle and the column has not already saturated to a value of 0 (pruned), the permanence values are incremented.

**Boost Factor Update**

The overall goal of boosting is to reduce column starvation, leading to an increase in the number of columns contributing to the process of creating a model of the input data. This aspect of the learning algorithm is modeled as

$$\beta^*_i = \begin{cases} \frac{1-\beta_{max}}{D_A}D_A[i] + \beta_{max}, & D_A[i] < \tilde{D}_A \\ \beta_i, & \text{otherwise} \end{cases}$$ (3.12)

where $\beta^*_i$ is the next value for the boosting factor, $\beta_i$, for the $i^{th}$ column; $\tilde{D}_A$ is the minimum activity duty cycle, and $\beta_{max}$ is the maximum boost factor. This equation is a piece-wise linear function, where $D_A[i]$ is the only quantity that varies between iterations. In the case of the original
HTM design proposal, $\tilde{D}_A$ is also a variable quantity. However, as a hardware simplification, this parameter is expected to be extracted from a more complete software model before implementing the model on the hardware platform.

3.2 Transaction-level Model

Memory latency and bandwidth limitations have presented a significant challenge to scalability in modern computer architectures. The pure von Neumann concept has been replaced with a hierarchical memory paradigm capable of reducing average latency whilst maintaining coherency amongst multiple processors. Creative methods have been employed in an attempt to close the growing gap between host processor performance and the ability of main memory to supply the associated demand. It has been proposed in the literature that, for a subset of applications, latency and memory bandwidth may be improved by implementing computational cores close to storage.

FHTM is driven by this concept.

FHTM TLM, depicted in Fig. 3.2, is a PCIe SSD serving as an SPU that implements the spatial pooling algorithm. Within the system model, control and the overall flow of data are the primary concerns, resulting in some details of the microarchitecture being taken for granted. This SPU acts as a slave device that preprocesses pages of data at the request of a host processor. The remainder of this section is dedicated to describing the flow of data and control through the TLM.

Using both approximately-timed computation and communication, the FHTM TLM simplifies the design effort in modeling the complex SPU subsystem. Each subsystem in model was implemented as a SystemC C++ sc_module; the associated delays and notion of timing are explicitly back-annotated into the model. PCIe communication between the host processor and attached SPU is modeled as a blocking transport interface, namely paired TLM sockets. The host transmits data to the SPU by creating a generic payload structure containing command, data stream length, address, an acknowledgment pointer, and other ancillary parameters. This packet is
Figure 3.2: The FHTM transaction-level model, implemented using the SystemC C++ extension, provides a behavioral implementation of the PCIe interface that models the associated delay using (3.13). This model utilizes an abstract packet structure to transmit data between the host processor and the FHTMSPU. An abstraction of the SP was modeled using an approximately-timed computation core. Flash translation and garbage collection were also modeled. Blocking TLM sockets were utilized to facilitate intercommunication between components.

received by the FHTM system, decoded by the FTL, resulting in a read or write operation being issued to the downstream NAND array.

After a read or write has been issued to the NAND array, it is operated on by the proposed scalable FHTM core. For read operations, data is reduced in a pipelined core from pages of data into the respective outputs for SP–columnar activation. Training of network parameters is represented as a three step process. First data corresponding to a proximal segment is read from memory during the aforementioned feedforward computation phase. Network state information, stored within a DRAM-cached tables, are updated to correspond with the changing network state. Thereafter, pages of segment state data are read, modified, and written back into the memory.
3.2.1 Simulation Controller

A simulation controller (FHTMEnv), system clock, host cpu (FHTMHost), attached SPU (FHTMSPU), and pointers to the configuration payload and MNIST dataset are initialized within the sc_main. The simulation kernel is initialized by making a call to sc_start, resulting in a trace being generated, demonstrating the interaction between host and attached SPU. FHTMEnv manages initializing and subsequently connecting the host and SPU by binding their sockets together. Two sc_thread functions are used to issue the control and monitor the clock, respectively. Once the simulation time has been exhausted the sc_stop command is issued by FHTMEnv to conclude execution and close the trace file.

FHTMEnv generates a sequence of commands that are issued from the host to attached SPU; a state machine facilitates this process. Ready and done flags are shared between the host and simulation controller to support interoperability. System clock, a command, address, and input region vector sc_in ports are all driven by FHTMEnv to control the host. Spatial pooler output is monitored through an sc_out interface. FHTMEnv begins the simulation by prompting the host to issue a configuration command to the SPU. Thereafter, the SP algorithm is executed, followed by a proximal segment update step. The simulation flow is depicted in Fig. 3.3.

3.2.2 Host Processor Module

FHTMHost extends the sc_module class and specifies interfaces to the FHTMEnv and utilizes a TLM simple initiator socket to model the PCIe interface. A pointer to the configuration structure defined by the simulation controller, the generic payload object, and a data buffer are defined in FHTMHost. During initialization, following the binding of ports, an sc_thread is bound to the mk_pcie_packet method to access the attached SPU through the blocking transport interface. Thereafter, the aforementioned data structures are initialized, with the host being placed in the ready state; allowing FHTMEnv to transmit a valid command sequence.

Packet construction is initiated in the host as a result of valid command
control from the simulation controller. This distinction between \textit{FHTMEnv} and \textit{FHTMHost} was made with the intention of promoting a clear separation of concerns—simulation control and functional modeling. Once the host receives a valid command from \textit{FHTMEnv}, the packet structure is updated to contain the corresponding acknowledgment. Each abstract simulator command is distilled down into a TLM write or read operation. A page address is then associated with the packet structure and eventually decoded by the SPU FFS.

Payload information is associated with the packet by attaching a generic byte pointer to the structure along with size information. All of the packet configuration follows the standard TLM generic payload structure with no modifications. Finally, relevant \textit{sc\_time} delay information is associated with the payload before being transported through the blocking transport interface. As an approximately-timed communication TLM, the previously calculated delay is explicitly submitted to the SystemC simulation kernel by making the call \texttt{wait(delay)}, which stops execution of the \texttt{mk\_pcie\_packet sc\_thread} until the time expires.

High-level details of the PCIe x4 Gen3 interface were used to model the transmission delay for a packet. Following the wait period, the packet
response status (acknowledgment) is checked to verify that a valid command was successfully transmitted to the attached SPU. *FHTMHost* decodes several abstract commands received from *FHTMEnv*, generating a series of TLM reads and writes; commands include *configure, train_sp, test_sp, read_page*, and *write_page*. The data are then transported through the blocking transport interface, applying a delay defined by the relation

\[ t_D = 1000E \frac{P}{T_RN_L} + \tau_L \]  

where \( E \) is the data transmission efficiency due to the encoding overhead–for PCIe Gen3, the 128B/130B scheme is assumed, resulting in \( E \) being 130/128; \( P \) is the size of the packet, including the header, in bits; the transmission rate, \( T_R \), for PCIe Gen3 of 8GT/s was also factored in (\( T_R = 8 \)); a constant factor of 1000 was applied to scale the equation to be in terms of picoseconds; \( N_L \) is the number of PCIe lanes, assumed to be 4; and \( \tau_L \) is the inherent latency due to transferring data across the physical link–this was assumed to be zero for convenience.

### 3.2.3 SPU Module

A functional FFS, host interface, HTM core, and the attached NAND memory arrays form the SPU subsystem. Each subsystem is implemented as a self-contained *sc_module* that operates on the packet structure in some manner, eventually returning data to the original caller–i.e. *FHTMHost*. The host interface is modeled as a simple target socket that pairs with the *FHTMHost* initiator socket; memory-related interfaces are represented using blocking transport interfaces. *FHTMSPU* is initialized with minimum and maximum address limits–allowing future models to support several attached multi-channel *FHTMSPUs*.

During initialization, the SPU *sc_module* constructor binds its target socket to the *host_interface* function, which receives, decodes, and executes incoming TLM generic payloads. Additionally, the garbage collection routine is executed as a background *sc_cthread*. Flash translation tables are setup in the *sc_module* constructor to be in a clean initial state; there
are four tables maintained by the FFS—namely, the physical block state table (PBST), the physical sector state table (PSST), the logical-to-physical (LTPTT) and physical-to-logical (PTLTT) translation tables.

PBST tracks the number of stale and free pages within the block in addition to the write-erase cycle count. PSST identifies each physical sector as being free, in-use, or stale and also identifies the page size (in bytes). LTPTT converts a logical sector number (LSN) to a physical sector number (PSN) by using the LSN to index into an array of PSN values. For instance, $LTPTT[2]$ will return the physical page number corresponding to the second logical sector; PTLTT works in like manner to convert PSNs back to logical sectors.

The host interface decodes incoming TLM generic payloads, calling the internal `read` and `write` utility functions to execute the command associated with the packet of data. If no errors occur during these operations, the packet status is set to a clean state. Read operations require an initial LSN to PSN translation and subsequent data steering. A flash channel is selected by comparing the PSN against a set of base address values. Once a hit is made, the packet address is set to the PSN and a blocking transport method is called on the corresponding channel.

Address masking is repeated by the channel to select an attached NAND target as the final data source. The previously invalid payload pointer within the packet is updated by the NAND target to reference the desired data. As each `sc_module` terminates, the function call stack unravels, eventually returning to the host processor, which has access to the, now updated, payload structure, containing the data and associated TLM operation success status.

Write operations are considerably more complex than reads, requiring an update to the LTPTT and PTLTT. First, the old PSN is acquired using a lookup into the LTPTT. Thereafter, a new PSN is acquired using a helper function, `get_free_psn`, for the purpose of selecting a channel and NAND target in similar fashion to the read operation. FTL tables are updated at the final stage of the write process, flagging the new PSN as being used. The PSST entry corresponding to the new physical sector is updated to indicate that the entry is being used by valid data and the old physical page is marked as stale. An update is made in the LTPTT and PTLTT to indicate that the
### FHTM Logical Memory Map

<table>
<thead>
<tr>
<th>LPN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Configuration Page</td>
</tr>
<tr>
<td>1</td>
<td>Column J Proximal Segment</td>
</tr>
<tr>
<td>2</td>
<td>Column, Proximal Segment</td>
</tr>
<tr>
<td>i(JN + 1) + 1</td>
<td>Column, Proximal Segment</td>
</tr>
<tr>
<td>i(JN + 1) + jN + 2</td>
<td>Column, Distal Segment</td>
</tr>
<tr>
<td>(K-1)(JN+1) + 10</td>
<td>Column, Proximal Segment</td>
</tr>
<tr>
<td>(K-1)(JN+1) + (J-1)N + N + 2</td>
<td>First Unused Address</td>
</tr>
</tbody>
</table>

**Column 0 Memory Space**

Figure 3.4: The logical memory organization for the FHTM architecture. Physical location of data within the flash memory is not guaranteed to be contiguous. Consequently, a logical mapping is used to hide the underlying structure of the memory from the host device. Each column is assigned a contiguous set of logical addresses on page boundaries to represent proximal segment state. The logical page number (LPN) is defined using several parameters, the first of which corresponds to the column index, \( i \); \( J \) and \( j \) are the number of cells per column and the cell index, respectively; \( N \) and \( \rho \) are the number of segments per cell and the segment index, respectively.

LSN and new PSN are mapped to one another.

For the FHTM design concept, LSN zero is assigned to be a configuration page. Writes to this logical sector update registers in the SPU, altering its functionality and providing semantic meaning to the organization of all other remaining LSNs. All of the parameters are also backed up in the physical NAND array, accounting for unexpected system shutdown. The logical memory organization, subject to parameters of the configuration, is depicted in Fig. 3.4. Information found in the configuration space for the FHTM TLM is presented in Table 3.1.
Table 3.1: Configuration information stored in the zeroth logical sector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncol</td>
<td>Number of HTM columns</td>
</tr>
<tr>
<td>ncell</td>
<td>Number of HTM cells per column</td>
</tr>
<tr>
<td>nsyn_sp</td>
<td>Number of synapses per each proximal segment</td>
</tr>
<tr>
<td>psigma</td>
<td>Standard deviation used during the random initialization of permanence values</td>
</tr>
<tr>
<td>Pth</td>
<td>Permanence threshold</td>
</tr>
<tr>
<td>Cth</td>
<td>Minimum overlap</td>
</tr>
<tr>
<td>Mth</td>
<td>Desired local activity</td>
</tr>
<tr>
<td>learning_sp</td>
<td>Learning mode enable</td>
</tr>
<tr>
<td>Pinc</td>
<td>Permanence increment</td>
</tr>
<tr>
<td>Pdec</td>
<td>Permanence decrement</td>
</tr>
<tr>
<td>Damin</td>
<td>Minimum active duty cycle</td>
</tr>
<tr>
<td>Domin</td>
<td>Minimum overlap duty cycle</td>
</tr>
<tr>
<td>Td</td>
<td>Duty cycle evaluation period</td>
</tr>
<tr>
<td>xtlm</td>
<td>Input vector size</td>
</tr>
</tbody>
</table>

Each logical sector holds the state information for a proximal segment—corresponding to a column. Pages containing proximal segment state store the duty cycle and boost factor in the first few words of data. The remaining portion of the page contains words of synaptic permanences. In the context of this discussion, a “word” is equivalent to the data width of the flash interface channel. Given this memory hierarchy, the network size is loosely bounded by page sizes. Modern SSDs support page sizes in the range of 4kB to 64kB. This allows each proximal segment to map into an input region proportional to the page size divided by the number of bits used to represent the synaptic permanence.

### 3.3 Register Transfer Level Model

The microarchitecture for FHTM was implemented in VHDL to gain insight regarding feasibility of the model with respect to area, power, and functionality. FHTM RTL serves as a direct representation of the mathematical model as a scalable, parallel, pipelined processor. An FFS layer of abstraction manages scheduling read and write operations to leverage the
inherent parallel nature of a multi-channel SSD. FHTM processor logic is added to the read and write data paths, trading off increased latency for improved parallelism and greater potential for scalability.

A page of data, representing a proximal segment, is steered into the FHTM read pipeline and operated on via a reduction process. The resultant data to be delivered from the SSD back to the host is significantly compressed. Removing processing from the host to storage allows for attached SSDs to act as accelerators, greatly reducing the bandwidth requirement for the PCIe data link. Latency added by the pipeline also scales more productively as the model size increases, requiring more memory operations. In a standard system, all data would necessarily be transferred from storage to host, leading to eventual bottlenecks in the PCIe interface. However, with an FHTM SPU, attaching more storage increases the bandwidth requirement at a much slower rate, because data being transferred over the PCIe interface is limited to control (the broadcast of the $X_t$ vector and HTM network configuration) and the compressed output.

In a standard non-SPU enabled processor, the host initiates data processing by requesting data from the SSD, resulting in a transfer of entire pages back to the host’s main memory or cache, leading to a large amount of page swapping and cache line evictions. After completing the computation of HTM CLA in the host, only an overlap value along with several words of segment state information would remain in the host. This data would then be stored in a data structure that would be at risk of being evicted from the cache due to the large quantities of data being read by the host. During learning, the host CPU would also have to reload all pages of data, update them, and write them back to memory. Because computation is far from the storage, round trip delays for each operation would have a significant impact on system performance and power consumption.

Utilizing an SPU allows for the host to issue commands to all attached SSDs, which may then independently operate on the data. Furthermore, because the storage elements are not on the same die as the host, power dissipation requirements will be much more lenient. Command data would utilize a small amount of bandwidth and would be broadcast to all attached SPU cores, allowing them to work in parallel. Each FHTM core would exploit
the multi-channel infrastructure present within the SSD to allow for parallelization across 8-20 memory channels.

At the end of computation, each SPU would prepare a packet of output data for transmission to the host. Whereas a non-SPU enhanced design may lead to gigabytes of data being transferred back to the host, this design results in kilobytes of data be transferred back to the host. Also, in this case, the data in cache would be less volatile. Considering that the memory is accessed in both scenarios, an SPU-enhanced model presents significant benefits for designs where scalability is a major priority. FHTM augments read operations to support active column calculations for SP; write operations are expanded to implement learning functionality. The remainder of this section discusses the nuances of FHTM microarchitecture. Each path is described at a high-level and then its subsystems are expounded upon in later sections.

Spatial pooling in FHTM is modeled by augmenting the data path with an overlap pipeline, inhibition engine, and learning component. Data flow of the FHTM SP, depicted in Fig. 3.5 follows a clear process. (1) The SSD controller initiates operation by issuing a read command to the flash channel. (2) The overlap pipeline (OVPipe) is enabled, along with the other functional pipes in the read path and provided a pointer to the input vector and configuration data, broadcast by the host; then, data arrive from the flash interface one beat at a time. (3) Overlap values are selected by the channel arbiter (Charb) for boosting and inhibition. (4) After the single columnar overlap and segment state information traverse the pipeline to this point, the inhibition engine (Inheng) computes an insertion sort operation to determine the subset of columns that will enter the active state.

In the case of learning, (5/6) the controller issues column indexes to the content addressable memory arbiter (Camharb), which arbitrates between channels to select a column index to be sent downstream. (7) The write-back content addressable memory (WBCam) is used to determine which columns are candidates for learning by comparing the issued column index against valid indexes acquired in the inhibition phase. (8/9) Hits in the CAM are detected (Camhit) and used to control the write-back ALU (WBPipe), which writes the modified data, cached in the segment cache, back to memory.
through the flash interface. Timeout signals are forwarded back to the SSD controller to indicate a CAM miss. State information within the inhibition engine is used to select operands for the ALU within $WB_{Pipe}$, which updates each segment one synapse at a time. The remainder of this section is devoted to discussing the details of each functional block depicted in Fig. 3.5.

### 3.3.1 Overlap Pipeline

The first step in spatial pooling is the calculation of columnar overlap values. This step is the primary reduction step, compressing a page of proximal segment data into a single natural number. First, the host writes configuration data to logical page zero (Fig. 3.4), triggering a write to configuration registers within the functional pipeline. Initialization defines the logical structure of the architecture corresponding to the memory map depicted in Fig. 3.4. Following this step, all valid permanences must be initialized to
random values centered around the permanence threshold. In this phase, \textit{WBPipe} is configured to operate in a pass-through mode, allowing explicit writes from the host CPU into memory with randomized permanence values.

After the FHTM architecture has been configured by the host processor, the SPU, waits for an input stimulus, \(X_t\), to be written to the SSD. In the case of MNIST, the host processor transmits a packet containing the input vector. Upon reception of the input vector, the FFS creates commands that signal the NAND interface to supply pages of data corresponding to stored proximal segments. Each beat of data supplied by the interface corresponds to synaptic permanences and other SP column state. The codewords of data
sourced from the NAND, after LDPC decode, are supplied as input to the OVPipe.

The overlap pipeline, depicted in Fig. 3.6, need only operate at the data rate of the memory interface. Once the first beat of data arrives to the overlap engine from the LDPC, all subsequent beats of data will become available in the immediately following cycles. Furthermore, because data being read have a predictable structure, a drastically simplified overlap engine was designed. Additional simplification is obtained by exploiting the simplicity of mathematical constructs used by the algorithm.

On the order of tens of cycles are added to the data path due to the vector processor. To put this into perspective, the number of cycles required for the read and write operations are on the order of hundreds to thousands of cycles. Taking into account the interface limitations, additional pipeline complexity is likely to have a negligible impact on performance. As design complexity scales, resulting in larger page sizes, more channels, and higher data rates, this simple model will continue to be feasible and effective. This is because the pipeline has few opportunities for significant critical path limitations, because (1) the data interface is only 8-bits to 16-bits wide in most flash implementations, leading to lower computation delays than common 32-bit and 64-bit hardware; (2) there are a small number of inputs to each logical component; and (3) operating at the memory interface frequency provides a large upper bound on allotted cycle time.

As a new \( X_t \) input vector arrives from the host, a pointer, \( j \), is reset to reference the first bit of the vector. The multiplexer is utilized to select the respective bit within the vector. This first portion of the overlap engine (separated by a dashed line) computes the pre-boosted overlap for the column using an accumulator. Input data arriving from the host are one-for-one matched with the number of potential candidate synapses. Each synaptic value arriving from the flash interface is thresholded using a comparator. An AND-gate determines that an active-connected synapse was found. In the event that a proximal synapse is both active and connected, the overlap value is incremented. The internal state of the accumulator is reset whenever a new input vector arrives. This complete process takes an amount of time proportional to the page size divided by the channel width—it spans the
time required for the read operation plus the overlap pipeline latency.

3.3.2 Channel Arbiter

Following data accumulation in OVPipe, the channel arbiter, shown in Fig. 3.8, is notified that valid data is ready for boosting and inhibition. Only overlap values that have overcome the minimum-overlap constraint issue a valid data request to the downstream logic. A full handshake is computed using the logic depicted in Fig. 3.7. The purpose of this arbitration is to ensure that only one overlap value is sent downstream and that the other channels are stalled until the inhibition engine is able to service each request in order.
Figure 3.8: The channel arbiter (Charb) manages transmitting overlap values from multiple overlap engines to the downstream inhibition engine. Furthermore, the overlap values accumulated in OVPipe are boosted at this phase prior to being distributed to the inhibition engine. Proximal segment data selected to go downstream results in the associated channel receiving a data grant acknowledgment, dgnt.

3.3.3 Inhibition Engine

Inhibition for SP is modeled as an insertion sort in which values within the sorted list below a predefined index are discarded from said list. To accomplish this, a pipeline capable of sorting elements in linear time was created. Inheng, depicted in Fig. 3.9, is represented as a multi-directional shifting queue. Overlap values generated from the activation due to the feedforward input are transferred to the inhibition engine after being selected by the channel arbiter. Considering the fact that overlap requires hundreds of cycles to compute, the inhibition engine is allotted a large window of time to complete each sort operation. As the overlap pipeline operates on the page data, the inhibition engine attempts to place, and subsequently sort it within
Figure 3.9: The inhibition engine for the FHTM architecture. Control logic is generated by the swap state registers in conjunction with data valid bits and the results of comparison operations (Fig. 3.10). The inhibition engine state is cleared upon the arrival of a new input vector, $X_t$. Note that the column index data path is abbreviated for clarity—it follows a similar path to that of the overlap swap logic, $D_i$.

This design assumes that the large number of cycles required to compute the overlap eliminates any possibility of overrunning the channel arbiter and in-turn, the inhibition engine. Inheng applies back-pressure to Charb, stalling it until the prior sort operation has complete. Back pressure acts in a flexible way such that the complete pipe need not be stalled by downstream blockage. Pressure is funneled back upstream stage-by-stage, allowing gaps in the pipe to be filled where possible. For example, if Inheng is applying back-pressure, but the output of arbitration is invalid, valid data may still move forward in the pipe, calculating the boosted overlap.

Data being shifted around in the inhibition queue are $C + C_{IDX}$ bits wide; where $C$ is the flash interface width, and $C_{IDX}$ is the number of bits required
Figure 3.10: The inhibition engine control logic, dictating when and how to swap data. Overlap values are swapped in one of two cases: (1) the magnitude of the $i^{th}$ overlap is less than that of the $(i-1)^{th}$ overlap preceding it, and both $D_i$ and $D_{i-1}$ terms are valid; or (2) the magnitude of the $(i+1)^{th}$ overlap is less than that of the $i^{th}$ overlap, and both $D_{i+1}$ and $D_i$ terms are valid. The swap state register output, $sv_i$, identifies the swap direction; $sv_{i-1}$ is the propagated swap direction and is connected to the prior swap state register output. If the queue is not full, a shift_data signal is asserted to automatically load incoming valid data; $dv_i$ is used to indicate the valid data corresponding to the $i^{th}$ columnar overlap.

to encode column index. The linear sort implemented by the inhibition engine is composed of multiple complementary swap operations, governed by the sequential control logic depicted in Fig. 3.10. Each swap operation is controlled using a two-bit signal, $M_i$, generated by a swap state register in conjunction with additional combinational control logic. The most significant bit of the swap indicates direction, while the other bit is utilized as a functional enable; data elements within the queue are swapped using a multiplexer.

Valid bit, column index, and overlap values are moved through the pipeline, controlled by the comparison of overlap data. As new data are loaded, swap requests propagate through the queue until a valid swap operation is no longer possible. The swap logic checks for validity between the two elements, compares them with regard to magnitude, taking into account queue fullness. Inheng is deactivated after completing the swap operation as it waits for new overlap data to become available.

The control signals are specified to allow all registers to shift in the same direction when new data are loaded, and to subsequently sort that data. After
3.3.4 Content-Addressable Memory Channel Arbiter

Read data for SP flows from *OVPipe*, to the *Charb*, finally ending in *Inheng*, which supplies output to a packet formatter to be emitted through the host interface. Learning follows a slightly different data flow and must immediately follow the completion of inhibition for all columns. Indexes for all columns in *Inheng* are fed into the *WBCam* for use in the learning phase.

all HTM columns have been processed by *Inheng*, the column indexes corresponding to the overlap values stored within the queue are used to generate an SP output packet due for transmission to the host processor. Column indexes generated by the inhibition phase are also used in the learning phase to influence proximal segment update.
Proximal segment update is initiated by the SSD controller, which must schedule reads for all columns in the network and also issue corresponding notifications to the content-addressable memory arbiter (Camharb), shown in Fig. 3.11. The SSD controller is notified of hits and misses in the CAM by Camhit.

### 3.3.5 Write-back Content-Addressable Memory

WBCam is comprised of chained compare units, depicted in Fig. 3.12. For each stage other than the first, if a hit is detected, the data is invalidated in the next stage in the pipeline to avoid another subsequent comparison. This feature is requisite to avoid invalid detection of timeouts in the pipeline. For all compare components other than the first ($i > 0$), the data valid output is defined by (3.14).

$$dv_i = dv_{i-1} \land \overline{inv_{i-1}} \land (cidx_{i-1} = inhidx_{i-1})$$  (3.14)
Proximal segment data are read from the memory, and subsequently cached in the lower latency DRAM. Upon completion of the read operation, the data is written from the segment cache in DRAM back to the SSD NAND array through \textit{WBPipe}. Operands for the write-back operation are selected based upon whether a hit was detected in \textit{WBCam}, and upon the overlap duty cycle. Proximal segment state information is updated in the \textit{WBPipe} as well. In this scheme, the memory can queue up several reads to occur in order, followed by segment updates.

### 3.3.6 Content-Addressable Memory Hit Notifier

Hits in the CAM are associated with their originating channel via the \textit{Camhit} component, which utilizes a series of equality comparators to aggregate hits corresponding to a channel of interest. This scheme also follows the assumption that each channel will only issue a single column index into \textit{WBCam} at a time to remove the possibility for ambiguity with respect to detected hits. An additional comparison is made with the final phase of the \textit{WBCam}—this index is assigned to the timeout operation. Column indexes are indicated as having timed out if they have been compared against all data in the inhibition engine without having matched any other data. A hit detector unit for a single channel is depicted in Fig. 3.13—in an SSD with \( N \) channels, this logic is repeated \( N \) times.

Using the results of CAM hits and misses, the SSD controller modifies the input operands to the \textit{WBPipe} ALU. A hit in the CAM indicates that the column was active and therefore a valid candidate for learning. All segments are still re-read and written back to memory in the event that an update is pending for the column. Segments may be updated due to being in the active state following feedforward input or as a result duty cycle updates. In the case of active duty cycle, the whole segment need not be updated, only the active duty cycle and boost factor; columnar activity and overlap duty cycle result in an update being made to the entire set of synapses on the segment. Duty cycle parameters may be cached in the DRAM as proximal segments are first read from memory by the \textit{OVPipe} to control scheduling segment updates.
Figure 3.13: The Camhit component is comprised of comparators used to determine whether the ID for the \(i^{th}\) channel, \(chid_i\), has been matched in the CAM and returns notification to the SSD controller indicating the event. Valid signals, \(cv_{0..D}\), are used to qualify each comparison. A timeout signal, \(chtimeout_i\), may also be asserted indicating that a compare operation timed out, resulting in a CAM miss.

### 3.3.7 Write-back Pipeline

State information regarding the proximal segment is updated by the write-back data pipeline shown in Fig. 3.14. This pipeline is capable of updating segment duty cycle and boost parameters in addition to the synaptic permanences on the segment itself. Flash interface data is redirected to the pipeline associated with control information provided by the SSD controller. With this topology, data can be streamed in and out of the update pipe at the flash interface speed to reconstruct the original proximal segment page, given the correct control sequence. Output from \(WBPipe\) may be sourced from the duty cycle pipeline, the old boost factor value register, or the proximal segment update data path.

Ancillary segment state data are updated using the pipeline presented in Fig. 3.15; segment update is handled by the ALU presented in Fig. 3.16. Configuration inputs, labeled \(y_{1..9}\), each correspond to various pre-calculated constants necessary to compute updates in the learning phase. Each constant is defined in (3.15). The segment update formulas were modified to be more amenable to the FHTM design concept, meaning that variable parameters were separated out and divisions were represented as pre-calculated constants instead of in the hardware. This modification made
Figure 3.14: High-level view for the write-back pipeline, *WBPipe*. This structure enables data from the flash interface to be steered toward either the duty cycle or proximal segment update pipe. Furthermore, an additional staging register is added to forward the original boost factor during its update phase to allow for the new boost value to be rejected in the event that active duty cycle is above threshold. Sequencing is controlled by the SSD controller, allowing data to be sourced from either of the aforementioned pipes.

it possible to eliminate the need for dividers and also reduced the number of multipliers to one per write-back pipeline. Although, this could be improved further by using an out-of-order execution core, capable of issuing commands from a fixed number of agents to a fixed set of resources. Wavefront allocators have been employed in the literature to provide effective resource management in hardware architectures.

\[ y_1 = (\tau_D - 1)/\tau_D \]  

\[ y_2 = 1/\tau_D \]  

\[ y_3 = (1/\tilde{D}_A)(1 - \beta_{max}) \]
Figure 3.15: The duty cycle and boost update pipeline is a three phase ALU that utilizes control from the SSD controller to select arguments for the multiplication and accumulation phases. State update is concluded by writing the data back into a corresponding register. Control signals, $ps[3:0]$, are forwarded to each phase in the pipe to continue the multi-phase calculation. Input to the ALU, $d_{in}$, is sourced from the memory, and from the DRAM ($da_{in}$ and $do_{in}$). Data sourced from DRAM correspond to the overlap and activity previously calculated for the column being processed.

\[ y_4 = \beta_{max} \]  \hspace{1cm} (3.15d)

\[ y_5 = P_{th}/10 \]  \hspace{1cm} (3.15e)

\[ y_6 = P_{inc} \]  \hspace{1cm} (3.15f)

\[ y_7 = P_{dec} \]  \hspace{1cm} (3.15g)
Figure 3.16: The segment update data path portion of *WBPipe*. A synaptic permanence is presented at the input, $d_{in}$, and incremented by a constant corresponding to column learning state–dependent upon the overlap duty cycle and columnar activity. An operand select, $oprsel[2:0]$ is used to identify which of the configuration terms (3.15) should be applied to the segment value.Operand select is equivalent to the concatenation of learning phase state information relevant to the current segment and its synapse; $oprsel[2:0] <= camhit \& (D_s \geq \bar{D}_o) \& X_t[j]$.

\[ y_8 = P_{inc} + P_{th}/10 \]  \hspace{1cm} (3.15h)

\[ y_9 = P_{dec} + P_{th}/10 \]  \hspace{1cm} (3.15i)
Chapter 4

Results & Analysis

4.1 Math Model Classification Results

A simplified mathematical model for SP, has been presented. The SP was paired with a linear SVM and evaluated against MNIST. Networks with a differing number of columns were trained over five epochs and subsequently tested at the end of each epoch; classification accuracies were averaged across five runs. Boost factor, execution time, and columnar activation were measured for each iteration.

The complete MNIST training set was utilized to train proximal segment state for the spatial pooler. Image pixel values were pre-thresholded to suit the binary input requirement for the SP network. Given that an image pixel is represented as an unsigned 8-bit gray scale intensity, the thresholding boundary was set at half the maximum, rounded up to 128. Each thresholded training sample was presented to SP as the input region with learning enabled. This process was repeated for five epochs. At the end of each epoch, all of the 50,000 training and 10,000 test samples were fed into the SP again with learning disabled to generate output activity vectors corresponding to each image sample. These columnar activation outputs were then written to a file in the libsvm-compliant format. A linear SVM was trained to model the SP outputs using svm-train; svm-predict was utilized to test the model against both the training and test data.

Parameters were selected for the SP model with several core goals in mind. First, trivial representations were seen as undesirable, namely all or zero columns being in an active state. Stability in the boost factor indicated
Table 4.1: The constant parameters selected for the spatial pooler model.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Overlap</td>
<td>8</td>
</tr>
<tr>
<td>Desired Local Activity</td>
<td>0.11*K</td>
</tr>
<tr>
<td>Permanence Increment</td>
<td>0.001</td>
</tr>
<tr>
<td>Permanence Decrement</td>
<td>0.002</td>
</tr>
<tr>
<td>Permanence Threshold</td>
<td>0.5</td>
</tr>
<tr>
<td>Permanence Sigma</td>
<td>0.05</td>
</tr>
<tr>
<td>Maximum Boost Factor</td>
<td>3</td>
</tr>
<tr>
<td>Duty Cycle Period</td>
<td>50,000</td>
</tr>
<tr>
<td>Minimum Active Duty Cycle</td>
<td>0.2</td>
</tr>
<tr>
<td>Minimum Overlap Duty Cycle</td>
<td>2</td>
</tr>
</tbody>
</table>

that the network as a whole was tending toward a stable state. Fair utilization of the complete set of columns was also seen as a desirable quality, because this reduces the number of atrophied columns. Finally, good classification performance was requisite.

All simulation parameters were kept constant, save for the number of columns, $K$, which was varied between 49, 98, 196, 392, and 784. Table 4.1 contains key SP model parameters in addition to their corresponding values. Minimum overlap was selected to be the ceiling of 10% of the number of bits in the input space (784). Desired local activity, an inhibition parameter, was defined using a ratio, that would determine the percentage of columns active. For these simulations, this ratio was fixed at 11%, as seen in Figs. 4.1d, 4.2d, 4.3d, 4.4d and 4.5d. Given a good set of parameters, the percentage of columns active begins at zero, fluctuates during the beginning of epoch one, quickly saturating at the upper bound–desired local activity. Under undesirable configurations, the percentage active may continue to fluctuate at later epochs of the training process.

Permanence increment and decrement were set at arbitrarily small values, and the permanence sigma value was selected to be a fraction of the threshold. This sigma value was utilized for initializing the permanence values for all synapses on the proximal segment of each column. A normal random distribution, centered around the permanence threshold was sampled to initialize each synaptic permanence.
(a) Maximum, average, and minimum boost factor across all columns for each input sample, over five epochs.

(b) Simulation execution time, for each iteration of the spatial pooling process with learning enabled.

(c) The number of times each column was in the active state, placed next to the expected average, based upon the desired local activity, with the actual average shown. This demonstrates that all columns participate in representation.

(d) The percentage of columns active (influenced by desired local activity) starts off at zero and quickly rises to its limit. In a poor configuration, the percentage active may drop back to zero, yielding trivial representations.

Figure 4.1: Change in network parameters over time for SP with number of columns, $K$, equal to 49 for one of the five runs. Dashed vertical lines in figures indicate epoch boundaries. This data demonstrates that the SP is operating in a desirable manner—(1) all columns are participating; (2) the network’s parameters are stable; (3) and trivial representations are avoided. Learning, a matrix-vector operation, consumes the majority of the simulation time, and is relatively consistent.
(a) Maximum, average, and minimum boost factor across all columns for each input sample, over five epochs.

(b) Simulation execution time, for each iteration of the spatial pooling process with learning enabled.

(c) The number of times each column was in the active state, placed next to the expected average, based upon the desired local activity, with the actual average shown. This demonstrates that all columns participate in representation.

(d) The percentage of columns active (influenced by desired local activity) starts off at zero and quickly rises to its limit. In a poor configuration, the percentage active may drop back to zero, yielding trivial representations.

Figure 4.2: Change in network parameters over time for SP with number of columns, $K$, equal to 98 for one of the five runs.
(a) Maximum, average, and minimum boost factor across all columns for each input sample, over five epochs.

(b) Simulation execution time, for each iteration of the spatial pooling process with learning enabled.

(c) The number of times each column was in the active state, placed next to the expected average, based upon the desired local activity, with the actual average shown. This demonstrates that all columns participate in representation.

(d) The percentage of columns active (influenced by desired local activity) starts off at zero and quickly rises to its limit. In a poor configuration, the percentage active may drop back to zero, yielding trivial representations.

Figure 4.3: Change in network parameters over time for SP with number of columns, $K$, equal to 196 for one of the five runs.
(a) Maximum, average, and minimum boost factor across all columns for each input sample, over five epochs.

(b) Simulation execution time, for each iteration of the spatial pooling process with learning enabled.

(c) The number of times each column was in the active state, placed next to the expected average, based upon the desired local activity, with the actual average shown. This demonstrates that all columns participate in representation.

(d) The percentage of columns active (influenced by desired local activity) starts off at zero and quickly rises to its limit. In a poor configuration, the percentage active may drop back to zero, yielding trivial representations.

Figure 4.4: Change in network parameters over time for SP with number of columns, $K$, equal to 392 for one of the five runs.
(a) Maximum, average, and minimum boost factor across all columns for each input sample, over five epochs.

(b) Simulation execution time, for each iteration of the spatial pooling process with learning enabled.

(c) The number of times each column was in the active state, placed next to the expected average, based upon the desired local activity, with the actual average shown. This demonstrates that all columns participate in representation.

(d) The percentage of columns active (influenced by desired local activity) starts off at zero and quickly rises to its limit. In a poor configuration, the percentage active may drop back to zero, yielding trivial representations.

Figure 4.5: Change in network parameters over time for SP with number of columns, K, equal to 784 for one of the five runs.
Figure 4.6: Classification results for the SP; each data point is provided in Table 4.2.

Various maximum boost factor values were explored before finally selecting the value found in Table 4.1. For boost factor values above unity, network behavior was consistently comparable. Change in boost factor over multiple iterations is a function of the duty cycle parameters, with maximum boost making no significant contribution. The duty cycle period, $\tau_D$, determined the boost factor resolution. Plots depicting the trend in boost factor for each of the five network sizes are found in Figs. 4.1a, 4.2a, 4.3a, 4.4a, and 4.5a. Varying boost factor improves the involvement of individual columns in representing the input space; columnar activity, separated per column, is depicted in Figs. 4.1c, 4.2c, 4.3c, 4.4c, and 4.5c.

Results for SP were obtained by simulating the mathematical model using a single thread process on a fixed platform. Execution time required to simulate five epochs across five unique, randomly initialized networks grew from approximately two hours ($K = 49$) to about a week ($K = 784$). The Matlab model was not optimized for speed, but instead is a demonstration of the functional fidelity of the model. Run times for each of the various data points are presented in Figs. 4.1b, 4.2b, 4.3b, 4.4b, and 4.5b.

Classification performance, shown in Fig. 4.6 and Table 4.2, was found to reach near optimality after the first epoch, following the law of diminishing returns for subsequent epochs. The impact of training for additional epochs appeared to play a more significant role in networks with fewer columns. This suggests that an SP-SVM configuration may be trained with a single SP training epoch for larger network sizes, while still achieving
Table 4.2: Per-epoch SP-to-SVM classification results for training data (left) and test data (right), averaged across five runs. Results were obtained for networks with varied number of columns, \( K \).

<table>
<thead>
<tr>
<th>( K )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>( K )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>53.21</td>
<td>53.21</td>
<td>53.42</td>
<td>56.45</td>
<td>61.05</td>
<td>49</td>
<td>54.17</td>
<td>54.22</td>
<td>54.32</td>
<td>57.79</td>
<td>62.00</td>
</tr>
<tr>
<td>98</td>
<td>71.87</td>
<td>71.86</td>
<td>73.68</td>
<td>76.86</td>
<td>79.77</td>
<td>98</td>
<td>72.20</td>
<td>72.15</td>
<td>73.80</td>
<td>77.08</td>
<td>80.13</td>
</tr>
<tr>
<td>196</td>
<td>84.18</td>
<td>84.17</td>
<td>85.62</td>
<td>87.18</td>
<td>88.63</td>
<td>196</td>
<td>82.65</td>
<td>82.55</td>
<td>84.27</td>
<td>85.43</td>
<td>87.03</td>
</tr>
<tr>
<td>392</td>
<td>93.00</td>
<td>92.96</td>
<td>93.78</td>
<td>94.55</td>
<td>95.20</td>
<td>392</td>
<td>87.45</td>
<td>87.48</td>
<td>88.30</td>
<td>89.36</td>
<td>90.42</td>
</tr>
<tr>
<td>784</td>
<td>98.81</td>
<td>98.80</td>
<td>98.98</td>
<td>99.00</td>
<td>99.17</td>
<td>784</td>
<td>89.61</td>
<td>89.65</td>
<td>90.41</td>
<td>91.12</td>
<td>91.98</td>
</tr>
</tbody>
</table>

comparable performance. There also seems to be an upper limit on the benefits of network scaling; the improvement from 392 to 784 columns was significantly smaller than that observed between 49 and 98 columns.

4.2 Transaction-level Model

4.2.1 Timing Estimates

Data storage applications are very timing-sensitive, and are significantly impacted by even slight variations in latency and bandwidth. To be considered robust, an SPU must contribute a negligible amount of latency to the mainline data path—this places a hard constraint on SPU complexity. Because spatial pooling is capable of being parallelized and pipelined to a significant degree, additional delay may be hidden. The latency incurred in the read path may be generally defined as

\[
L_{RP} = t_D + L_R + L_T + L_{LDPC} + K(S-1)L_{FHTM}^1 + L_{FHTM}^2 \quad (4.1)
\]

where \( t_D \) is the transmission delay for the PCIe packet as defined in (3.13); \( L_R \) is the time taken to decode the packet and read physical NAND; \( L_T \) is the transmission time of the read data between NAND target and channel controller; \( L_{LDPC} \) is the latency incurred by having the data flow through the LDPC unit; \( L_{FHTM}^i \) is the latency due to data passing through the SPU
core towards the SSD output, and it is divided into two phases, namely the
gather phase (overlap calculation) and the output phase (inhibition and ar-
bitration); $S$ is the number of synapses on the column’s proximal segment;
and $K$ is the number of columns.

Notwithstanding the latency incurred by passing data through the FHTM
pipeline, the typical interface latency for SSD technology is sub 100$\mu$S on
average. The numbers assumed in this analysis are conservative estimates,
given that the state-of-the-art performs closer to 50$\mu$S for read operations.
From the perspective of timing, SSD performance is determined by three
factors—namely, the flash interface bandwidth, ECC overhead, and NAND
physical read times. For this analysis, the LDPC decoding process is taken
for granted and is only discussed here with respect to timing. Interface
bandwidth is influenced by the channel width, and transfer rate. In a typical
SSD embodiment page sizes may scale up to 64kB in size, where the LDPC
operates on 2kB at a time. The code rate, or ratio of useful (post PCIe
packet decoding) to encoded data (stored in the NAND), is determined by
the relationship

$$r = \frac{k}{n} \quad (4.2)$$

where $k$ is the number of useful data bits; $n$ is the number of bits within
the codeword. In modernity, 0.9 code rates are common in practical enter-
prise LDPC-based SSD implementations.

Data need only be processed in the FHTM pipeline at the rate that it is
received from the flash interface, such that each channel in the SSD has one
functional pipeline, because for every 2kB codeword, only 1.8kB needs to
be processed by the pipeline, providing a budget of extra cycles. This is in
addition to the fact that processing need not wait on the complete page to be
read, but may begin after the first complete codeword has been processed
by the LDPC engine. Therefore, the first chunk of data may be operated
on after $L_R + L_T + L_{LDPC}$ time has passed. Timing for read path oper-
aton over the flash interface is depicted in Fig. 4.7. After the first time
data becomes available to be read, the OVPipe may continue processing the
proximal segment until the entire page has been read, adding itself only a
Figure 4.7: $L_T$ and $L_{LDPC}$ are hidden by pipelining page reads and utilizing a pipelined LDPC unit to 100% availability of the LDPC decode functionality. Therefore, after the initial latency of $L_R + L_T + L_{LDPC}$, the channel output becomes available to the OVPipe. For every subsequent codeword, data are made available every $L_R$ seconds. Note that $t_D$ is not depicted in the figure because it was assumed to be zero in the TLM for simplicity and that the figure is not drawn to scale; $D_i$ corresponds to the period of time that the $i^{th}$ codeword is available on the flash interface; $CW_i$ is the point in time that a complete codeword of data is available to the input of OVPipe; $OVP_i$ indicates that the overlap pipeline output is available; $INH_i$ depicts the period in time after the $i^{th}$ columnar overlap has been subjected to inhibition. A new column may be processed immediately following the final $D_N$ codeword.

negligible latency, $L_{FHTM}^1$.

Given the aforementioned 2kB codeword with a 0.9 code rate results in unused cycles between each received codeword. For an 8-bit flash data interface width, this affords the FHTM pipeline an additional 200 cycles to process each codeword. Because less latency, $L_{FHTM}^1$, is incurred by the FHTM read path logic than the duration of a read operation, $L_R$, this logic makes a nominal impact upon overall system latency with the benefit of improving computational efficiency significantly without consuming processing resources in the host. FHTM RTL microarchitecture was used to
Figure 4.8: Demonstration of a read operation for an SPU featuring 8 channels, 16-bit wide data bus, and a 5 element deep inhibition queue.

define the specific latencies directly as

\[ L_{FHTM}^1 = \tau_F \]  

(4.3a)

\[ L_{FHTM}^2 \leq (1 + CN)\tau_F \]  

(4.3b)

where \( L_{FHTM}^1 \) is the latency incurred in the overlap pipeline during the accumulation phase; \( \tau_F \) is the clock cycle period—inversely proportional to the flash interface transfer frequency; \( L_{FHTM}^2 \) is the latency due to channel arbitration, boost multiplication phase, and inhibition; \( C \) is the number of SSD channels; \( N \) corresponds to the number of cycles incurred in the worst case operation of the inhibition engine.

Overlap, being a computational simple operation, requires a single cycle to compute, leading to (4.3a). Inhibition has the potential to consume much
more cycles, subject to three factors (4.3b). First overlap data is used to generate a valid data request to \textit{Charb}, which requires one cycle, as seen in Figs. 3.6 and 3.7. In the worst case scenario, all channels will make a request at the same time, causing the channel with minimum priority to wait for all other $C - 1$ channels to pass through arbitration. In reality, the average wait time is proportional to half the number of channels even in the worst case, as depicted in Fig. 4.8. Following arbitration, valid columnar activation and metadata pass through the boost phase, incurring one more cycle of latency. Finally, inhibition results in the only non-deterministic latency, being a single cycle in the best case and the inhibition queue depth, $N$, in the worst case. Therefore, it takes $((C - 1) + 1)N$ cycles to pass through the channel arbiter in the worst case, which implies that all channels sort data to the end of \textit{Inheng}.

Due to adaptive back-pressure and cycle filling, the pipeline is neither overrun, nor does it incur the worst case delay in the average case. In the scenario where \textit{Inheng} applies back-pressure to the channel arbiter, pipeline cycles may be filled with valid data to reduce delay, as described in the prior chapter. Timing for the design, depicted in Fig. 4.8, demonstrates that pipeline cycles will tend to be filled over time; \textit{Inheng} will thus apply less back-pressure. These factors are reflected in the \textit{charb\_ch\_gnt} (cycle fill) and \textit{sort\_ready} (back-pressure) signals.

### 4.2.2 Logical Memory Mapping

FHTM has the potential to represent large scale networks constrained by the memory size and network configuration. Taking model configurations into consideration, a function was derived to ascertain the potential network size. Total memory and page sizes were used as key parameters. The number of distal segments was considered to demonstrate how this model could be expanded for TM computation support; TM is explained in the appendix. Maximum network size is limited to the number of pages, because each page represents a single proximal or distal segment, as depicted in Fig. 3.4.
Number of pages within the memory can be simply defined as

$$N_{\text{pages}} = \frac{\text{Mem}_{sz}}{\text{Page}_{sz}} \quad (4.4)$$

where $\text{Mem}_{sz}$ is the size of user memory space in bytes; $\text{Page}_{sz}$ is the size of pages in bytes. Each column has a set of distal segments associated with it, all of which are connected to cells within the respective column. Furthermore, columns are assumed to only have one proximal segment assigned to them. This is used to determine the number of proximal segments as a function of memory size using the relationship

$$N_{\text{prox}} = \frac{N_{\text{pages}}}{1 + N_{\text{dist}}} \quad (4.5)$$

where $N_{\text{prox}}$ is the number of proximal segments; and $N_{\text{dist}}$ are the number of distal segments per column—equivalent to the cells per column multiplied by the number of distal segments per cell. Each column is assumed to have a single proximal segment.

Using these equations, the quantity of synapses may also be defined by noting that each proximal segment page has a fixed number of synapses three words less than the page size—the first three words of data contain the overlap and active duty cycles, in addition to the boost factor. Distal segments, assumed to be represented in a sparse format, may only represent a number of synapses less than half the page size (in words), because synapse values are stored in index-value pairs [51]. This is defined using the following system

$$N_{\text{syn}} = N_{\text{prox}} \frac{P_{sz} - 3C}{C} + N_{\text{prox}} N_{\text{dist}} \frac{P_{sz} - C}{C + C_{\text{idx}}} \quad (4.6)$$

where $N_{\text{syn}}$ is the number of synapses within the model; $P_{sz}$ is page size in bytes (flash interface width); $C$ is the number of bytes required to represent a synaptic permanence, assumed to be the flash data interface width (8bits/16bits); $C_{\text{idx}}$ is the number of bytes required to represent the cell index for the sparse distal segments. A sample plot of (4.6) is shown in Fig. 4.9.
Figure 4.9: Maximum number of synapses possible for HTM network, as a function of segment size, assuming a fixed memory size ($Mem_{sz} = 240\text{GB}$), and fixed page size ($P_{sz} = 32\text{kB}$); channel width assumed to be 16-bits; column index width, $C_{IDX}$ is assumed to be 24 bits—the number of cells that may be accessed is on the order of millions.

### 4.3 Synthesized Hardware

The RTL model was synthesized against the Taiwan Semiconductor Manufacturing Company (TSMC) 180nm design library to extract area and power data. LeonardoSpectrum was used to analyze, elaborate, and subsequently map the generic structural VHDL to a subset of TSMC018 Verilog components. A series of physical design files were generated using this tool—namely, synthesized Verilog, physical VHDL model, and the standard delay format (SDF) file. Schematics were generated corresponding to the synthesized Verilog using the Mentor Graphics Pyxis tool suite. The final result of the synthesis flow was a physical layout, created through the auto-layout tools—this flow is depicted in Fig. 4.10. Automatically-synthesized logical components were constrained to abide by the TSMC018 design rules.

TSMC has provided a synthesis library targeted at 1.8 volt applications. TSMC018, a 180nm process, supports 6 metal layers, and a single poly (silicided) layer. This process was selected for this work due to availability, with
the understanding that selecting a more modern process technology does not necessarily impact the core function of the design. Furthermore, because timing closure is not a focus of this work, specific technology node selection is not paramount. The HDL that was synthesized against TSMC018 was organized into three libraries:

1. **fhtm**—*The core library containing the unique Flash-HTM components proposed as a part of this thesis*

2. **fhtmcomm**—*The FHTM common library, containing generic components such as multiplexers, arbiters, adders, multipliers, counters, registers, and more*

3. **fhtmverif**—*The primary functional verification testbenches*
LeonardoSpectrum contains a scripting interface capable of analyzing and elaborating designs that utilize only the synthesizable subset of VHDL. First, key internal parameters were set—such as design rule check (DRC) settings. Thereafter, the TSMC018 library was loaded. Each design within the fhtmcomm and fhtm libraries were analyzed against the tsmc018_typ technology, and subsequently elaborated. At this stage, the FHTM model generic parameters could be set to meet requirements—namely, data bus widths, inhibition queue depth, and number of SSD channels. The synthesis step was concluded by writing the aforementioned output Verilog, VHDL, and SDF files. Schematics and their associated layouts were generating using Mentor Graphics Pyxis.

Typical flash controllers are implemented as a programmable SoC, making it feasible for a full implementation of FHTM to leverage much of the already existing technology. Consequently, the SSD controller model was not created nor synthesized; valid interaction with the FHTM model was handled in the testbenches. FHTM was synthesized assuming a SSD with 8 channels, taking the LDPC, host interface and SSD controller for granted.

4.3.1 Full Custom Layout & Area Estimates

Each of the FHTM components shown in Fig. 3.5 were synthesized individually to reduce synthesis time and to eventually extract independent power data. Area estimates were obtained from design dimensions and actual area consumed by physical shapes. The former is useful for gaining an intuition for the required area budget, while the latter is useful for aggregating area information for the design that is transparent of specific placement. Synthesized hardware was verified to abide by the TSMC018 DRC and also functionally qualified by conducting a layout-versus-schematic (LVS) audit. Throughout the design process, experience with the tools resulted in improved custom layout results, eventually resulting in an exploration into hierarchical synthesis and a more complete knowledge of the placement and routing workflow.

FHTM SP is divided into two portions—the read path and write-back path;
Figure 4.11: Synthesized hardware implementation of the overlap pipeline, *OVPipe*, with an 8-bit wide data bus. The component dimensions are 1920\(\mu m\) \times 1688\(\mu m\), resulting in an area of 2.310 \(\times\) 10^6\(\mu m^2\).

the former and latter being associated with the columnar activation calculation and learning state update, respectively. The read path is comprised of *OVPipe*, *Charb*, and *Inheng*; the write path is comprised of *Camharb*, *WBCam*, *Camhit* and *Camhit*. A generic synthesis process was utilized for each of these components, resulting in efficient routes under the assumption that they are all self contained, but adds additional complexity to a complete system layout, because data buses were not placed in a spatially aware manner. For example, inputs to the *OVPipe* in Fig. 4.11 corresponding to the 8-bit data bus are scattered all around each side of the abstract subject to the automatic placement from the tool.

The solution to this problem is to create designer specified abstracts for each component that require each I/O pin to be placed with respect to a set of constraints. Despite this, creating an unconstrained layout is useful
Figure 4.12: Synthesized hardware implementation of an 8-channel fixed priority arbiter, \textit{Charb}, with an 8-bit wide data bus and integrated multiplier. The component dimensions are $2624 \mu m \times 2552 \mu m$, resulting in an area of $5.745 \times 10^6 \mu m^2$.

For estimating area overhead, albeit to a limited extent. Another limitation to the manner in which the tool was used in this work is that placements followed a very simplistic row-based scheme, which required that all components be aligned regularly to simplify routing. Additionally, no custom power-routing was employed to ensure that the design could meet loading requirements; for this technology node, this did not present any issues during the parasitic extraction phase. As a work-around, power routes were routed on slightly thicker wire than standard signals, and with relative success—however, these results would require more in-depth auditing. For more modern technology nodes, routing power on such limited wire for large scale designs would result in functional failures.

\textit{OVPipe} is expected to scale well with growing network sizes, because its complexity only scales as a function of flash interface bus width. This
is a useful reality, because \textit{OVPipe} is duplicated per flash channel. Modern SSDs typically support 16-bit data buses for each channel, which allows for a sufficient amount of resolution in data received from the proximal segment. Data stored in the NAND include synaptic permanences, and duty cycle parameters. Column indexes correspond to compressed addresses—they only address individual pages, and do not constitute a byte address. The logic used to sequence input into the overlap pipeline was used to test the read path (Fig. 4.8), but was not synthesized, because it is not strictly part of \textit{OVPipe} and may potentially be represented by the flash controller.

It must be noted that the overlap pipeline width need not be limited to the flash interface width, but should be at least a multiple of it. Several beats of data from the flash interface may be aggregated together to obtain greater data resolution. This has the added benefit of reducing the clock frequency necessary for the SPU core by a factor proportional to the overlap pipeline input width divided by the flash interface width—the number of beats required to read a complete package of data. Because \textit{OVPipe} reads data when \(d_{val}\) is high, it is flexible enough to support this without any significant internal modification; a series of registers would have to be added between \textit{OVPipe} and the LDPC.

The channel arbiter (Fig. 4.12) is expected to be one of the larger modules, because it receives wide buses from multiple overlap pipelines as inputs to arbitration. \textit{Charb} also contains the boost multiplier within it, which results in a larger number of logic gates being generated. Because the overall function of the pipeline was emphasized, specific multiplier optimizations were not made. FHTM exists as a proof of concept that such a design is feasible and potentially scalable. The cost of having a multiplier within \textit{Charb} is acceptable because only one channel arbiter exists per FHTM SPU. Simple structure within \textit{Charb} facilitated high routeability, which led to the large buses of wire being routed to the bottom and right of the component. Area for \textit{Charb} is proportional to columnar index, and flash interface bus widths, and to the number of SSD channels. A majority of the area is consumed by the adder and channel selection multiplexers.

\textit{Inheng}, depicted in Fig. 4.13, is by far the most challenging component to scale because its ability to scale is dependent upon network size and it also
places a hard constraint upon which network parameters are possible. In this embodiment, the depth of $Inheng$ corresponds to the number of columns that may be active at any given time due to feedforward input—namely the desired local activity. Because this parameter is a proportion, it results in the queue size being defined by the following relation

$$Q_{sz} = N_{prox} d_{th}$$

(4.7)

where $Q_{sz}$ is the minimum inhibition queue size necessary; $N_{prox}$ is the number of proximal segments; and $d_{th}$ is the desired local activity as defined in the mathematical model. Possible network sizes available for a given queue depth may be obtained by computing the intersection of $Q_{sz}$ with a plane representing the desired queue size. All points at or below that intersection result in valid network parameters for a fixed inhibition queue
Figure 4.14: Synthesized hardware implementation of an 8-channel Camharb with 8-bit wide data bus. The dimensions are $1496\mu m \times 1424\mu m$, resulting in an area of $1.613 \times 10^6 \mu m^2$.

This methodology works fine for a network with limited number of columns, but does not scale particularly well. However, to support a larger number of columns a memory-assisted inhibition engine would be requisite. In this scheme, Inheng would be required to store the lowest $N$ columnar overlaps (Fig. 3.9). Whenever the overlap engine is filled up, its data would be transferred to the DRAM cache and stored in an in-memory table. A background process would continue the remainder of sorting on the data. Such a change would also require that WBCam be modified. These modifications were not included in the model to simplify this baseline design, because they would require a generic DRAM and SSD controller models to estimate behavior accurately.
Figure 4.15: Synthesized hardware implementation of an 5-element WBCam with 8-bit wide data bus. The dimensions are $2216\mu m \times 1928\mu m$, resulting in an area of $3.548 \times 10^6 \mu m^2$.

FHTM is the only known neuromorphic hardware architecture that implements the SP learning rule—utilizing three components to facilitate the complete proximal segment update process. Camharb, shown in Fig. 4.14, begins the learning process by arbitrating between requests issued to NAND by the SSD FFS and checking WBCam for hits. Although, in principle, Camharb is functionally similar to Charb, it requires significantly lower area. This is because it uses smaller data bus widths and does not feature a multiplier phase. The channel content addressable memory arbiter selects between buses composed of constant channel IDs, paired with column indexes associated with the request.

WBCam, shown in Fig. 4.15, is placed downstream to Camharb and processes incoming queries to the CAM. This component scales in a fashion similar to that of Inheng, because there is a one-for-one correspondence between their depths. Each entry in the WBCam is used to compare an
Figure 4.16: Synthesized hardware implementation of a 5-element, 8-channel Camhit component with 8-bit wide data bus. The dimensions are $1768\mu m \times 1592\mu m$, resulting in an area of $2.225 \times 10^6 \mu m^2$.

incoming column index against one stored within the inhibition engine. If a DRAM memory-assisted version of Inheng is used, a more complex control scheme is needed for the WBCam. In light of this, WBCam is not necessarily a vital component in this case, because if the values are already stored in DRAM, the SSD controller may compute the CAM operation without the write-back control logic (Camharb, WBCam, and Camhit). Another method could artificially extend the length of WBCam by loading all groups of data equal in size to the inhibition queue depth and process the data through WBCam multiple times. The Camhit component, presented in Fig. 4.16, is simple, only being composed of several logical or-reduce structures and equality comparators. It would require no modification in the scalability-driven design modifications.

Write-back logic was found to require the largest amount of area because it required large data bus widths, and three-stage stage ALU that supports
duty cycle, boost factor, and synapse updates. This additional complexity leads to \textit{WBPipe}, shown in Fig. 4.17, having the largest area footprint. One limitation of this design is that it requires each channel to have a dedicated multiplier. However, designs such as wavefront allocators have been employed in the literature to facilitate the allocation of \( N \) resources to \( M \) requesting agents [71]. Consequently, future HTM hardware may be able to decouple arithmetic units from \textit{WBPipe}, potentially consuming less area.

Hierarchical synthesis was explored for \textit{WBPipe} and \textit{WBCntl} to make synthesizing these larger designs more feasible. Furthermore, instantiated components were compacted to reduce area and an iterative workflow was
Figure 4.18: Synthesized hardware implementation of a WBCntl, comprised of Camharb (bottom-right), WBCam (left), and Camhit (top-right), an with 8-bit wide data bus. The dimensions are $3008\mu m \times 3280\mu m$, resulting in an area of $9.866 \times 10^6\mu m^2$.

used to ensure that DRC and LVS requirements were met. More aggressive routing strategies were required to ensure that all of the overflows between nets would be properly routed, such as wrong-direction routing, ripping and congestion analysis. Furthermore, some primary I/O pins had to be manually moved to ensure routability could be obtained. Each of the sub-components in WBPipe were multiplexers—these may be identified by inspecting Figs. 3.15 and 3.16. WBCntl (Fig. 4.18)—composed of Camharb, WBCam, and Camhit—was created to explore block-based hierarchical synthesis. The power results for each of these components were obtained by extracting the parasitic netlist of WBCntl.

Area footprints were collected from reports generated by Mentor Graphics Pyxis IC design suite and have been summarized in Fig. 4.19. The
Figure 4.19: Area (top) and dimensions (bottom) for each of the synthesized RTL components. The read and write pipe logic are expected to consume the greatest amount of area, because they are duplicated per channel, resulting in an 8x-20x increase in area for these components.

dimensions of each component are also provided. Total combined real area is obtained by taking the sum of all subsystem areas, yielding a footprint of 30.538 mm$^2$. However, the estimated area required for an actual implementation must take into account the number of times that each component is expected to be instanced in an actual SSD SPU. Estimated area is defined as

$$A_{FHTM} = N_{ch}(x_0 + x_1) + x_2 + x_3 + x_4 + x_5 + x_6$$ \hspace{1cm} (4.8)

where $N_{ch}$ is the number of channels, $x_0$ and $x_1$ are the areas for OVPipe and WBPipe, respectively; the other $x_i$ terms correspond to each of the other components. Under this assumption, the area becomes 104.26 mm$^2$.

### 4.3.2 Power Estimates

A parasitics extraction netlist was derived from the layout and evaluated using the Eldo analog simulator in conjunction with the EZWave graphical user
interface to view the corresponding power traces. Hardware created at this phase was found to consume power on the order of a few milliwatts, as depicted in Fig. 4.20. Components consuming the most power did so due to having a larger number of memory elements switching state at each clock cycle. Average power consumed was determined by calculating the mean of all power consumption data, yielding $5.171mW$. Power data was sought to establish a baseline for a hardware implementation of SP—similar data exists for other neuromorphic algorithms, as presented in Table 2.2. However, wattage per area is not presented, because this potentially creates more questions than it answers regarding methodology for data extraction. Comparing across technology nodes, implementation platforms, and for varying data workloads would not provide an honest nor useful comparison.
Chapter 5

Conclusions & Future Work

5.1 Conclusions

As our understanding of the underlying mechanisms that drive intelligence continues to expand, applications that leverage this body of knowledge will increase. It is likely that machine intelligence, applied to enterprise-scale big data analysis of unstructured data, will serve as catalyst for future innovations; leaders in the technology industry are invested in this prediction. Advances in algorithms will, in turn, lead to the creation of new architectures. Storage processors present a promising opportunity for product differentiation in this space—companies may provide robust virtual platforms capable of handling pattern recognition workloads at scale with low cost.

Through this exploration, concepts and an analysis have been presented that aid in addressing some of the design concerns regarding the implementation of an SPU spatial pooler. The impact of augmenting a storage unit with processing capabilities degrades the upfront latency for reading data, consumes additional area resources, and may potentially limit the usefulness of this storage unit for other applications. Despite this, there are many benefits to taking the SPU approach:

1. Parallelism may be more thoroughly exploited than if the algorithm were implemented by a multi-core processor

2. Vital resources within the host are conserved, allowing the storage unit to act as an accelerator at scale

3. A key challenge for storage elements is I/O availability—an SPU does
not require any additional I/O

4. Memory-mapped configuration facilitates the design of a microarchitecture that is simple to implement, configure, and extend

5. An in-path SPU may be sold as locked IP that may be dropped into other SoC environments

6. Significant power savings are obtained over in-host processing, which operates at an order of magnitude higher frequency

7. Scalability of the design is significantly improved over external accelerator designs, which are bottlenecked by memory bandwidth limitations

A clear explanation of the model utilized to implement the HTM spatial pooler has been presented, providing clear insight into some of the design limitations and benefits. SP, originally developed with high-level programming in mind features significant branching behavior and unclear parameter limits. Despite these challenges, a simplified hardware model has been presented along with an accompanying system model. The use of TLM may have a positive impact on system architecture research, improving design productivity—the primary challenge being overcoming the learning curve.

Power, area, and latency estimates were extracted from each phase of the design process to acquire a baseline for feasibility analysis. In conclusion, the large delays characteristic of SSDs (when compared to main memory or cache) mean that the latency added by the FHTM pipeline is orders of magnitude less than the baseline latency for a standard SSD. This supports the case for employing storage processor units in hardware, as has been discussed in the literature. However, the specialization required by the hardware limits the potential for deploying this to large scale markets. Implementing the SPU functionality on a reconfigurable platform integrated into the SSD would be an attractive option. FHTM, paired with SVM for classification, present results comparable to those found within the literature for HTM MNIST (91.98%).
5.2 Future Work

Prior art in the body of literature has addressed some of the potential areas of interest for HTM, some of which have been addressed within this work. FHTM is the first scalability-centric HTM SP hardware microarchitecture that leverages non-volatile storage to enable in-memory computations. Because HTM requires significant quantities of memory, an SPU-enabled architecture is an appropriate medium. For modern SSDs, page sizes may reach scales of 64kB, allowing HTM to scale up significantly. A large-scale simulation for HTM hardware has yet to be conducted, but may be premature at this point due to volatility of the algorithmic framework.

A functional TLM was presented to model system behavior. The TLM methodology should be applied to create a detailed latency model to extract more fine-grained timing parameters than are discussed here. Additionally, as the HTM paradigm continues to develop, new hardware implementations should be explored. At the time that this research project was conducted, significant changes were made to HTM, which now includes the distinction between temporal memory and temporal pooling. Other work geared towards testing HTM against other datasets is also necessary. Numenta Anomaly Benchmark (NAB) has recently emerged and should be used as a baseline for future implementations of HTM. Before any of these efforts are pursued, the algorithm must be more thoroughly characterized as it goes through new iterations. Exploring a hardware implementation of Zeta and subsequently comparing the results of Zeta with CLA, and the emerging model is also of great value—to establish a case for using one over the other.

Growth in this research space must be preceded by success on the algorithmic front–HTM has yet to surpass state of the art machine learning algorithms such as deep belief networks (DBN) and convolution neural networks (CNN). Further hardware implementations are admittedly premature, given the lack of mathematical concreteness present in this space. Temporal memory also requires a more concrete mathematical formulation—at the time that this research was conducted, this was not present in the literature–some preliminary work in this path has been discussed in Appendix A. Ultimately,
knowledge regarding HTM needs to be more accessible to the research community. Recent work has been centered around addressing some of these challenges. This will lead to greater accountability and improved merit of works published in this space.

FHTM is the first full custom hardware design for HTM SP that implements the learning phase. One area ripe for future exploration is to the effect of data resolution upon results, which has yet to be discussed in the context of HTM CLA. This will help provide necessary insight to hardware designers regarding error limits for the spatial pooler. Furthermore, future hardware implementations must address the issue of overflow and saturation, because the permanence values tended towards saturation in the mathematical model. If this is not fully accounted for, hardware failure points will be reached–this is particularly important for multiplication. In addition to this, a more rigorous treatment of parameter selection for the spatial pooler must be addressed. A methodology that defines a concrete strategy for parameter selection must be provided to improve the case for a hardware implementation. Addressing these issues will enable a robust architecture to be developed that is guaranteed to be capable of successful deployment in a wide variety of applications under known preconditions.

Fault-tolerance must also be explored in a future embodiment such that the inevitable failure of flash pages does not result in the corruption of columnar data. FHTM also assumes that each column spans an entire page–this design decision was made for simplicity. However, it is advantageous for a new model to be defined that allows for proximal segments to be smaller than the page size, to allow for larger networks to be created; granted, the microarchitecture is not explicitly limited by this fact.
Bibliography


Appendix A

Temporal Memory

A.1 Temporal Memory Algorithm

Temporal memory (TM) is designed to model the inter-pattern association across various time steps. In many other machine intelligence algorithms, temporal association is obtained by training models designed to handle static inputs on a time window of data. For instance, to train a support vector machine (SVM) to classify audio signals (inherently temporal), the audio signal may be sampled in 10 ms chunks—these chunks may be used as inputs to a standard feedforward network. Although this methodology does work to some extent, it is not inherently time-based. Temporal memory was designed with the intention of going beyond this by also modeling time-sequences.

The first phase of the temporal learning process begins by determining which cells will become active or learning (not mutually exclusive). Cellular activity is determined by selecting cells within active columns. Recall that the output from SP is the set of active columns and that functionally-similar cells are associated with columns. Every cell of each column is checked to see if it was predicting activity in the current column at the prior time-step. If it was, then the segment that would most likely have led to that prediction is selected to be learning and the cell is indicated as being active.

Segments, are groupings of associated synapses. If no cell within an active column is selected to be active, all of the cells within the column will become active—this is the bursting state for the column. When no cell is placed into the learning state by this point, the most suited cell will be
Phase two of TM identifies the cells that will predict output for the next time step. This process checks all segments of every cell for all columns and sets the predictive state to true if the cell was previously a predicting cell. Cells only become predictive if they were previously predicted. Further synaptic updates are staged to be committed in the final step. However, these updates are non-creative and as such will not create new synapses nor segments (Fig. A.2).

Finally, the algorithm terminates by committing the synapse updates. Synapses are positively reinforced if the cell is continuing to predict between time steps, but negatively reinforces learning if the cell was previously predictive and ceased prediction during the current time step. This
Figure A.2: The second phase of the temporal learning process, namely the predictive state calculation [50]. Temporal memory output is a concatenation of the activity and predictive state, a 2-dimensional matrix, $P$.

seems to be a typo in the white paper, as this implies that the learning objective for the cells is for them to always be in the predictive state at each time step. It must be emphasized here that SP operates on a network with a fixed number of proximal synapses, while TM takes a network initialized to have no distal synaptic connections and adds synapses and segments as needed. Consequently, strict memory bounds must be placed on the algorithm or it will create distal segments and synapses in seemingly unrestricted manner. As noise is introduced into the network, memory requirements also increase.

A.2 Mathematical Model

A TM model was derived in like manner to the SP. Temporal learning is composed of three steps, the first of which being a determination of the
set of cells in the active state, followed by the predictive state calculation, and finally a synapse update commit phase. Constraints were placed on the model so that it would not consume an unbounded amount of memory, but instead a trimming rule was considered. All places in which modifications were made are clearly indicated and reasons for these changes are also provided.

**A.2.1 Phase I: Active State**

Temporal memory begins by determining the cellular activation; for this calculation, the symbol, $\alpha$, previously applied to the overlap, will be borrowed here and from henceforth will refer to the cellular activity. Recall that SP operates at the granularity of columns, and TM at the cellular level—the former being comprised of the latter. After spatial pooling, an activity vector $A_i$ is produced, containing the activations of $N$ columns, (3.6). An element-wise
multiplication of the columnar activity vector with a predictive state matrix is calculated as

$$\alpha^j_i = A_i \odot P^j_i$$  \hspace{1cm} (A.1)

where $P^j_i$ is the (binary) prediction state matrix, calculated at the prior time step; $\alpha^j_i$ is a state matrix, indicating whether the $j^{th}$ cell within the $i^{th}$ column was previously in the predictive state and is currently within an active column. Cells meeting these requirements become active. Note that in the white paper this point is not clear, because the explanation is worded this way, but the pseudocode does not emulate this [50].

Another feature of the first phase of the TM must be implemented before it is considered functional, namely bursting. An active column enters the bursting state if none of its cells become active due to the combination of feedforward input and the prediction state. This condition is described by a sum of cellular activation states within a column such that

$$B_i = \sum_{i=0}^{J-1} \alpha^j_i$$  \hspace{1cm} (A.2)

where $J$ is the number of cells per column; $\alpha^j_i$, as was previously mentioned, is the cellular state without bursting accounted for; $B_i$ is used to indicate the bursting state for a given column. A value of 1 anywhere within the binary predictive state matrix corresponds to an active cell. Cells enter a state of activity if they were predicting feedforward input in the prior time step and the column that they are present in became active during the current time step. Cells may also enter this state due to bursting.

If the aforementioned summation equals zero, this indicates that no cells within the active column entered a state of activity due to feedforward input. Consequently, (A.1) is extended to

$$\chi^j_i = \begin{cases} 1, & B^j_i = 0, A_i = 1 \\ \alpha^j_i, & otherwise \end{cases}$$  \hspace{1cm} (A.3)
where $\lambda^j_i$ is the cellular state with bursting accounted for; $A_i$ indicates that the column is currently active due to feedforward input from the SP.

The remainder of phase one was removed or relocated to phase three. TM implements certain aspects of learning in the midst of both primary and secondary phases, but this was modified for clarity and to make the model match its description more clearly. Active cellular state information is the output from TM phase one.

### A.2.2 Phase II: Predictive State

Determination of the predictive state of each cell for the current time step is the focus of the second phase of the TM algorithm. This process is initiated by calculating the segment activity for each cell

$$\hat{S}^j_i = S^j_i \geq P_{th} \quad (A.4)$$

where each $S^j_i$ is a different sparse 2D matrix of permanences for the $j^{th}$ cell within the $i^{th}$ column; each row in the $\hat{S}^j_i$ matrix corresponds to the $\rho^{th}$ segment, where each column represents a unique cell within the TM network; and $P_{th}$ is the permanence threshold. This structure is defined to make the following calculation of the activation for each segment less complex to characterize.

After determining the synaptic connectivity, the segment activity is ascertained by calculating the dot product of each segment matrix with the cell state matrix. The diagonal of the resultant matrix, if summed results in the segment activity—calculated as follows:

$$\gamma^j_i[\rho] = \sum_{c=0}^{J*K-1} (\text{diag}[\hat{S}^j_i[\rho] \cdot \lambda^j_i]c) \quad (A.5)$$

where $\gamma^j_i$ is the integer sum, representing the number of active-connected synapses on the distal segment; $\text{diag}[\hat{S}^j_i[\rho] \cdot \lambda^j_i]c$ is the $c^{th}$ diagonal value (top left is the $c^{th}$ diagonal) of the active synapse matrix; and $\rho$ is the index of the current distal segment, connected to the $j^{th}$ cell within the $i^{th}$ column.
Each segment (a row) within the matrix maps the 2D cellular state space into a single vector. Consequently, to make a multiplication of the segment state matrix with the cellular state, the latter must be remapped as well to be a single vector, in which each subsequent column occurs after the one preceding it. The calculation of interest here is to take the dot product of the \( \rho^{th} \) segment vector and the cell state.

Finally, the predictive state is calculated as output from the second phase. This results in an in-place modification of the prediction state matrix such that

\[
P_{i}^{j*} = \begin{cases} 
1, & \exists \rho \in \gamma_{i}^{j}[\rho] \geq S_{th} \\
0, & \text{otherwise}
\end{cases} 
\]  

(A.6)

where \( P_{i}^{j*} \) is the next value for the prediction state matrix; \( \rho \) is the segment indexing variable; \( S_{th} \) is the segment activation threshold—the number of active-connected synapses required to make the segment active; the activation threshold is analogous to the SP minimum overlap. The condition found in (A.6), \( \exists \rho \in (\gamma_{i}^{j}[\rho] \geq S_{th}) \), forms the set of indexes for segments that were above the activation threshold.

### A.2.3 Phase III: Synapse Update

If the learning mode for the temporal memory is enabled, then permanences may be trained or new distal segments may be created. The learning process was simplified from the official TM formulation by restricting all of its steps to occur within one particular phase. This drastically simplifies control for the TM, making it more feasible to implement in hardware. Learning begins by determining the subset of cells that are qualified to enter the learning phase. One cell per column is selected to learn at each iteration of the algorithm. The one exception to this is bursting, because when this condition occurs, all cells within the column learn.

The selection process begins by calculating the maximum segment activation due to feed-forward input. Distal segments with the largest number
of active-connected synapses will be selected to represent the cellular activation. Within active columns, cells with the largest activation are chosen to enter the learning state. The maximum segment activation is calculated as

$$\psi_j^i = \max(\gamma_j^i)$$ \hspace{1cm} (A.7)

where $$\psi_j^i$$ is the maximum segment activity for the $$j^{th}$$ cell on the $$i^{th}$$ column—the number of active connected synapses, which is defined in a similar manner as in SP; $$\gamma_j^i$$ is the vector containing the number of active connected synapses for each segment. An index is used to determine which distal segment resulted in the largest activation using the following relationship:

$$\hat{\rho}_j^i = \left\{ \rho | \gamma_j^i[\rho] = \psi_j^i \right\}$$ \hspace{1cm} (A.8)

where $$\hat{\rho}_j^i$$ is the index of the maximum segment within the sparse segment matrix—corresponding to a row in the matrix; $$\rho$$ is the index of the segment in question; the condition is used to specifically select the segment with the maximum activation. For implementation simplicity, the first identified max segment is selected. This means that if two segments share the same activation and the value thereof happens to also be the maximum, the first of the two (having the lowest index) is selected.

Next, each segment activation is associated with the source column. This results in the creation of a new matrix with the same dimensionality as the activity state matrix, $$\lambda_j^i$$, such that

$$L_j^i = \begin{cases} \psi_j^i, & \lambda_j^i = 1 \\ 0, & \text{otherwise} \end{cases}$$ \hspace{1cm} (A.9)

where every value within $$\psi_j^i$$ is the respective maximum segment activation for the $$j^{th}$$ cell of the $$i^{th}$$ column. This value is only calculated for currently active cells.

Using the new maximum segment activation matrix, the cell with the largest activity within each column is selected to be the chosen cell to enter
the learning state. This is a deviation from the pseudo-code within the white paper, but captures the high-level concept behind TM learning—this model is more closely aligned with the more recent version of TM [72]—however, this similarity is coincidental. The max activation for each column is the vector defined as

$$\hat{L}_i = \text{max}(L^j_i) \quad (A.10)$$

Knowing the maximum activation within each column leads to the identification of the index of the cell that yielded that result. The maximum index, $j_{i}^{\text{max}}$, is defined as

$$j_{i}^{\text{max}} = \{ j | L^j_i = \hat{L}_i \} \quad (A.11)$$

where $j$ identifies each cell within the $i^{th}$ column. Inactive columns will have a max value and $j_{i}^{\text{max}}$ value of zero, because all cells in that column have zero activation and the first largest value is always selected. Given the information up to this point, we now have a methodology for identifying learning candidates. Only one cell per active column is selected to learn and this selection is determined by the cell with the most significantly active segment connection. There are two learning rules—one that creates segments and synapses, and another that only updates existing synaptic connections on distal segments. The latter of the two is calculated as

$$\delta S_i^j [\hat{\rho}_i^j] = (P_{\text{inc}} + P_{\text{dec}}) \gamma_i^j [\hat{\rho}_i^j] - P_{\text{dec}} \quad (A.12)$$

where $\hat{j}$ is an alias for $j_{i}^{\text{max}}$ (to make the equation more easily readable); $\delta S_i^j [\hat{\rho}_i^j]$ is the segment permanence update—it is calculated such that all values will be incremented by $P_{\text{inc}}$ or decremented by $P_{\text{dec}}$; $\hat{\rho}_i^j$ is the index of the optimal segment for the $\hat{j}^{th}$ (best) cell within the $i^{th}$ active column. This delta is used to increase the permanences of synapses that were connected to active cells, and to decrement the permanences that were connected to inactive cells. Synaptic updates are committed to the best distal segment by
adding this value to the current value for the segment’s synapses

\[ S^j_i[\hat{\rho}^j_i] = S^j_i[\hat{\rho}^j_i] + \delta S^j_i[\hat{\rho}^j_i] \quad (A.13) \]

Because cells are initialized without any synaptic connections, a learning rule exists in TM to dynamically generate distal segments and add synaptic connections to them. This operation only occurs when the column enters the bursting state. To emulate this, upon entering this state, the best cell within that column will also generate a new segment and connect it to a subset of the active cells. Generation of new distal segments only occurs if there is space available—this enforces a memory bound upon the temporal memory.

Recall that bursting is indicated by the variable \( B^j_i \). All columns meeting this criteria have new segments generated for them until the maximum amount is reached, each of which is connected to a subset of the set of active cells. A pseudo-random scheme is utilized to determine which cells to connect to. Connections that fall below a threshold are pruned.

**A.3 Hardware Model**

Temporal memory is significantly more complex than SP with regards to computational complexity. On the order of \( J \times N \) more calculations are required for TM over SP—where \( J \) is the number of cells per column and \( N \) is the average number of distal segments. Another challenge to modeling TM is the unbounded segment growth. To manage the latter concern, a hard limit was placed upon the number of segments per cell. This allows enough flexibility to represent similarly complex network dynamics as an unbounded model with the added benefit of being able to make key assumptions about the memory organization (Fig. 3.4).

Cellular activity is a function of the prior predictive state and the current input. A cell enters the active state if the containing column made it through the inhibition phase, and the cell previously predicted this behavior. Prior cellular predictive state data are loaded into the active state pipeline, gated by the associated columnar activity. The bursting condition, \( B_i \), is determined to be the logical or-reduction of the predictive states of all cells
Figure A.4: The first task of TM is to ascertain the subset of cells that will enter the active predictive state based upon the current feedforward activation, $A_i$, and the prior predictive cellular state, $P_{i}^{j}$. Intermediate variables, are also indicated–namely, $\alpha_i$, and $B_i$.

within the column. In parallel, the pre-bursting cellular activity, $\alpha_i$, is then combined with the active state and the bursting condition to determine the final state for the cell.

In the event that a column is in bursting mode—none of the cells on an active column responded to the feedforward input—all of its respective cells enter the active state. Otherwise, only a subset of the cells exhibit activity. Output from SP, $A_i$, serves as the columnar activity and is extended to contain one bit for every cell within the column. Active state, $\lambda_i$, is calculated as the bit-wise OR of the two asymmetric (with respect to $B_i$) inputs by the pipeline depicted in Fig. A.4.

The second phase of TM is to calculate a prediction for the next expected output state for the region. Distal segments are sourced from the NAND using a series of page read operations. Synapses on distal segments are stored in sparse format, resulting in each being represented as a value-index pair. Consequently, each permanence, $S_{i}^{j}[\rho, x]$, and its corresponding index, $x$, are loaded into the pipeline in subsequent cycles as indicated in
Figure A.5: Phase one of the segment activation, $\gamma_j^i[\rho]$, calculation. Distal synapses are compared against the permanence threshold, $P_{th}$, and combined via an AND-gate to determine if each of them are also connected to an active-predictive cell; $S_j^i[\rho, x]$ corresponds to magnitude for synapse $x$ on the segment $\rho$, within cell $j$ of column $i$. Cellular activations, $\lambda_i$, are looked up from a DRAM table, using the distal synapse index, $x$.

Fig. A.5. Synaptic activity is determined as the product between thresholded permanence and the activity, $\lambda_i$, corresponding to the cell connected to that synapse.

Acquire the associated cellular activation required for the predictive state calculate necessitated that a lookup operation be conducted on the DRAM table. At each index are cellular activations grouped by column, $\lambda_i$. Specific $\lambda_i$ values are selected by using a modulo operation, using the index assigned to the synapse. Outputs from this preliminary phase of the predictive state update unit are passed, at every other cycle, to an accumulator, depicted in Fig. A.6. After the entire distal segment page has been gathered, its activation, $\gamma_j^i[\rho*]$ is compared against the segment activity threshold, $S_{th}$.

Using the cellular index, $j$, the activation is expanded into a vector of width corresponding to the number of cells per column. For instance, if the active distal segment in question is on the second cell within the given column, output of the expand unit would be “0100”, assuming that $J = 4$. This vector is then combined with the $P_j^*$ vector using an OR operation. A consequence of this is that if any distal segment corresponding to a cell
Figure A.6: The predictive state update and final step of the gamma calculation. Distal synapse state, $S_{th}[\rho, x]$, is accumulated to calculate the segment activation, $\gamma_j^i[\rho]$. If the segment activation is above a threshold, $S_{th}$, the cell is indicated as being in the predictive state for the next time step. In the event that learning is enabled the maximum segment values will be stored in a cached table, containing the maximum segment activation, $\gamma_i[\rho_i]$, corresponding to the cell, and its respective segment index.

is above threshold, it will enter the predictive state. At the beginning of subsequent time steps the predictive state matrix, $P$, is reset to zero. This step is conducted immediately following the active state calculation, which depends upon the prior predictive state.

In Fig. A.6 work corresponding to the learning mode calculation are also depicted. The process of training the network requires that an optimal segment for each cell be selected as a candidate for learning. As distal segments are read from memory, each activation magnitude is compared against the prior maximum value to determine which should be saved. This corresponds to the calculation for maximum segment activation, $\gamma_i^j$.

Because it is calculated in this way, once $\gamma_i^j$ is known, the maximum segment activation for each cell, accounting for cellular activity, $L_i$, is also
known. This is because only cells within the active state have learning conducted on them, so this step is trivial, resulting in the simplification $\gamma_i = \mathcal{L}_i$. Segment activations are stored with their index so that they may be retrieved later for the segment update portion of the learning phase. All of the aforementioned tables are stored within the DRAM for quick access.

Temporal memory learning is conducted using principles similar to those found in the proximal segment write-back phase. The primary change is that a specific segment, corresponding to one cell within the column, having the largest segment activation, $\mathcal{L}_i$, is selected using a comparator tree; although this could be done iteratively to improve design flexibility and reduce area in a physical implementation. The index, $j_i^{\text{max}}$, of cell with maximal segment

---

**Figure A.7:** After all of the segment activations are calculated, the best candidate cells for learning are selected—defined as the cell with a maximum segment activation threshold greater in magnitude than that of other cells sharing on a common column. The maximum segment activation, and its corresponding column are $\hat{\mathcal{L}}_i$ and $j_i^{\text{max}}$, respectively. This figure demonstrates the selection of a cell, given a maximum configuration of four cells per column; two cells per column is a common configuration presented in the literature.
Figure A.8: Learning is concluded by writing the distal segments back to memory in a fashion similar to that of SP. In learning mode, two reads, and one write operation are incurred, because the first read calculations the TM state required to influence the learning phase. The additional read and write operations are required to retrieve the distal segment and to subsequently write the updated state back to memory.

activation and the corresponding activity, $\hat{L}_i$, are the output from this phase–depicted in Fig. A.7. Subsequently, the page associated with all learning distal segments are read out from memory, updated, and written back in similar fashion to the proximal segment write-back phase (Fig. A.8).

A.4 Temporal Memory Prediction

Testing the TM model required a more involved process than SP. The temporal memory was trained to predict sequences of zero through nine. At each time step, TM would be presented a new MNIST digit of value one greater than the prior. Sequences would wrap around at nine, such that nine would predict zero. Because MNIST was not designed to be capable of this, the following testing methodology was used:

1. Each MNIST image in the training and test sets was fed into the previously trained SP model to generate a columnar activation, $C^i_K$.

2. $C^i_K$ vectors were then divided into label-sorted groups—all zeros were placed in one set, all ones in another, etc. This process was continued until all training and test vectors were categorized.
3. The length of the smallest set was used to define the number of complete test sequences.

4. With learning enabled, TM was presented with an input sample from the training set zero through nine in order until all full sequences had been seen.

5. When each input sample was shown, the previous prediction was assumed to be the prior digit that was seen. Therefore, at each training step, the TM was provided the current input and the prior predicted input, subsequently being tasked with learning the relationship between them. At this phase, the prediction generated by the TM was ignored.

6. Once all of the training samples were shown to the TM, learning was disabled and the unseen test data were provided as input to TM. The previous predictive state was assumed to be the correct prior digit.

7. The predictive state output of TM, represented as a $K \times N$ matrix ($K$ SP columns by $N$ TM cells), was flattened back into a vector of the same size as the original SP input by making the value of each component of the vector the sum of the number of cells in the predictive state. If the sum was greater than zero, the output for the reconstructed column vector would be a one.

8. This new vector, generated for each input sample, was written out to a file in the libsvm-compliant format. All labels were set to be that of the ground truth (expected) label.

9. The output from the TM was tested using svm-predict using the model generated by the same spatial pooler model used to train the temporal memory.

The ability for libsvm to correctly associate the output from TM with the ground truth label was identified as a correct single-step prediction. Considering that there were 10 labels, the performance of random selection would be expected to yield a result of around 10%. However, using TM in conjunction with the 784 column SP and SVM yielded a prediction accuracy
Table A.1: The constant parameters selected for the temporal memory model.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of columns</td>
<td>784</td>
</tr>
<tr>
<td>Max number of segments</td>
<td>50</td>
</tr>
<tr>
<td>Segment activation threshold</td>
<td>1</td>
</tr>
<tr>
<td>Permanence threshold</td>
<td>0.5</td>
</tr>
<tr>
<td>Permanence sigma</td>
<td>0.0005</td>
</tr>
<tr>
<td>Permanence Increment</td>
<td>0.001</td>
</tr>
<tr>
<td>Permanence Decrement</td>
<td>0.002</td>
</tr>
<tr>
<td>Connection probability</td>
<td>0.2</td>
</tr>
</tbody>
</table>

of 28.85% and 27.83% for training and test in a sample case, respectively. Although, it is anticipated that for significantly larger SP representations, TM would perform better. Furthermore, the MNIST dataset contains a considerable amount of variation in the input representation.