CORTICAL COLUMNS COMPUTING SYSTEMS: Microarchitecture Model, Functional Building Blocks, and Design Framework

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Department of Electrical & Computer Engineering

Harideep Nair

B.Tech. Electrical Engineering, Indian Institute of Technology (IIT) Bombay
M.Tech. Electrical Engineering, Indian Institute of Technology (IIT) Bombay

Carnegie Mellon University
Pittsburgh, PA

October, 2024

CORTICAL COLUMNS COMPUTING SYSTEMS:

Microarchitecture Model, Functional Building Blocks, and Design Framework

by

Harideep Nair

has been approved

October, 2024

Doctoral Dissertation Committee:

John Paul Shen, Chair
James E. Smith
Brandon Lucia
Perry Wang

Acknowledgements

My Ph.D. journey at Carnegie Mellon has been one filled with joy, excitement, and immense learning both on an academic and personal level. I am extremely grateful to everyone involved for helping me transform into a better researcher, ready to face the challenges of the real world. First and foremost, I would like to thank my advisor Prof. John Paul Shen for his continuous support and encouragement, and providing me with an opportunity to embark on this exciting adventure with him. It has been amazing how our casual conversations often turn into exciting new ideas. John embodies everything I could ever hope for in an advisor - vision, wisdom, experience (both industry and academia), passion, kindness, approachability, among many more indescribable qualities. John has played a major role in my personal growth. He has been a constant inspiration to me, not only with his visionary research ideas but also regarding managing professional as well as personal relationships. He taught me to always strive for excellence in research, writing, and presentation and at the same time to always have fun - this has become the motto of my life. I couldn't be luckier to have him not only as my advisor but also call him my friend and mentor for life. *John, my cortical columns look forward to continually learning from you through our endless lunch meetings.*

I am grateful to my thesis committee members Prof. Jim Smith, Prof. Brandon Lucia, and Perry Wang for their time and invaluable feedback towards my research. Specifically, I would like to thank Jim for his tremendous support and guidance throughout my Ph.D. journey. It's his foundational research and hot-off-the-press fascinating draft documents that I anchored my research on, often going through the exercise of replicating his Matlab simulations in my PyTorch simulator before delving into microarchitecture design. I have learned so much from him by working closely with him and watching the research wheels turn in his head. Being a world-renowned expert and one of the most well-recognised names in Computer Architecture, Jim's brilliance coupled with his humility will never cease to inspire me. He has taught me what it means to be a true Computer Architect. I hope to continue our discussions on cortical columns, sprinkled with interesting life experiences and travel observations.

I am proud, humbled and eternally grateful for having had the opportunity to work with John and Jim who are both my heroes in Computer Architecture - I grew up reading John's book

and Jim's seminal papers throughout my undergraduate curriculum and research.

My internship at MediaTek, which continued over two years, has been an exhilarating experience for me, filled with immense learning and camaraderie. I am very grateful to Perry Wang and Tsung-Han Lin for embracing me as more than just an intern and making me part of a wonderful team consisting of Nien-En Lee, Gopal Srinivasan, Hank Liu, Santha Bhasuthkar, Akshay Ramanathan among others. They provided me with the opportunity to make impactful contributions across the stack, significantly enhancing my professional growth. I would like to especially thank Perry since he has always been there for me since the beginning of my internship in 2020. His mentorship, technical guidance, and kindness were the major factors that made my industry experience so much smoother than I could ever imagine. Perry has always believed in me and that is something that I will strive hard to keep up throughout my life.

Back at CMU, another mentor I am very grateful for is Prof. Quinn Jacobson. I have thoroughly enjoyed our 2-hour long whiteboard conversations, which I hope to continue. Further, being an industry veteran, Quinn's advice on contemporary technological landscape has significantly enlightened me. Additionally, I am very thankful to Abhinav Jauhri who is like a big brother to me and has been my constant pillar of support. I would also like to thank my fellow Ph.D. peers and collaborators Shanmuga Venkatachalam, Prabhu Vellaisamy, Shreyas Chaudhari, and Tyler Nuanes. They have made my journey at CMU so much more enjoyable. Especially, thanks to Shanmuga and Prabhu for rooting for me throughout as my closest friends - looking forward to many more future collaborations and (adventurous) conference travels. I am also thankful to my graduate student collaborators, faculty members from different departments under whom I took courses, and my academic advisors Greta Ruperto and Nathan Snizaski.

This research was sponsored by MediaTek Fellowship, Qualcomm Innovation Fellowship, and CMU Dean's Fellowship, and I am very grateful for their generous support.

Last but not the least, I would like to thank my dear parents who have always been there by my side. A video call with them after a tough day is all I need to regain a smile on my face. None of this would have been possible without them. This one is for you, Amma and Appa!!!

Despite my attempt here, words really cannot describe how grateful I am for everyone involved in my academic and personal experience over the last six years - I will forever cherish my time at CMU and the lifelong friendships I made along the way, including those at MediaTek.

Abstract

Reverse-engineering the human brain has been a grand challenge for researchers in machine learning (ML), experimental neuroscience, and computer architecture. Current deep neural networks (DNNs), motivated by the same challenge, have achieved remarkable results in ML applications. However, despite their original inspiration from the brain, DNNs have largely drifted away from biological plausibility, resorting to intensive statistical processing on huge amounts of data. This has led to exponentially increasing demand on hardware compute resources that is quickly becoming economically, technologically, and environmentally unsustainable.

Recent neuroscience research has led to a new theory on human intelligence that suggests Cortical Columns (CCs) as the key compute units within the human neocortex, which contains about 150,000 such CCs. The neocortex gains its intelligence through CC's ability to model sensory information in structured Reference Frames (RFs) and continuously update its models as the sensor interacts with the environment. Each CC is computationally powerful and can model complete objects through continuous predict-sense-update loops. This leads to the overarching question: *Can we build Cortical Columns Computing Systems (C3S) that possess brain-like capabilities as well as brain-like efficiency?*

This dissertation presents pioneering research focusing on addressing this question, wherein a framework for designing and implementing *Cortical Columns Computing Systems (C3S)* is presented. This framework consists of three major components: 1) a microarchitecture model for designing and implementing cortical columns and CC-based computing systems using off-the-shelf digital CMOS technology; 2) a suite of specialized functional building blocks to implement application-specific CC-based processing units with significant improvements on Power-Performance-Area (PPA) efficiency; and 3) an automated design framework and toolsuite, *C3SGen*, consisting of a PyTorch-based software simulator for rapid application-specific design space exploration and a hardware design tool that generates post-layout netlists for direct CMOS implementation of special-purpose C3S sensory processing units. Initial findings indicate that designing truly intelligent and extremely energy-efficient C3S-based sensory processing units, using off-the-shelf digital CMOS technology and tools, is quite feasible and very promising, and certainly warrants further research exploration.

Contents

List of Tables

List of Figures

Part I

Prolog

Chapter 1

Introduction and Background

1.1 Scope and Overview of Dissertation

The field of deep learning (DL) [1] has seen extraordinary progress over the last decade driven by algorithmic advancements in deep neural networks (DNNs) and has established itself as the de facto standard technology for sensory processing tasks such as visual object recognition/detection, audio/time-series signal processing, natural language processing, etc. However, this progress has thrived with heavy dependence on increasing hardware resources, and power and energy consumption. These hardware resources such as CPUs, GPUs and specialized accelerators [2] primarily employ Turing computation model and von-Neumann computer architecture [3]. Such computation model and architecture were originally developed for numerical and symbolic processing tasks that were difficult for humans. In contrast, human neocortex (part of the brain that houses intelligence) is highly efficient in sensory processing and pattern recognition and capable of online continual learning. Current deep learning is attempting to replicate human-like sensory processing capability using conventional Turing-von Neumann computing machines. This mismatch between brain and computers is illustrated in Figure 1.1.

It has been observed by Hans Moravec and others that these two computation models and systems are quite different and distinct; this observation is known as the "Moravec's Paradox" [4–6]. This leads to the current overarching grand challenge: **Can we design much more energy-efficient computing hardware systems for human-like sensory processing with continuous learning capability by mimicking the architecture, organization, and operation of the human**

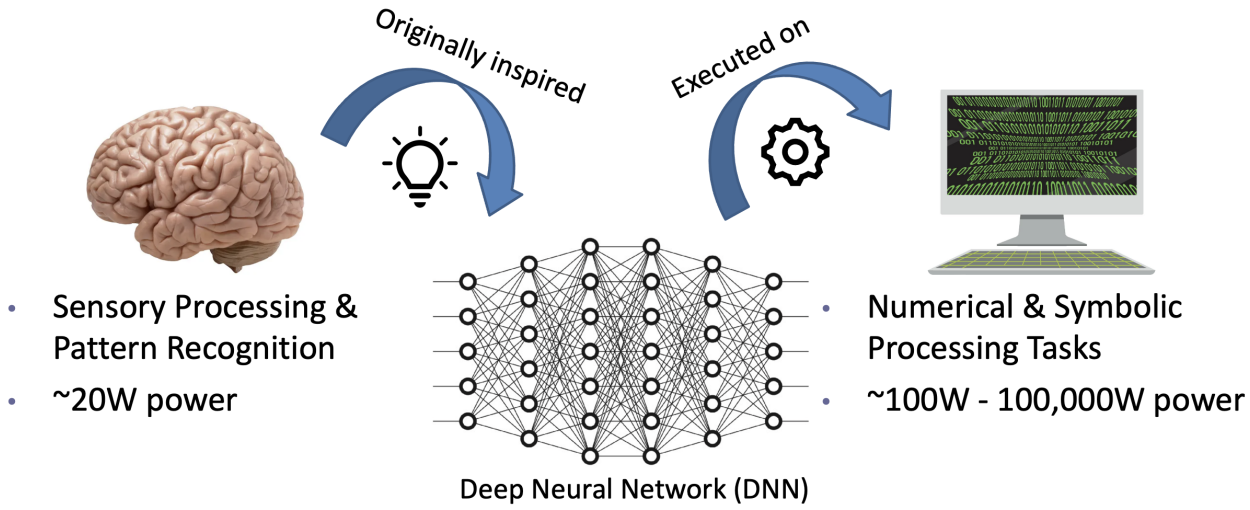


Figure 1.1: DNNs and Brain vs. Computer Mismatch: DNNs, originally inspired from the brain, primarily target sensory processing applications that humans are inherently highly adept at. However, these DNNs are executed on Turing, von Neumann computers that were originally designed for numeric and symbolic tasks considered difficult for humans. This mismatch is manifested in the orders of magnitude difference in power consumption between both compute substrates (20W for brain vs. upto 100kW for computers) and leads us to rethink compute hardware for artificial intelligence (AI).

neocortex? In his FCRC'19 keynote [7], Jim Smith lays out a roadmap for reverse-architecting the neocortex and emphasizes that significant work needs to be done in this regard. This grand challenge is not new and has been around for decades [8]. In 1990, Carver Mead first coined the term "neuromorphic computing" [9] for this grand challenge. But the same notion and aspiration can be traced back to Frank Rosenblatt's "perceptrons" from the late 1950's [10]. However, as a result of extraordinary scaling of silicon technology and significant advances in experimental neuroscience in the last few decades, there has been a resurgence of interest in neuromorphic computing [11, 12]. This dissertation adopts a very specific strategy in pursuing this grand challenge as will become increasingly conspicuous throughout the following chapters.

The research presented here builds on the seminal works by James E. Smith on temporal neural networks (TNNs) [13–15] and is strongly influenced by Jeff Hawkins' recent book "A Thousand Brains: A New Theory of Intelligence" [16]. Unlike convolutional neural networks (CNNs), TNNs are temporal spiking neural networks that embrace a strong adherence to biological plausibility and rooted in a rigorous space-time algebra [14]. TNNs encode and process

information in temporal form (i.e., relative timings of spikes) mimicking the brain's neocortical sensory signal processing along with local online continuous learning. On the other hand, Hawkins' new theory on intelligence [16], informed by extensive neuroscience research, suggests *Cortical Columns* (CCs) as the key compute units within the human neocortex. The neocortex gains its intelligence through CC's ability to model sensory information in structured *Reference Frames* (RFs), and continuously update its models as the sensor interacts with the environment. Higher the number of CCs (150,000 in humans), higher the intelligence. Each CC is computationally powerful and capable of learning models of complete objects. Further, unlike DNNs that separate training and inference, CCs support online continuous learning and can dynamically adapt to sensory input changes. We observe great synergy between CCs and TNNs.

If cortical columns (CCs) are indeed the key to intelligence and intelligent sensory processing, **can** we build such CCs and computing systems based on CCs using standard digital CMOS technology? If so, **how**?

This dissertation presents a framework for designing and implementing *Cortical Columns Computing Systems (C3S)*. This framework consists of three major components: 1) a microarchitecture model for designing and implementing cortical columns and CC-based computing systems using off-the-shelf digital CMOS technology; 2) a suite of specialized functional building blocks to implement application-specific CC-based processing units with significant improvements on Power-Performance-Area (PPA) efficiency; and 3) *C3SGen* toolchain which consists of a PyTorch-based software simulator tool for rapid design space exploration targeting specific applications and a hardware generator tool for direct CMOS implementation of special-purpose C3S sensory processing units, supporting automated generation of post-layout netlists from PyTorch models.

1.2 Motivation

The Artificial Intelligence (AI) boom that started with AlexNet [17] in 2012, propelled with the avalance of DNN architectures including CNNs, transformers [18], and most recently large language models (LLMs) [19], has ushered in a new golden era of computer architecture [20]. The demand for computing hardware to run these AI workloads has been more than ever, with upto more than half a million GPUs deployed in datacenters [21]. This insatiable demand for AI hard-

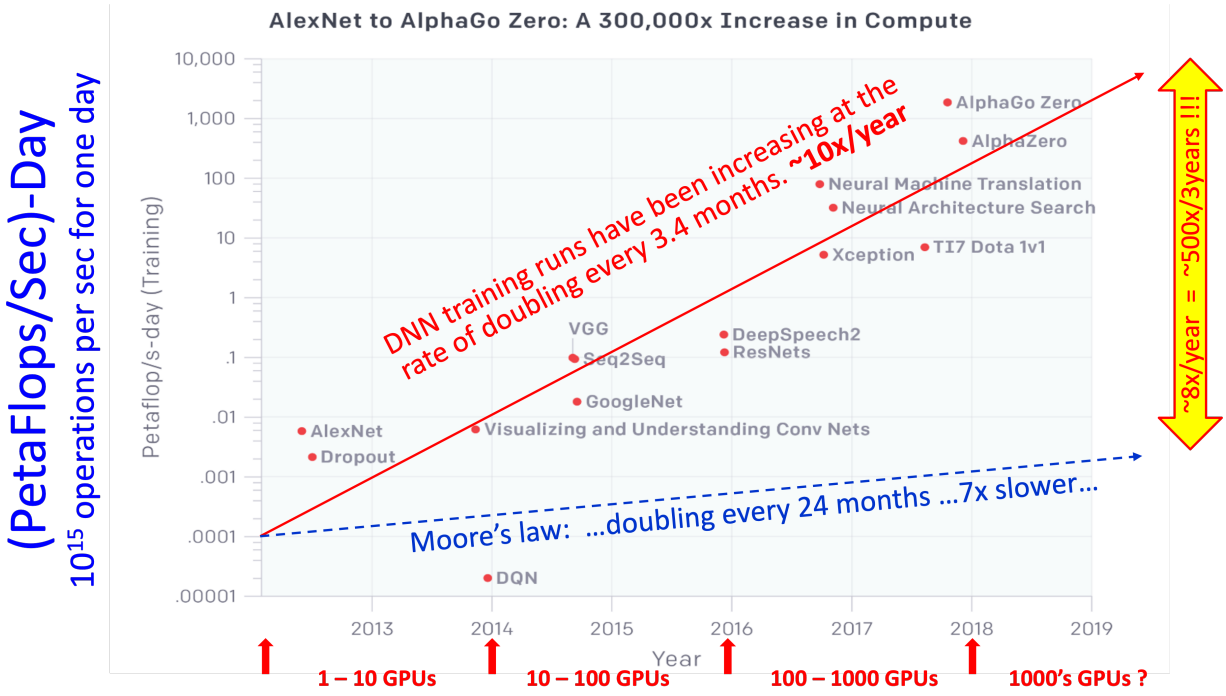


Figure 1.2: Figure adapted from [22] with annotations added. There is a significant trending gap between DNN computation demand and hardware resource supply provided by Moore's Law. This demand-supply gap is increasing at the rate of 8x/year. Thus, to sustain this trend, every year the amount of hardware resources would need to increase by 8x or the training time will need to increase by 8x. A promising candidate solution is to revisit and develop computing paradigms based on the brain's neocortex, which is an existence proof for a highly efficient computing machine for intelligent sensory processing.

ware has paved way for research on novel computer architectures. The specific research strategy adopted in this dissertation is primarily driven by the following two motivating factors.

1.2.1 Unsustainable Trend for Deep Learning Compute

Deep neural networks (DNNs) [1, 19, 23–25] have advanced state-of-the-art in a plethora of applications, particularly those mimicking human sensory processing tasks such as image recognition, object detection, time-series signal (e.g., speech) processing, natural language processing, etc. However, due to exponentially growing models performing high-dimensional tensor processing and global gradient backpropagation, their compute requirements are scaling unsustainably. Specifically, a study by OpenAI in 2018 [22] illustrated that DNN training compute is doubling every 3.4 months. Comparing this with hardware driven by Moore's Law doubling every 24

months, the gap between compute demands and hardware supply is increasing at an exponential rate of 8x per year or 500x every 3 years (see Figure 1.2). There is also growing evidence that the current trajectory of deep learning compute scaling is economically, technologically and environmentally unsustainable [26–28]. On a more optimistic note, some other recent studies have claimed that the rate of DNN model explosion has slowed down between 2018 and 2020 [29, 30], and that the trend is expected to plateau and shrink in the future [31].

There are significant ongoing efforts to deal with this DNN compute complexity explosion. Reducing data value precision through quantization and DNN model sizes through pruning can help mitigate the computation complexity [32–37]. More efficient ways of using the computation hardware infrastructure, including static or dynamic exploitation of value sparsity to avoid unnecessary computation, are also being developed [38–44]. These are all valuable efforts to mitigate the complexity explosion and to help sustain the continuation of the current productive trends. However, there is also the need to concurrently explore other potentially promising alternative paradigms and approaches.

1.2.2 Right Time to Revisit Neuromorphic Computing

All current commercial accelerators for AI compute employ the same computing paradigm that emerged more than 70 years ago based on the Turing computation model and the von Neumann stored-program computer architecture [3]. However, these systems were not originally developed for targeting human-type sensory processing workloads that constitute majority of modern AI compute. This underscores the potential need to explore a much more complexity- and energy-efficient computing model and architecture for AI computing. One approach is to revisit biology and examine how to mimic not just the functional behaviors, but also the structural organization of biological neural networks. Such an approach can potentially enable real intelligent computing with significantly less computation complexity and much better energy efficiency. One of the last major efforts taking such "neuromorphic computing" approach was by Carver Mead and his PhD students at CalTech back in the late 1980's utilizing analog VLSI circuits to model neural computation [9, 45, 46]. Since then, both experimental neuroscience research and digital silicon fabrication technology have made tremendous advancements in the past 40 years, making this

	Conventional Computers	Cortical Columns Computing Systems (C3S)	Neocortex Biological Systems
LEVEL 1 APPLICATION	Computing Systems	NEUROMORPHIC COMPUING: **SILICON NEOCORTEX** (Applications & System Architecture)	Cerebral Neocortex
LEVEL 2 KERNEL	CPU, GPU, TPU, NPU, MEM, I/O	DIVERSE SENSORY PROCESSING UNITS: **(MACRO) CORTICAL COLUMNS, RF, VOTE** (Processor Microarchitecture)	Neocortical Lobes/Regions
LEVEL 3 COLUMN	ALU, FPU, LSU, ARF, ROB, LSQ	FUNCTIONAL UNITS BUILDING BLOCKS: **MINI-COLUMNS: INFERENCE, GRID, PLACE** (RTL-Level Implementation)	Neocortical Columns
LEVEL 4 NEURON	Adders, Decoders, Multiplexers, etc.	HARDWARE DESIGN BUILDING BLOCKS: **NEURON, DENDRITE, SYNAPSE, STDP, WTA** (Gate-Level Implementation)	Dendrites, Neurons Excitatory/Inhibitory

Figure 1.3: Comparison of abstraction levels across three computing domains: conventional computers, proposed C3S neuromorphic computing systems (highlighted in yellow) and biological computing systems. The focus of this dissertation is marked by the red box.

an ideal time to revisit the neuromorphic computing approach [11, 12].

In recent years, several neuromorphic chips have been introduced both from academia and industry, including analog [47], digital [48–55] and mixed-signal [56] implementations. Additionally, analog compute-in-memory approaches based on emerging devices such as memristors [57] are actively being researched. While analog approaches possess desirable qualities such as ultra low power, they are susceptible to noise and robustness issues, and are relatively difficult to design and fabricate. Hence, we focus on digital CMOS implementation that leverages the highly standardized fabrications process flows to deliver mass market impact in near future. Furthermore, a dominant trend across existing neuromorphic approaches is to implement a large number of spiking neurons communicating with each other via event-driven packets that encapsulate spiking information. Such spike-based event-driven computations have been shown to be more energy-efficient than traditional compute. However, they still lack the structural hierarchy and functional abstraction in the form of cortical columns as seen in the neocortex.

This dissertation adopts a "neuromorphic computer architecture" approach by adhering to biologically plausible abstractions and hierarchy of composable building blocks in order to design cortical column-based processing units (see Figure 1.3). Interestingly, similarities can be observed

across conventional computer architectures and biological systems in terms of the different levels of computing abstraction. Based on this, computing abstractions for C3S can be developed as highlighted in yellow in Figure 1.3. For example. "neurons", "minicolumns", "cortical columns" (or "macrocolumns") can be viewed as analogous to adders, floating point units, CPUs respectively. These C3S abstractions will be discussed in detail in the following chapters. The focus of this dissertation (marked by the red box in Figure 1.3) is to develop microarchitecture model and RTL/gate-level functional building blocks for cortical columns.

1.3 Key Foundational Works

This dissertation bases its research on two large bodies of recent prior works in order to explore the potential of creating brain-inspired computing fabric, Cortical Columns Computing Systems (C3S), that can achieve both the sensory processing capabilities and the energy efficiency of the brain. Specifically, it builds on the foundational works of Jeff Hawkins and James E. Smith on reverse-engineering the neocortex from a neuroscience theorist's perspective and a computer architect's perspective respectively.

1.3.1 Hawkins' Thousand Brains Theory: Cortical Columns with Reference Frames

In 2021, Jeff Hawkins published a fascinating book entitled *"A Thousand Brains: A New Theory of Intelligence"* [16]. In this book, accompanied by additional publications from his group at Numenta [58, 60–62], Hawkins proposes a new theory of neocortical computation and intelligence backed by extensive neuroscience literature. Hawkins suggests that cortical columns (CCs) are the fundamental neocortical processing units that embody intelligence. CCs learn information through movement (sensorimotor learning), i.e., as the associated sensing agent moves through an environment. The environment could be physical (e.g., a room, an object such as mug, car, etc.) or metaphysical (e.g., abstract concepts like mathematics, democracy, etc.). CCs model this sensory information and knowledge in structured *Reference Frames* (RFs), and learn models/maps through continuous predict-sense-update feedback loops. RFs are directly inspired from the biological grid cells in the entorhinal cortex [63] and place cells in the hippocampus [64], that support spatial navigation for humans in the real world. Further, Hawkins postulates that such

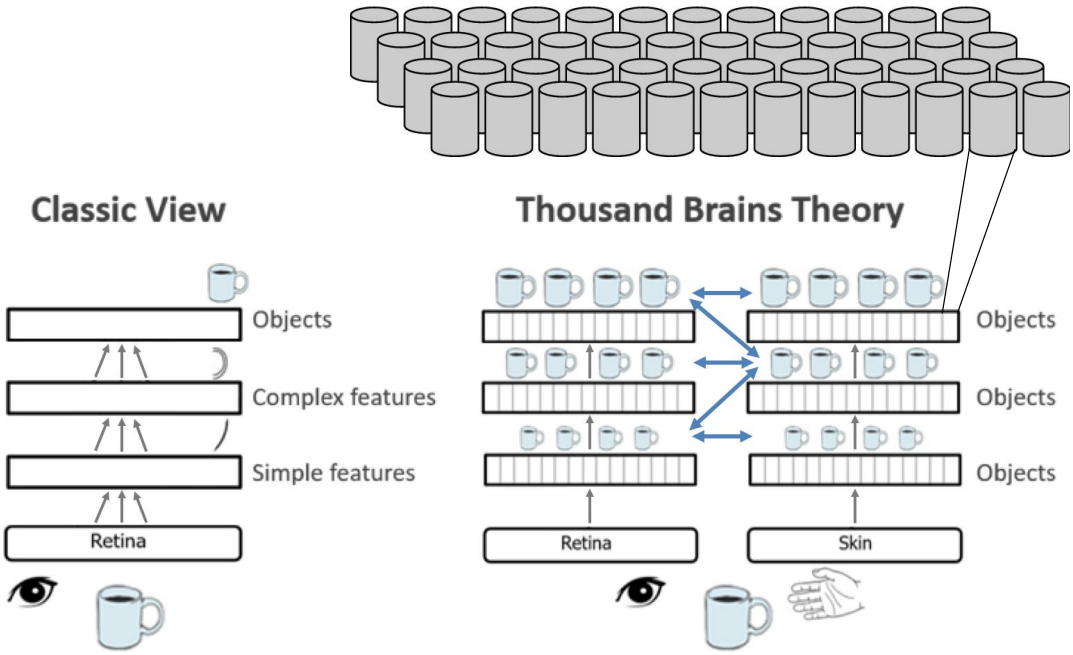


Figure 1.4: Figure taken from [58] and [59]. **Left:** Traditional hierarchical view where complexity of features recognized increases up the hierarchy with complete objects detected only at the topmost region. **Right:** Hawkins' view where neocortex contains about 150K cortical columns and multiple cortical columns across different hierarchies and modalities can all learn complete models of objects (e.g., coffee cup) and can communicate to reach a consensus on the output.

a reference frame implements the common universal algorithm for achieving intelligence within each CC, irrespective of the sensory modality feeding that particular CC.

Each cortical column is computationally powerful and can model complete objects in its RF, regardless of its modality or hierarchical abstraction. This is illustrated in the right half of Figure 1.4 where complete models of an object exist in multiple CCs across different sensory modalities (e.g., vision, touch) and across different hierarchy layers at different scales. Multiple CCs, across sensory modalities and layer hierarchies, can communicate and reach consensus via a voting process (marked by blue bi-directional arrows). This contrasts the traditional strict hierarchical view in deep learning (left half of Figure 1.4) where the lower layers only learn primitive features (e.g., lines, curves) and the higher layers learn bigger and more complete features (e.g., wheel, car). Hawkins suggests the increasing level of intelligence demonstrated by higher-functioning mammals is mainly due to the increase in the total number of CCs and not necessarily due to increased complexity or sophistication of the CCs. Human neocortex consists of about 150,000

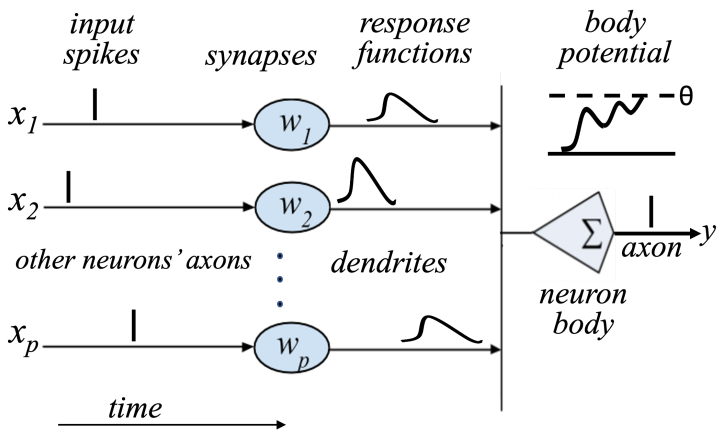


Figure 1.5: SRM0 Neuron Model: Each neuron has p inputs connected through p synaptic weights and one output. At every input, the arrival of an input spike elicits a response function whose amplitude is directly proportional to the synaptic weight value. Response functions, if any, from all inputs get accumulated into a body potential at the neuron body, which fires an output spike when the accumulated potential crosses a threshold θ.

CCs. In contrast to current DNNs that separate training and inference, CCs that store and process information in RFs can support online, concurrent, and continuous learning and can dynamically adapt to sensory input changes.

Authors in [60, 62, 65] demonstrate the application of this theory through software simulations by performing environment classification and visual object recognition on MNIST handwritten digits using sequences of sensory inputs correlating with corresponding sensor locations. However, currently no prior work has investigated the feasibility and the potential of direct implementation of reference frames and cortical columns in conventional off-the-shelf digital CMOS technology. Furthermore, recent discussions in DL community seem to align with the basic ideas of Hawkins' theory. They suggest continually learning structured models of the world via interactions with environment can help overcome the issues of brittleness and catastrophic forgetting that plague the current DNNs [66].

1.3.2 Smith's Temporal Neural Networks and Space Time Algebra

Another foundational body of work is by Jim Smith on reverse-architecting the brain with the goal of replicating it using digital CMOS silicon [13–15, 67–69]. At the foundational level, biological computing systems are composed of spiking neurons that process information in the form of

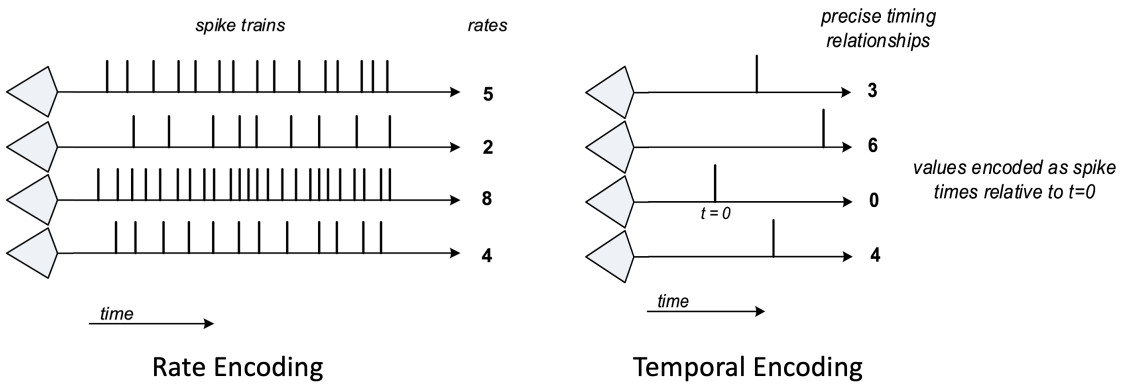


Figure 1.6: Rate Encoding vs. Temporal Encoding: Temporal encoding utilizes a single spike's timing to represent a value whereas rate encoding utilizes a train of spikes to encode a value in the form of number of spikes within a time window. In contrast to rate encoding, temporal encoding has wide experimental support [71–73] and has desirable properties such as higher sparsity and shorter coding window leading to higher energy efficiency and throughput.

a set of incoming (voltage) spikes to generate an output (voltage) spike. Spikes are essentially all-or-nothing events, characterized by sudden peaks in neuronal membrane voltage levels. A simple and widely used model for spiking neuron behavior, SRM0, is illustrated in Figure 1.5 which is based on the Spike Response Model (SRM) proposed in [70]. Smith proposed a new category of spiking neural networks called Temporal Neural Networks (TNNs) [13] that are architected to mimic the organization and attributes of the biological cortical columns in the neocortex. Unlike DNNs performing real-valued computations and global backpropagation, TNNs employ spiking neurons that encode and process inputs as timings of events or spikes (i.e., temporal encoding), and learn using local biologically plausible algorithm called spike timing dependent plasticity (STDP). TNNs differ from most other spiking neural networks (SNNs) that encode values in the rate of spikes as opposed to spike timings. TNNs and subsequent work presented here adopts temporal encoding due to its wide experimental support [71–73] and potential for high energy efficiency and throughput (see Figure 1.6). Further, the STDP learning mechanism [74, 75] is an unsupervised algorithm that utilizes information local to a synapse to determine its weight update. In its simplest form, if the input spike at a synapse arrives before the neuron fires an output spike, it implies a causal correlation between input and output spikes leading to an increase in the synaptic weight, and vice versa. More details will follow in subsequent chapters.

The taxonomy that distinguishes TNNs against other artificial neural networks including

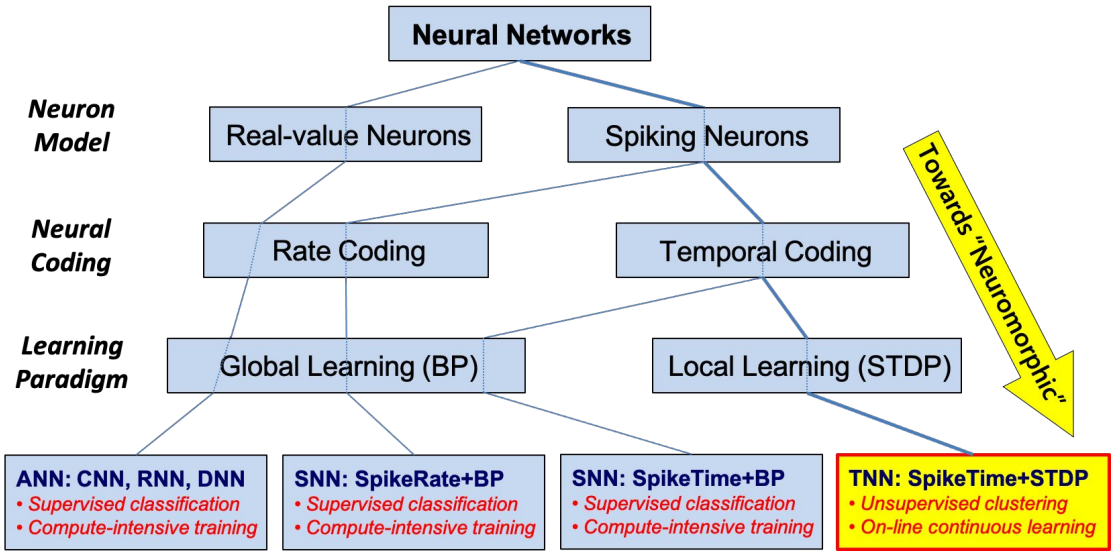


Figure 1.7: Neural network taxonomy contrasting neocortex-inspired temporal neural networks (TNNs) with other artificial neural networks (ANNs). In contrast to other ANNs including deep neural networks (DNNs), TNNs incorporate attributes with strong adherence to biological plausibility, including spiking neuron model, temporal coding of inputs, and local simple spike timing dependent plasticity (STDP) learning rules.

DNNs and SNNs is depicted in Figure 1.7. Hence TNNs, unlike most other ANNs, are closely "neuromorphic" in having strong adherence to biological plausibility. Smith has also developed a TNN-based architecture for cortical columns and demonstrated the capability for doing unsupervised learning with rapid convergence, and the capability to do online, concurrent, and continuous learning [15, 68].

Further, Smith has proposed a new algebra called Space-Time (Temporal) Algebra [14, 67] as the underlying mathematical basis for implementing TNNs. Based on this new Temporal Algebra (instead of Boolean Algebra), temporal functions can be implemented very efficiently in hardware by re-purposing the current digital logic gates to use time as a resource [76]. As an example, if values are encoded as timings of signal transition edges from logic '1' to logic '0', "min" and "max" operations across two inputs can each be performed with a single AND and OR gate respectively. This approach of encoding inputs as signal transition edges has been proposed earlier as Race Logic [77]. Smith has extend this approach to Generalized Race Logic (GRL) in [14] wherein he defines four space-time primitives that can form a complete set for implementing TNNs. As shown in Figure 1.8, these primitives are "min" (AND gate), "max" (OR

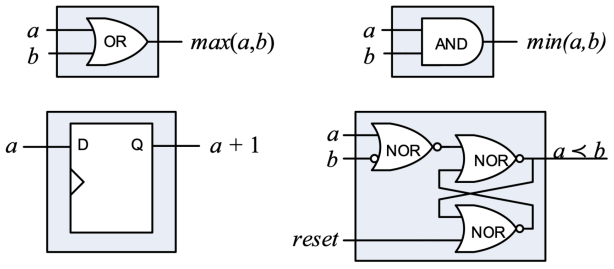


Figure 1.8: Figure taken from [14]: The four Generalized Race Logic (GRL) primitives are shown, with input values encoded as timings of 1→0 transitions of logic signals. In contrast to traditional bit-parallel binary logic, "min", "max", "increment", and "less-than" operations can be implemented using significantly simpler logic with just one or upto a few logic gates.

gate), "inc" or increment (latch), and "lt" or less-than (three NOR gates). Each of these primitives ingests inputs encoded as time (1→0 voltage transition timings of signals). This implies TNNs can be implemented using off-the-shelf digital CMOS. The research presented in this dissertation defines a different form of "Temporal Logic" implementation that is significantly more efficient than GRL for implementing TNNs. This will be discussed in detail in Chapter 2.

1.4 Major Contributions

There is remarkable synergy between Smith's TNNs and Hawkins' theory with cortical columns and reference frames. This dissertation serves as the first work that leverages the convergence between these two foundational bodies of works to design and implement Cortical Columns Computing Systems (C3S) [78] with strong adherence to biological plausibility and biological structural hierarchy. Five major contributions are made here, addressing the following five major research questions:

1. Can we implement C3S using standard off-the-shelf digital CMOS? [Chapters 2, 3, 4]
2. Can we further optimize the C3S designs by building custom macro cells, still using the digital CMOS framework? [Chapter 5]
3. Can synaptic array compute, which incurs bulk of the hardware complexity, be efficiently implemented entirely in memory (SRAM) instead of using flip-flops? [Chapter 6]

4. Can a reference frame within a cortical column be implemented using content-addressable memory (CAM)? [Chapter 7]

5. Can C3S design space be explored and automated using a design framework spanning PyTorch all the way down to chip layout? [Chapters 8, 9]

1.5 Dissertation Structure

This dissertation is mainly organized into 3 parts: Part II on microarchitecture, Part III on functional building blocks, and Part IV on design framework and tools. They are bookended by two parts, one on introduction (Part I) and the other on conclusions (Part V). These parts amount to 10 chapters. Additionally, there is a separate part on Appendix containing a chapter on miscellaneous related work done by the author. This organization into chapters is detailed further below.

Chapter 2 introduces *Temporal Logic Design* (TLD) as an implementation methodology used for repurposing standard digital CMOS circuits to implement temporal functions that process values encoded in time. Rigorous analysis across different TLD implementation styles is performed to determine the most optimal approach for C3S implementation.

Using TLD, Chapter 3 proposes a microarchitecture framework for implementing feedforward Temporal Neural Networks. Here, a key building block for TNNs, *minicolumn*, is introduced which comprises of multiple neurons sharing a set of inputs through a synaptic crossbar followed by lateral inhibition. This chapter uses a "point neuron" model and acts as the key foundation for Chapter 4. See [79, 80].

Chapter 4 introduces a more nuanced neuron model incorporating "active dendrites" which closely resemble the neuronal hierarchy in biology. This chapter shows how cortical columns with reference frames can be implemented using TNNs consisting of these active dendrite neurons and minicolumns. See [78].

Chapter 5 demonstrates how the efficiency of aforementioned microarchitecture can be enhanced by identifying the key functional building blocks of a minicolumn and building highly optimized custom macros for the same. These building blocks still utilize flip-flops or registers. See [81].

Chapter 6 proposes a novel type of compute-in-memory (CIM) functional building block that models the entire synaptic array within a minicolumn in SRAM. Synaptic array, being the largest contributor to hardware complexity of C3S, when implemented in SRAM, has the potential to achieve significant improvements in power, performance, and area (PPA). See [82].

Chapter 7 observes resemblance between the functionality of a reference frame and content addressable memory (CAM), and proposes a CAM-based building block specifically for implementing reference frames within cortical columns. See [83].

Building primarily on the works in Chapters 3 and 5, Chapter 8 proposes an automated design tool, *TNNGen*, that generates post-layout netlists of single-layer TNNs from PyTorch models. The designs presented here follow the simpler "point neuron" model. See [84].

Chapter 9 builds on TNNGen to propose a more comprehensive design framework, *C3SGen*, that expands the capabilities to implement multi-layer TNNs, and cortical columns and reference frames incorporating "active dendrite" neurons.

Finally, Chapter 10 summarizes the conclusions and discusses avenues for future work. Additionally, Chapter 11 in Appendix discusses two works that apply Temporal Logic Design to general matrix multiplication for conventional deep neural networks (see [85–87]).

1.6 Compendium of Research Publications by the Author

A list of research publications by the author is presented below. The main publications that directly contribute to this dissertation are starred.

★ Prabhu Vellaisamy[1], **Harideep Nair**[1], Vamsikrishna Ratnakaram, Dhruv Gupta, and John Paul Shen. "*TNNGen: Automated Design of Neuromorphic Sensory Processing Units for Time-Series Clustering*", TCAS-II, 2024 [Published by Invitation after Acceptance in ISCAS]. [84]

★ **Harideep Nair**, William Leyman, Agastya Sampath, Quinn Jacobson, and John Paul Shen. "*NeRTCAM: CAM-Based CMOS Implementation of Reference Frames for Neuromorphic Processors*", NICE, 2024. [83]

[1]Both co-first authors contributed equally to this work.

★ **Harideep Nair**, David Barajas-Jasso, Quinn Jacobson, and John Paul Shen. "*TNN-CIM: An In-SRAM CMOS Implementation of TNN-Based Synaptic Arrays with STDP Learning*", AICAS, 2024. [82]

★ John Paul Shen and **Harideep Nair**. "*Cortical Columns Computing Systems: Microarchitecture Model, Functional Building Blocks, and Design Tools*". Neuromorphic Computing, Ch. 8, IntechOpen, 2023. [Book Chapter] [78]

★ **Harideep Nair**, Prabhu Vellaisamy, Santha Bhasuthkar, and John Paul Shen. "*TNN7: A Custom Macro Suite for Implementing Highly Optimized Designs of Neuromorphic TNNs*", ISVLSI, 2022. [81]

★ Shreyas Chaudhari, **Harideep Nair**, José M.F. Moura, and John Paul Shen. "*Unsupervised Clustering of Time Series Signals using Neuromorphic Energy-Efficient Temporal Neural Networks*", ICASSP, 2021. [80]

★ **Harideep Nair**, John Paul Shen, and James E. Smith. "*A Microarchitecture Implementation Framework for Online Learning with Temporal Neural Networks*", ISVLSI, 2021. [79]

- **Harideep Nair**[1], Prabhu Vellaisamy[1], Tsung-Han Lin, Perry Wang, Shawn Blanton, and John Paul Shen. "*Commercial Evaluation of Zero-Skipping MAC Design for Bit Sparsity Exploitation in DL Inference*", VLSI-SoC, 2024.

- Shanmuga Venkatachalam, **Harideep Nair**, Prabhu Vellaisamy, Yongqi Zhou, Ziad Youssfi, and John Paul Shen. "*Realtime Person Identification via Gait Analysis using IMU Sensors on Edge Devices*", ICONS, 2024. [88]

- Prabhu Vellaisamy, **Harideep Nair**, Di Wu, Shawn Blanton, and John Paul Shen. "*Exploration of Unary Arithmetic-Based Matrix Multiply Units for Low Precision DL Accelerators*", ISVLSI, 2024. [87]

- Prabhu Vellaisamy, **Harideep Nair**, Joseph Finn, Manav Trivedi, Albert Chen, Anna Li, Tsung-Han Lin, Perry Wang, Shawn Blanton, and John Paul Shen. "*tubGEMM: Energy-Efficient and Sparsity-Effective Temporal-Unary-Binary Based Matrix Multiply Unit*", ISVLSI, 2023. [86]

- **Harideep Nair**, Prabhu Vellaisamy, Albert Chen, Joseph Finn, Anna Li, Manav Trivedi, and John Paul Shen. *"tuGEMM: Area-Power-Efficient Temporal Unary GEMM Architecture for Low-Precision Edge AI"*, ISCAS, 2023. [85]

- Shanmuga Venkatachalam, **Harideep Nair**, Ming Zeng, Cathy Tan, Ole Mengshoel, and John Paul Shen. "*SemNet: Learning Semantic Attributes for Human Activity Recognition with Deep Belief Networks*", Frontiers in Big Data, 2022. [89]

- **Harideep Nair**, Cathy Tan, Ming Zeng, Ole J. Mengshoel, and John Paul Shen. "*AttriNet: Learning Mid-Level Features for Human Activity Recognition with Deep Belief Networks*", UbiComp-ISWC, 2019. [90]

Part II

Microarchitecture Model

Chapter 2

Neuromorphic Temporal Logic Design

Time is a resource that is eternal and freely available. So why not use it as a medium for encoding and processing information? This is precisely what biological computing systems are widely postulated to implement, in order to enhance their energy efficiency. In late 1980s, Thorpe and Imbert [71, 72] made a persuasive experimentally-supported argument for inter-neuron communication via precisely timed spikes, followed by further experimental evidence in [73]. In this dissertation, we define *Temporal Logic Design* (TLD) as an implementation methodology used to *directly* map temporal (space-time) functions to digital CMOS circuits. In a direct CMOS implementation, the actual hardware clock cycle is used as the basic time unit for temporal processing, i.e., time itself is not stored as a binary value but implicit in the hardware clock. This approach differs from existing digital neuromorphic implementations such as IBM's TrueNorth [52], Intel's Loihi [53] which use packetized messages storing binary values to represent spikes and spike times. This chapter uses neuron implementation as a vehicle to demonstrate Temporal Logic Design. Additional work by the author on utilizing TLD for general matrix multiplication for conventional DL workloads is discussed in Appendix (Chapter 11).

In Temporal Logic Design, the primary decision to make is on how to represent arrival of spikes (or events or encoded values in general) at a particular time in hardware. This can be done in two ways: 1) *edge temporal* (ET) that utilizes timings of edge transitions of logic signals ($1 \rightarrow 0$ or $0 \rightarrow 1$), and 2) *pulse temporal* (PT) that uses timings of logic pulses. With edge temporal approach, TLD devolves to GRL introduced in Chapter 1. However, as will be discussed in this

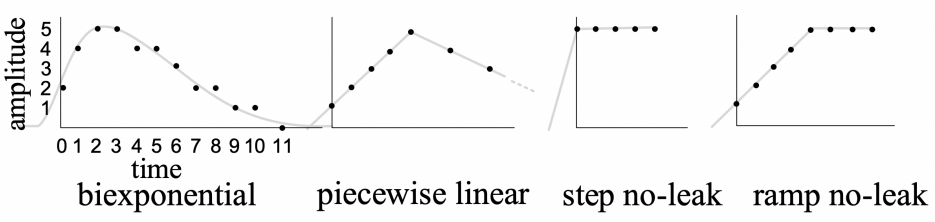


Figure 2.1: Four Well-known Discretized Response Functions

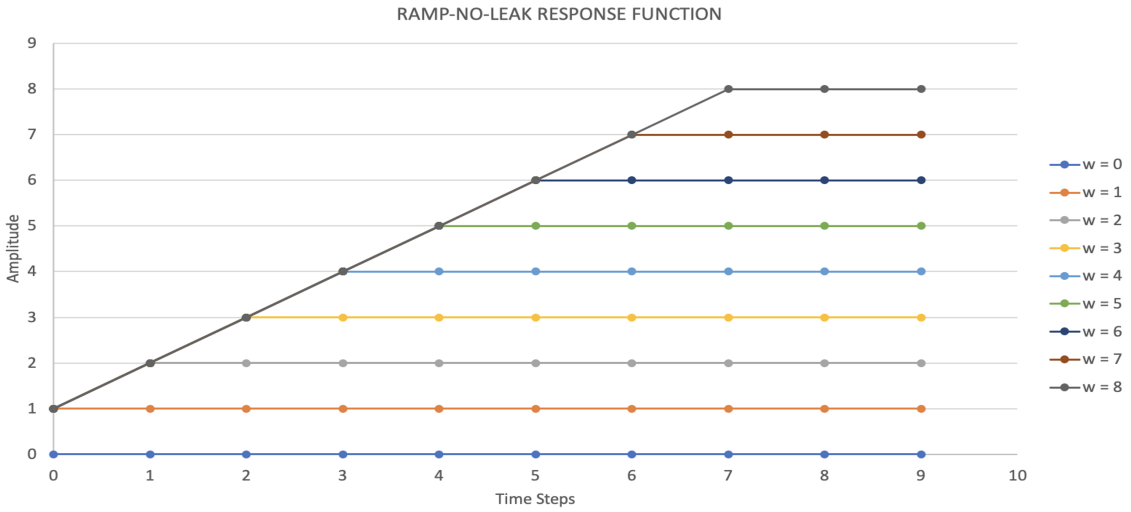


Figure 2.2: Discretized ramp-no-leak (RNL) response function saturating at a maximum amplitude of 8.

chapter, pulse temporal approach leads to more efficient TNN implementations. Here, we use a single neuron as a vehicle for investigating the tradeoffs between the two approaches, extending the pulse temporal approach to an additional optimized variant. This optimized variant serves as the baseline for feedforward TNN microarchitecture presented in Chapter 3, which uses a clever optimization to eliminate the need for separate weight storage and further improve the hardware efficiency.

This chapter presents rigorous qualitative analysis, including parameterized gate count equations, of three TLD neuron implementations followed by post-synthesis results that corroborate the qualitative equations. First, the specific response function used for the neuron model is discussed in detail.

2.1 Ramp-No-Leak Response Function

Although a wide variety of response functions may be used as shown in Figure 2.1, the response function of interest here is the ramp-no-leak (RNL) function due to its temporal computational benefits and implementation efficiency. The RNL function (Figure 2.2) increases by a unit step at every time unit until it reaches its peak and then remains constant until it is reset prior to the next computation cycle. The "ramp" allows responses from different synapses to be distributed temporally based on the synaptic strengths (weights), which proves to be particularly powerful for TNNs that operate temporally. Note that this model doesn't "leak". This is based on arguments that the leak is actually just a reset mechanism [91, 92], and hence is much easily done in hardware via a synchronous reset instead of modelling complex leaking patterns. This is one example of a key principle we adopt in our research strategy - attempt to replicate the functional and structural aspects of the brain that contribute to computational power, while discarding features that are purely artefacts of constraints imposed by the organic biological substrate.

2.2 Edge Temporal (ET) Logic

Figure 2.3 shows the block diagram for a ramp-no-leak neuron implemented using edge temporal logic (equivalently, GRL) assuming a maximum synaptic weight value of 8. This design was proposed in [14] and mainly serves as a baseline comparison for the following designs.

Both input spikes (x_1,...,x_N) and output spike (z) are represented using $0\rightarrow1$ signal transition edges. Each input synapse fans out to multiple lines, where each line represents a unit up-step in response function amplitude. Hence, the fan-out for each synapse is equal to the maximum response function amplitude. The time of signal transition at any of these lines essentially denotes the time at which the corresponding up-step in amplitude occurs. The up-steps are sorted temporally by a bitonic sorter which is implemented using GRL *min* and *max* operators (each operator is just a single gate). Since the sorter sorts all the up-steps in ascending order of their timings, a signal edge at the θ^{th} output line indicates the output spike time when the threshold θ is crossed. The design mainly consists of two components, a synaptic array and a bitonic sorter, which will be discussed next.

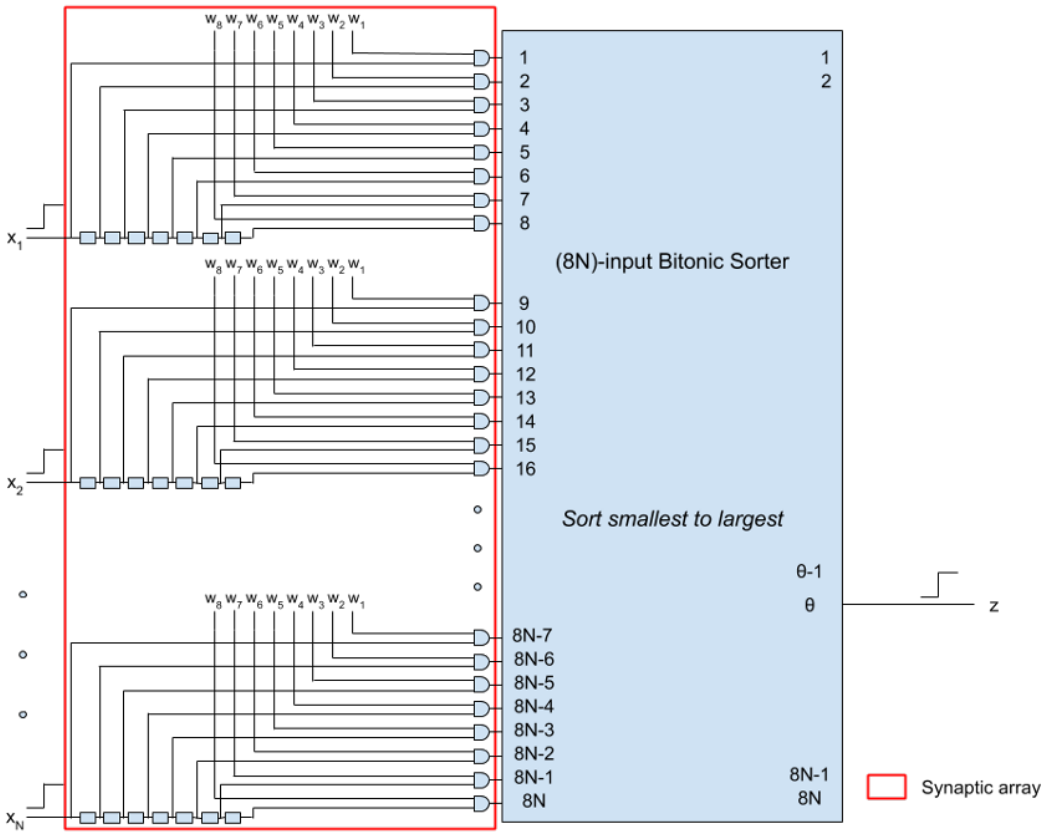


Figure 2.3: An edge temporal RNL neuron with N synapses and weights ranging from 0 to 8. Note that 0→1 transitions are used here.

2.2.1 Synaptic Array

The synaptic array generates delayed signals on the 8 fanned out lines for each input synapse which represent the time values at which up-steps occur. Weights are provided as 8-bit thermometer-encoded input with the number of trailing ones denoting the value. For example, '00000001' represents a weight of 1, '00011111' represents a weight of 5, and so on. For RNL response function, there is an up-step of 1 at every time instant starting from $t = 0$ (time at which input spike arrives at a synapse) and there are as many up-steps as the weight value. When an input spike arrives, 7 delayed versions of it are generated using 7 delay elements with each delay element inserting a unit delay. Simple D flip-flops are used as delay elements. The delayed spikes and the original input spike for every synapse are fed to the bitonic sorter as configured by the corresponding weight using a set of *max* elements. Note that due to thermometer-encoding of weights and 0→1 encoding of spikes, each *max* element is a simple 2-input 'AND' gate with the

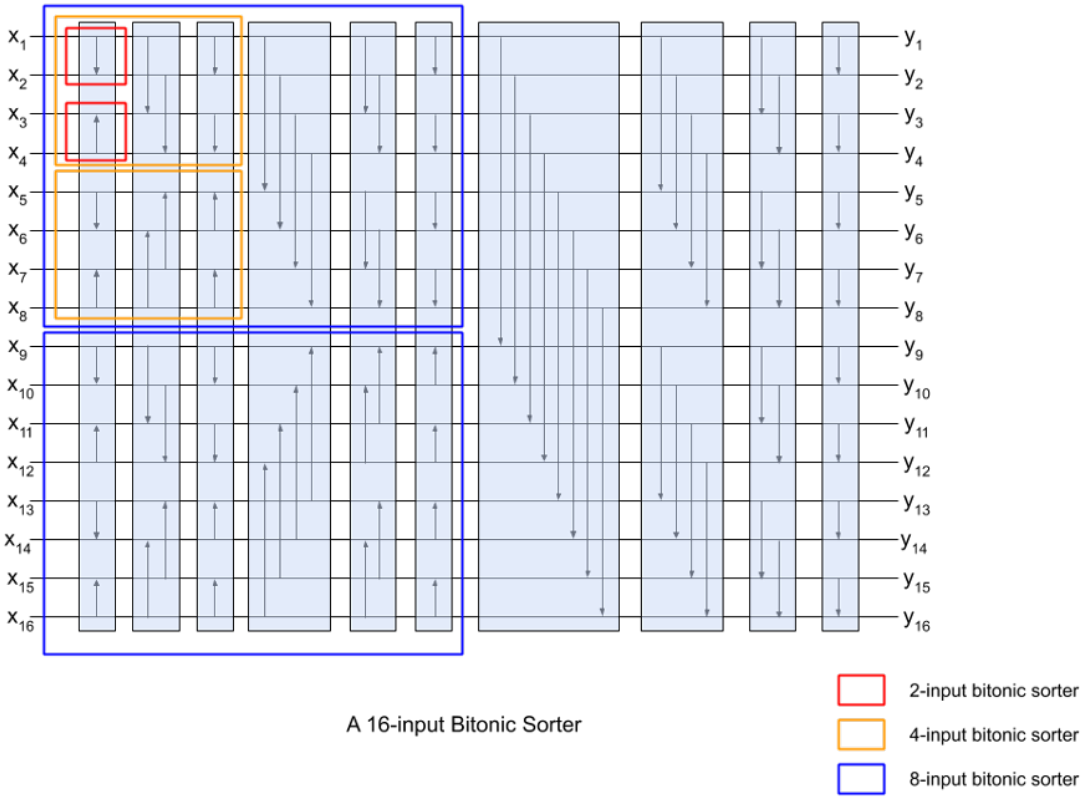


Figure 2.4: An example bitonic sorter with 16 inputs. With ET or GRL, every arrow represents a pair of *min* and *max* operators.

appropriately delayed spike and corresponding weight bit as inputs.

2.2.2 Bitonic Sorter

Figure 2.4 shows the logic diagram for a bitonic sorter with 16 inputs. Each arrow represents a 2-input min-max module (down-facing arrow sorts in ascending order and vice versa). In ETL based on 0→1 transitions, *min* and *max* modules are just simple 'OR' and 'AND' gates. A 2-input min/max module is essentially a 2-input bitonic sorter. Any N-input bitonic sorter can be built recursively using two N/2-sized sorters followed by $log_2 N$ stages at the end. For example, the 16-input sorter in Figure 2.4 consists of two 8-input sorters (upper one sorts in ascending order and bottom one in descending order), followed by 4 stages. In total, a 16-input bitonic sorter has 10 stages (blue boxes in Figure 2.4) with each stage having 8 *min-max* modules.

One key optimization that can be made for ET RNL neuron is that the first 6 stages of the

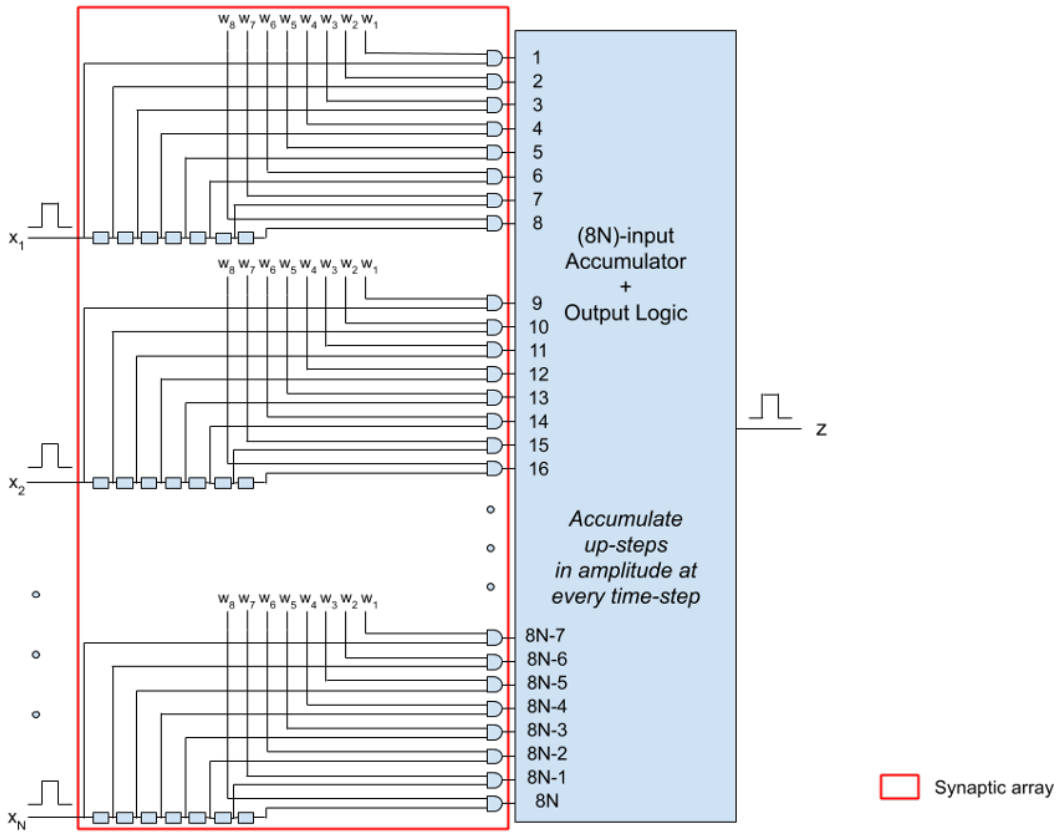


Figure 2.5: A pulse temporal RNL neuron with N synapses and weights ranging from 0 to 8. Note that single-cycle pulses are used here.

sorter which essentially (temporally) sort the input in sets of 8 can be eliminated. This is because the 8 up-steps for each synapse are already sorted among themselves.

Observation: The bitonic sorter is very inefficient and incurs O($Nlog_2N^2$) complexity. If we use pulses instead of edges for representing spikes, the bitonic sorter can be replaced by an accumulator which incurs a significantly lesser O(N) complexity.

2.3 Pulse Temporal (PT) Logic

Figure 2.5 shows the block diagram for a ramp-no-leak neuron implemented using pulse temporal logic assuming a maximum synaptic weight value of 8. Both input spikes (x_1,...,x_N) and output spike (z) are represented using signal pulses with a width of 1 time unit (i.e., single cycle). The fan-out of input synapse into multiple lines is same as in ET, with the only difference being pulses travel through it rather than edges. The time of arrival of pulse at any of the fanned-

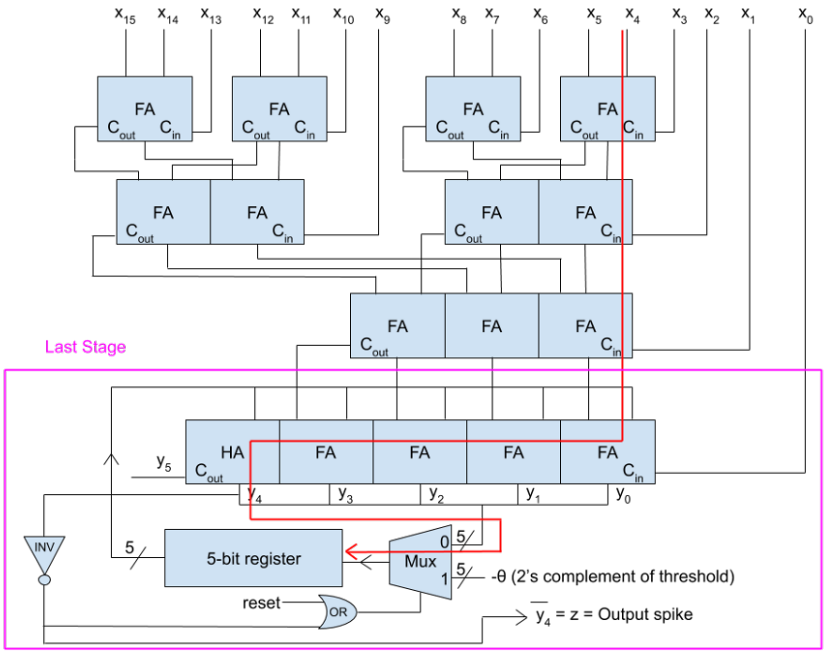


Figure 2.6: A 16-input accumulator consisting of half adders (HA) and full adders (FA) that stores output into a 5-bit register and compares the stored value with a threshold to generate an output spike.

out lines essentially denotes the time at which the corresponding up-step in amplitude occurs. Since these 1-bit pulses can be counted and accumulated as they arrive in time to calculate the aggregate potential of the neuron, a simple accumulator and comparator can replace the huge bitonic sorter. This design consists of the synaptic array and accumulator/output logic. Since the synaptic array is the same, only the accumulator and output logic is explained.

2.3.1 Accumulator and Output Logic

The accumulator is implemented as a parallel accumulative counter that counts the number of pulses coming in at every instant and accumulates the count to a stored value, along with threshold comparison and output spike generation logic built into it. An efficient $(8N)$-input accumulator is designed by integrating an $(8N-1)$-input parallel combinational counter and a $(log_2 8N+1)$-bit adder into one design. The remaining 1 input bit is provided as carry-in to the adder. Assuming N to be a power of 2, this design hides the latency of the adder and incurs almost similar delay as the combinational counter. This design is inspired from [93].

Figure 2.6 shows the block diagram for a 16-input accumulator, with integrated output spike

generation. All adder inputs are utilized as much as possible by sending input bits to carry-in of the different stages. Comparison with threshold is simply implemented by initializing the $(log_2N + 1)$-bit register with a signed 2's complement representation of (-θ). Thus, comparison with threshold is essentially checking the $(log_2N + 1)^{th}$ bit of the output. If it is '0', the output is non-negative, i.e., the accumulated value has crossed the threshold and complement of $(log_2N + 1)^{th}$ bit is simply the output spike (z). A pulse signal is enforced at the output spike by using the output line (z) as a control signal to the multiplexer which resets the register back to (-θ) at the next cycle, thereby bringing down the output line to 0 at the next cycle. Here, N is assumed to be a power of 2, which gives full utilization of all the component adder inputs. Another assumption made here is that threshold (θ) is less than N. This is typically the case for practical purposes. If threshold is equal to N, then an extra full adder and register bit need to be added.

Observation: The input is still fanned out to 8 lines which affects the size of the accumulator. For RNL response function, the different up-steps at different times can all be time-multiplexed into a single line with the help of a simple finite state machine (FSM) at every synapse. The complexity of this FSM is similar to the synaptic array; however, the accumulator is now significantly reduced in size.

2.4 Optimized Pulse Temporal Logic (PT+)

Figure 2.7 shows the block diagram for a ramp-no-leak neuron implemented using an optimized pulse temporal logic. Both input spikes (x_1,...,x_N) and output spike (z) are represented using single cycle pulses as before. However, the synaptic array with fan-out of input synapse into multiple lines has been replaced with an FSM at each input synapse. This FSM generates the appropriate up-step in response function amplitude at every time instant. The FSM's output line is just 1-bit wide since RNL involves only unit magnitude up-steps. The accumulator and output logic is the same as in PT neuron. Note that this modification reduces accumulator size significantly due to smaller input size. This design consists of the following two main components, namely FSM and accumulator/output logic. Since the only change in accumulator is the input size, only the FSM is discussed here.

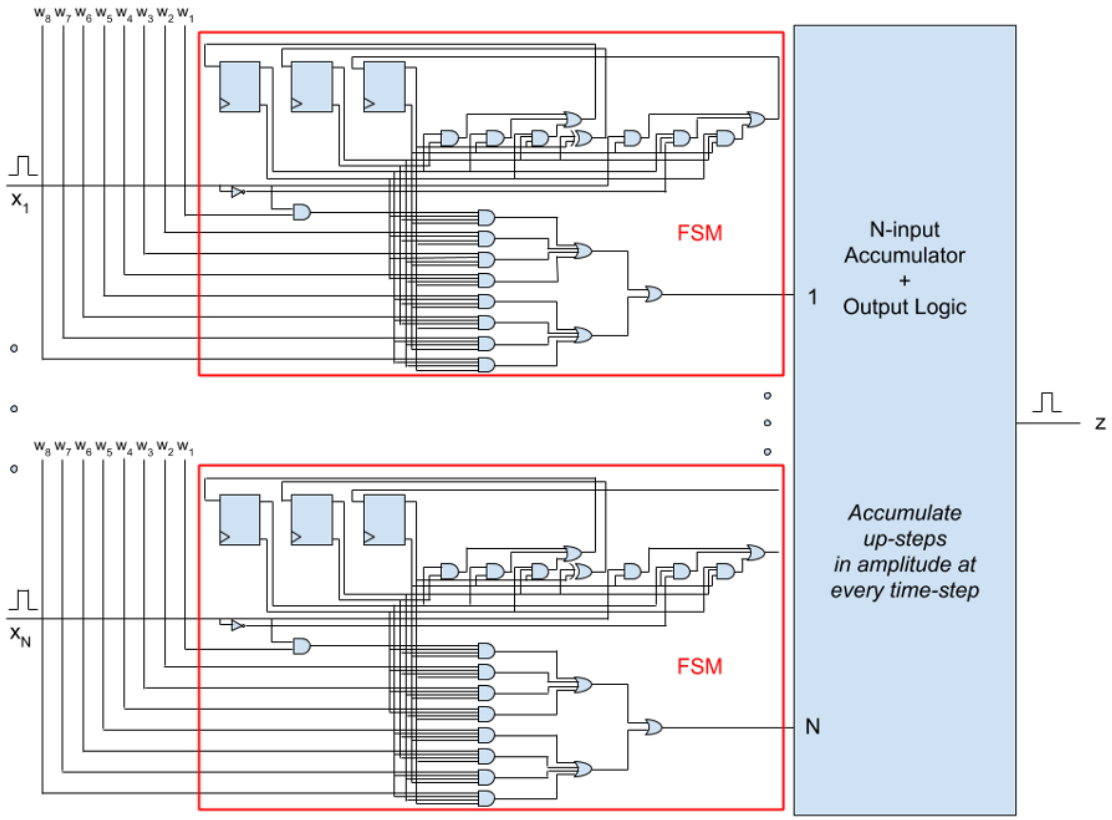


Figure 2.7: An optimized pulse temporal RNL neuron with N synapses and weights ranging from 0 to 8. Note that the synaptic array is replaced by a set of finite state machines.

2.4.1 Synaptic Finite State Machine

As in both ET and PT, weights here are also provided as 8-bit thermometer-encoded input with the number of trailing ones denoting the value. The main difference between PT and PT+ neuron designs is that the synaptic array in the former is replaced by an FSM in the latter. As mentioned above, this FSM for each synapse is responsible for generating the appropriate up-step for RNL response function at every time step, based on the synaptic weight. The state diagram from which the FSM logic in Figure 2.7 is derived, is shown in Figure 2.8.

There are 8 states from S0 to S7 representing the 8 time-steps for RNL response function, starting from t=0. At each time-step, the output up-step value is equal to the corresponding weight bit. This is due to thermometer encoding of weights. For example, if the weight is 3 (00000111), RNL function will have up-steps of 1 from t=0 to t=2 and then no up-steps (or up-steps with value 0). The FSM is initialized to S0 at reset (not shown in the diagram). Note that a

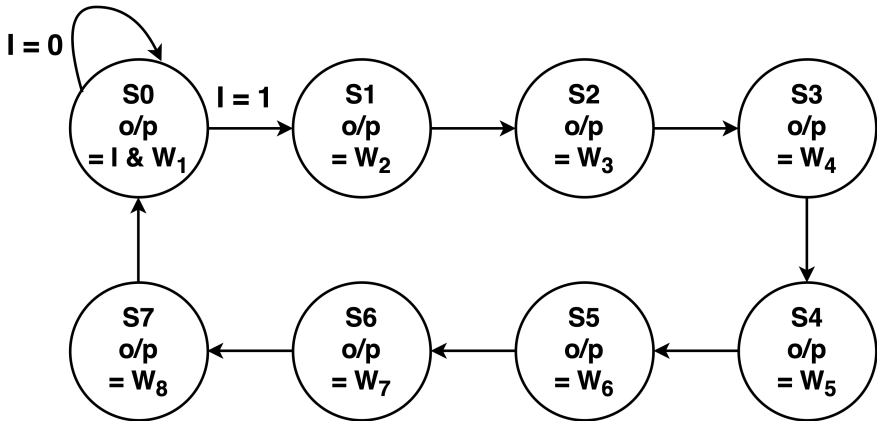


Figure 2.8: FSM state diagram with eight states assuming maximum weight value of 8. 'I' represents input spike pulse and 'W_i' denotes i^{th} weight bit - note that weight bits are 1-indexed.

state transition from S0 to S1 is triggered only when an input spike arrives and then all the other states are reached sequentially until it falls back to S0 again. Due to pulse encoding of input spike, S0 will not transition again. Also, the output at S0 is enabled only when the input pulse stays at a value of 1, thereby eliminating repeated unnecessary outputs at S0. The mechanism mentioned above essentially time-multiplexes all the pulses across the 8 synaptic lines in PT RNL neuron into 1 line, thereby reducing the input size of accumulator by a factor of 8.

Observation: Weights are stored separately as 8-bit thermometer values. Can those weights be rolled back into 3 bits stored within the above FSM, thereby eliminating need for separate weight storage? The proposed TNN microarchitecture in Chapter 3 includes this optimization.

2.5 Results and Discussion

In this section, we perform a comparison of the three Temporal Logic -based neuron implementations based on two types of evaluation.

2.5.1 Qualitative Gate Count Equations

Characteristic gate count equations for a neuron with N synapses are derived for area, computation time and power. The methodology used is as follows:

- Gate count can be used as a surrogate for area as area is proportional to the number of logic gates instantiated.

	ET RNL Neuron	PT RNL Neuron	PT+ RNL Neuron
Area (A)	$4N(log_2 8N)^2 + 4Nlog_2 8N - 5N$	$83N + 8log_2 8N + 8$	$42N + 8log_2 N + 8$
Computation Time (T)	$11.5(\log_2 8N)^2 + 11.5 \log_2 8N - 115$	$138 \log_2 8N + 45$	$138 \log_2 N + 92$
Power Consumption (P)	$12N(log_2 8N)^2 + 12Nlog_2 8N - 15N$	$249N + 284log_2 8N + 284$	$126N + 284log_2 N + 284$

Figure 2.9: Qualitative gate count characteristic equations for area, computation time, and power for the ET, PT, and PT+ TLD neuron implementations.

- Time for a single computation can be calculated using the critical path delay as follows. These delays are reported in gate delays or the number of gates in the corresponding path. Assume, without loss of generality, that the latest input spike arrives at or before 15 clock cycles with respect to the first input spike. This is in accordance with neuroscience-backed claim that human brain typically operates within a resolution of 3 to 4 bits.

$$T = 15 * \text{(critical path delay)}$$

- Gate count can also be used for estimating static power consumption, whereas number of gate transitions can be used as an estimate for dynamic power consumption. Note that these estimates are overly conservative since spikes are sparse in general (typically only 10% of the lines have spikes), thereby reducing the dynamic power consumption significantly.

Figure 2.9 summarizes the gate count equations for area, computation time, and power derived for a single neuron with N synapses. Some key observations are:

- Area and power consumption reduce from O($N(log_2 N)^2$) to O(N) from ET to PT neuron. Both area and power further reduce to almost half from PT to PT+ neuron.

- Computation time reduces from O($(logN)^2$) to O($logN$) from ET to PT neuron. With PT+, it further reduces by ∼50% for tens of synapses, ∼30% for hundreds of synapses, and ∼20% for thousands of synapses.

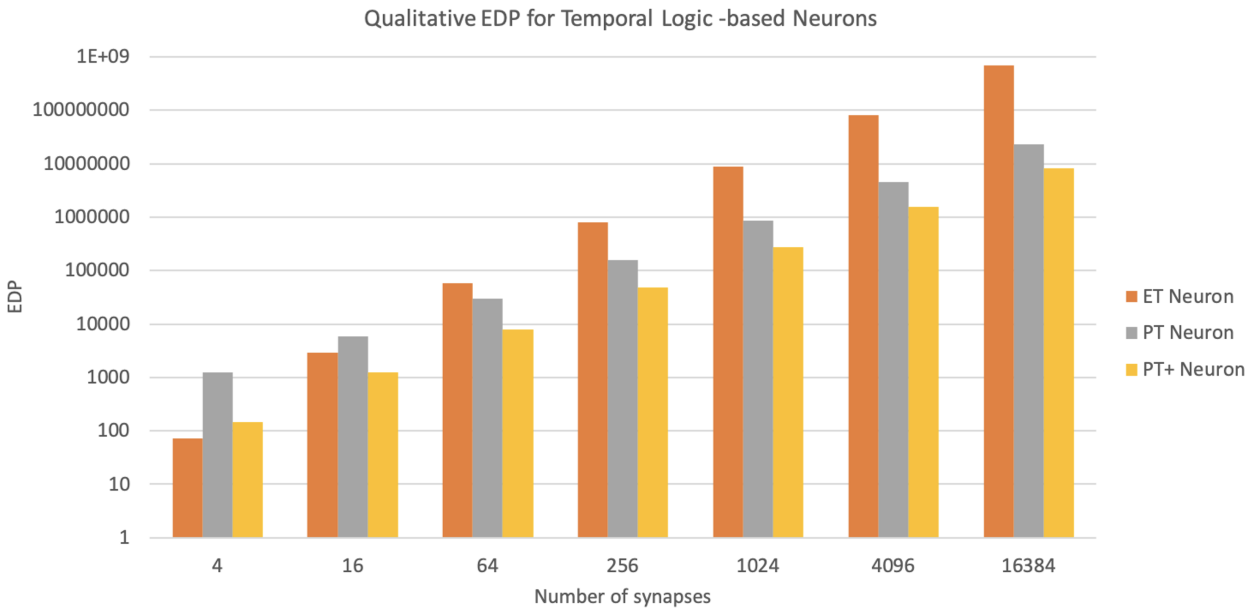


Figure 2.10: Qualtitative energy-delay product (EDP) for the ET, PT, and PT+ TLD neuron implementations. EDP on Y-axis is on log scale and unitless since it is based on gate count. Synapses are varied from 4 to 16384.

- Relative to ET, PT significantly improves hardware complexity by orders of magnitude due to differences in spike representation strategies. The gap between PT and PT+ that both use pulse-based strategy is evident yet not as large as the gap between ET and PT.

In order to estimate hardware complexity for very small upto very large number of synapses, varying values of N ranging from 4 upto 16,384 are substituted in Figure 2.9 equations. A reasonable metric to summarize hardware efficiency is energy-delay product (EDP), which is derived by multiplying power with square of the computation time. The results are plotted in Figure 2.10. The trends in this graph can be used to compare against actual post-synthesis results in the next subsection to corroborate the qualitative characteristic equations.

2.5.2 45nm CMOS Post-Synthesis Results

All models are implemented in System Verilog and synthesized at 100 kHz clock frequency using open-source Nangate 45nm standard cell library [94] with Synopsys Design Compiler. The resulting post-synthesis EDP results are shown in Figure 2.11. Synapses are only varied from 4 to 256 for fast netlist generation runtimes, and this also tends to be the reasonable range for

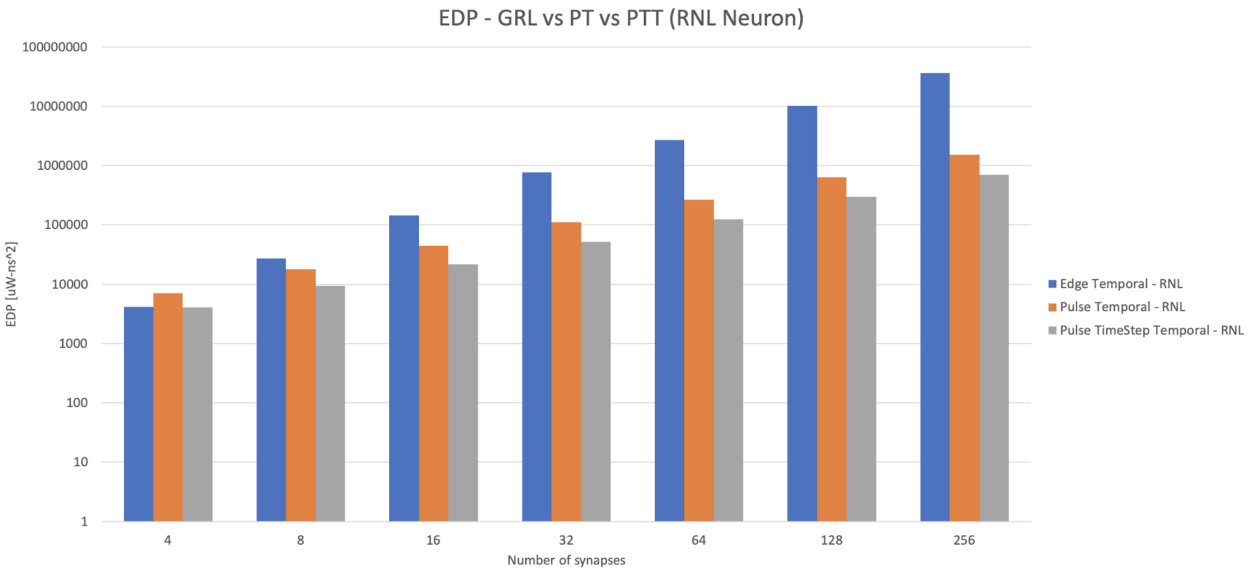


Figure 2.11: 45nm post-synthesis energy-delay product (EDP) for the ET, PT, and PT+ TLD neuron implementations. Synapses are varied from 4 to 256. Note that EDP on Y-axis is not unitless and measured in μW-ns^2.

actual implementation. Here, we are not concerned with the exact post-synthesis numbers as the neuron design in Chapter II supplants the PT+ design here. Rather, the relative trends are of importance here. From Figure 2.11, it can be seen that the relative trends between ET, PT, and PT+ match almost perfectly with the predicted qualitative trends. For N=256, PT is about 30x better than ET, and PT+ is about 3x better than PT in EDP. These results corroborate the characteristic equations in Figure 2.9, thus rendering them useful for quick assessment of gate count complexity for arbitrary values of N.

Chapter 3

Feedforward Temporal Neural Networks

This chapter builds on Temporal Logic Design, specifically optimized pulse temporal (PT+) approach, to define a microarchitecture implementation framework for direct CMOS implementation of feedforward Temporal Neural Networks (TNNs). TNNs leverage insights from both conventional computer systems as well as biological neocortex systems. Both are hierarchical systems with multiple levels of abstraction. The hierarchical TNN architecture is composed of multi-synapse *neurons*, multi-neuron *(mini)columns*, and multi-column *layers*, resulting in multi-layer feedforward TNNs, as illustrated in Figure 3.1. In this chapter, we adopt a "point neuron" model, however this hierarchy is further enhanced by incorporating more fine-grained levels in Chapter 4 for increased biological plausibility. This chapter focuses on direct CMOS implementation of TNNs and building a scalable microarchitecture framework for the same.

3.1 TNN Organization and Operation

3.1.1 Temporal Encoding and Processing

A distinctive attribute of TNNs involves the use of temporal encoding. With temporal encoding, information is represented by relative timings of spikes. In a TNN, computation occurs in volleys or waves of spikes. A volley consists of at most one spike per synaptic input (some may have no spike). The value represented by each input spike is based on its spike time relative to the first spike in the volley.

Temporal encoding is illustrated in Figure 3.2. The first spike in the volley represents a value

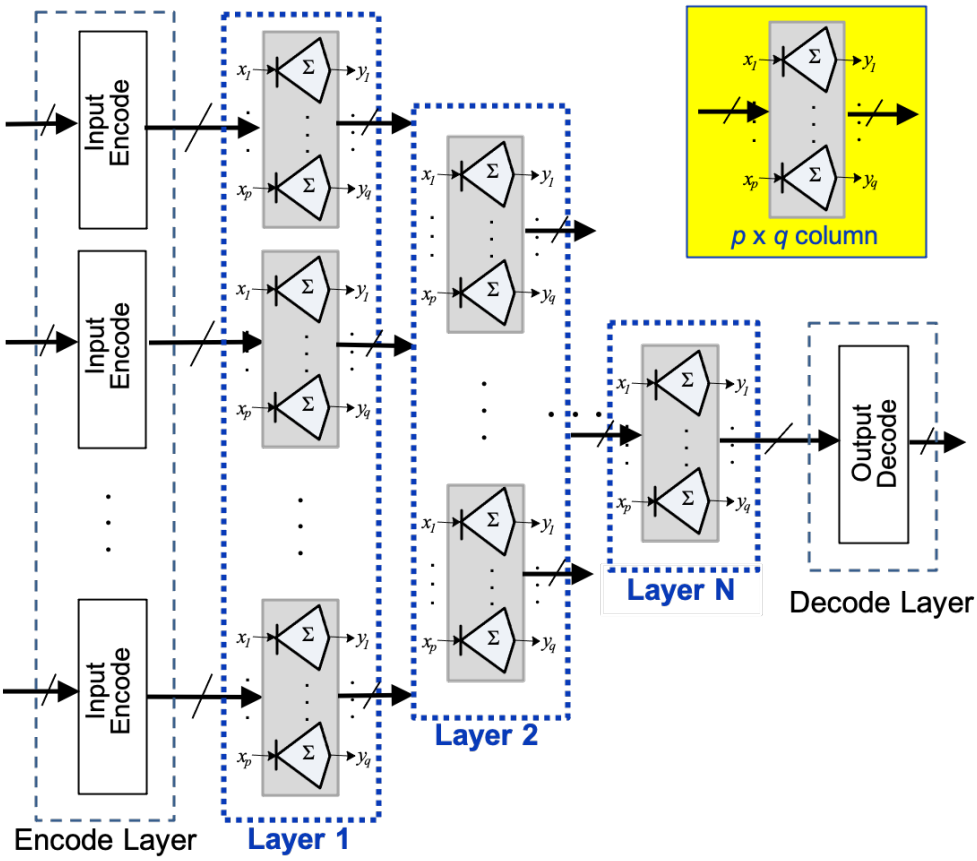


Figure 3.1: Generic Feedforward TNN Organization: with stacking of multi-neuron columns in each layer and cascading of multi-column layers into a multi-layer TNN, with dimensions: **TNN{[k_1x(p_1xq_1)] + [k_2x(p_2xq_2)] + ... + k_nx(p_nxq_n)]}** where k_i denotes the number of (p_ixq_i) columns in layer i. TNN is bookended by input-encode and output-decode layers.

of 0 and subsequent spikes are assigned increasing values based on increasing delays relative to the first spike. If no spike occurs on an input, it is assigned the symbol "∞". It has been proposed, with experimental support, that computational volleys are synchronized by inhibitory oscillations at gamma frequencies (50-100 Hz) [95]. These gamma oscillations divide neocortical processing into alternating inhibitory and excitatory phases. The inhibitory part of the cycle essentially performs a reset function and the excitatory part performs computation. This leads to a temporal coding interval of 5-10 msec during which spikes are transmitted in a coordinated volley. Neuron spiking behavior has been shown to be repeatable to within 1 msec [96, 97]. A 5-10 msec window combined with the 1 msec encoding precision implies only 3-4 bits of precision is needed within the encoding window. Consequently, the computing model used here is based

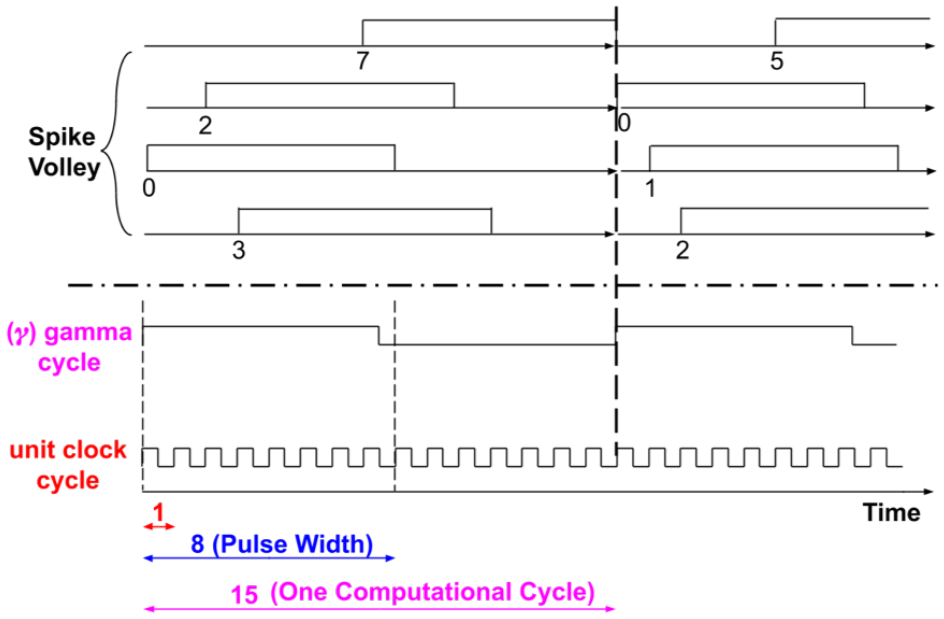


Figure 3.2: Temporal Encoding and Processing

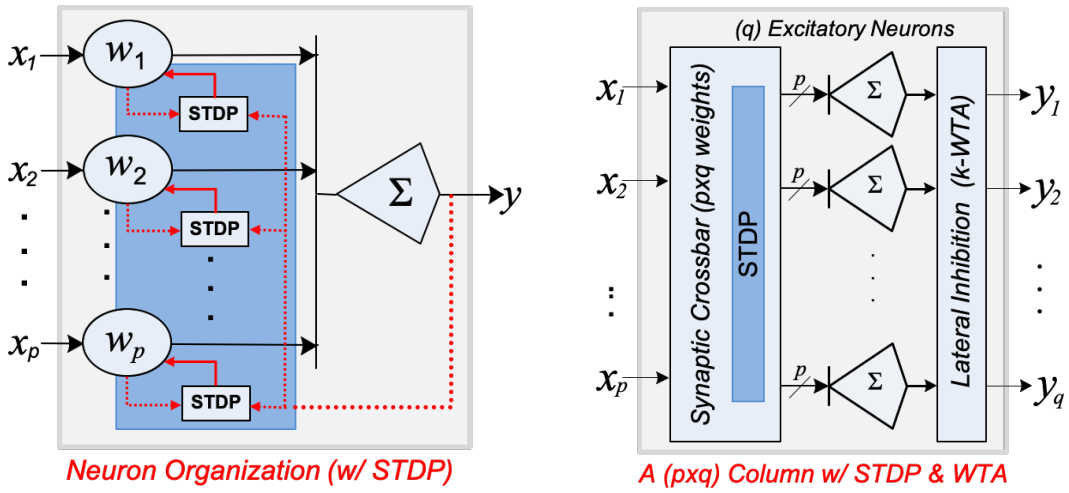
on low resolution integers.

In this work, temporal encoding and processing are employed with the actual hardware clock cycle directly serving as the basic time unit. This work uses 3 bits of precision for temporal encoding and synaptic weights. Spikes in a volley are represented using pulses, a form of unary encoding, and volleys are separated using gamma clock cycles. With unary encoding, it takes up to 7 time units to encode a 3-bit value. To allow additional time for a column to process a spike volley, the gamma cycle comprises of 15 time units. This is explained further in Section 3.2.2.

In summary, the proposed design uses two clocks. The *unit clock* is the finest temporal resolution in the computation model and is also the synchronizing clock used in the digital hardware. The *gamma clock* frames the computing window and is the time required to communicate and process spike volleys and update synaptic weights (i.e., one computation wave).

3.1.2 Key TNN Building Blocks

The most fundamental TNN building block is a neuron. As shown in Fig. 3.1.2, each neuron has p synaptic inputs and one output. Each synaptic input carries a synaptic weight, which is updated locally based on the relative timing of the incoming spike to that synapse and the outgoing



(a) Neuron: p Synapses, STDP

(b) Column: q Neurons, WTA

Figure 3.3: Key TNN Building Blocks

spike from the associated neuron body. The rules for updating synaptic weights constitute the STDP learning method - the key building block that imparts TNNs their functionality. Through STDP, a neuron learns an input feature by adapting its synaptic weights to closely match the corresponding input pattern.

The smallest operational building block is a column which, in itself, is a fully-functional TNN. As shown in Fig. 3.1.2, a column is a stack of q parallel neurons. Every neuron in a column shares the same set of p inputs, known as a *receptive field*. A $p \times q$ synaptic crossbar contains $p \times q$ synaptic weights, each of which is updated by STDP. On the output side of the q neurons, one winner-take-all (1-WTA) *lateral inhibition* is performed by selecting the earliest spiking neuron from among the q neurons as the one winner. Output spiking is disabled for non-winning neurons. This introduces competition among the neurons and enables the column to learn a set of distinct features local to its input receptive field.

This chapter presents the CMOS implementation of a neuron (Section 3.2) and a (mini)column (Section 3.4). In Section 3.3, STDP rules for updating synaptic weights are discussed. The baseline STDP method is unsupervised. We also introduce a variation, called *reinforcement* STDP, which is similar to the *reward modulated* STDP in [98]. Post-synthesis and online learning evaluations are performed in Sections 3.5 and 3.5.3 respectively.

3.2 Neuron Implementation

This work adopts the widely used Spike Response Model [70] SRM0. This section presents the components of this excitatory neuron model and their detailed gate level designs.

3.2.1 Synaptic Response Functions

A *synapse* connects the *axon* (output) of a pre-synaptic neuron and a *dendrite* (input) of the post-synaptic neuron. An SRM0 neuron takes multiple input spikes and generates a response function for each spike based on its corresponding synaptic weight. All the individual response functions are then added to form the neuron's membrane potential. When (and if) the membrane potential crosses a threshold, the neuron fires an output spike on its axon. In this work, we adopt the ramp-no-leak (RNL) function for its temporal computational benefits and implementation efficiency [99, 100]. The RNL function increases by a unit step at every time unit until it reaches its peak and then remains constant until it is reset prior to the next computation cycle. The "ramp" allows responses from different synapses to be distributed temporally based on the synaptic strengths (weights), which proves to be particularly powerful for TNNs that operate temporally. The no-leak model is based on arguments that the leak is primarily a reset mechanism [91, 92]. For silicon implementations, there are simpler ways to reset at the end of each gamma cycle.

3.2.2 Synapse Modeling

Fig. 3.4 shows the block diagram for the proposed SRM0 neuron implementing RNL response function. Its operation consists of three main stages: 1) temporal arrival of input spikes, 2) serial thermometer readout of RNL response functions based on the corresponding synaptic weights, and 3) binary accumulation of thermometer-coded response functions into the membrane potential. Synapses are implemented as finite state machines (FSMs) operating as binary counters. If the maximum weight is w_{max}, the number of counter bits is $ceiling(log_2(w_{max}+1))$. The counter has three modes, two controlled by STDP (described in Section 3.3): increment (up to w_{max}) and decrement (down to 0). The third *readout* mode is controlled by the input pulse. Readout mechanism is meticulously integrated into the same FSM used for storing synaptic weight and is described below.

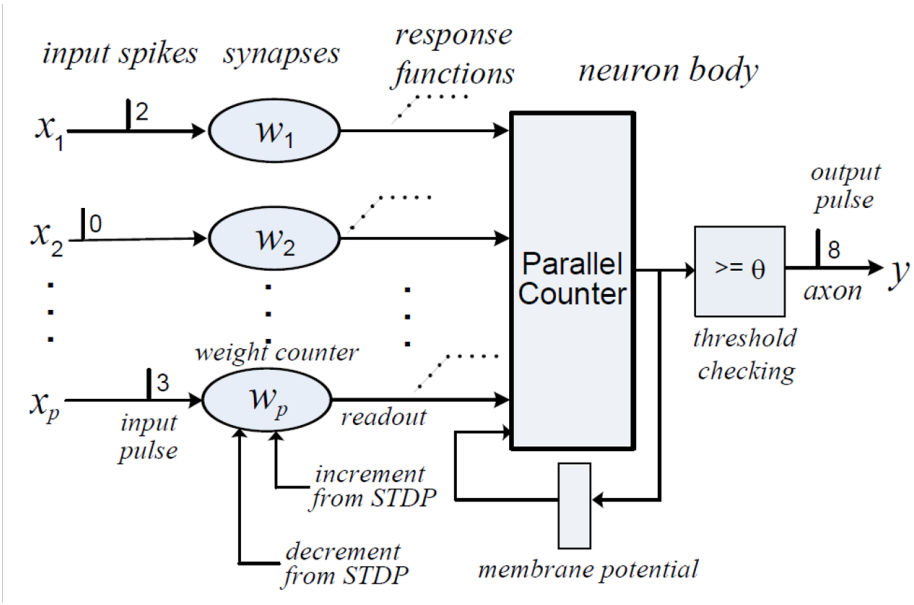


Figure 3.4: SRM0 Neuron with RNL Response Function

As will become apparent, synapses dominate hardware complexity and hence the synapse design must be highly optimized. Our approach uses a pulse of width $w_{max} + 1$. The input pulse directly controls the counter readout. When the leading edge of an input pulse occurs (0→1 transition), the weight counter is decremented and an output of 1 is emitted each unit clock cycle until the counter reaches 0. This essentially converts the binary weight value in the counter to a serial thermometer code. After the counter reaches 0, it wraps around to w_{max} and continues to count down until the trailing edge of the input pulse (1→0 transition) when the weight in the counter is restored to its original value. Thus, once an input spike arrives, readout takes an additional 7 cycles. (Although we assume $w_{max} = 7$ in this work, this technique can be generalized to any w_{max}.) STDP (Section 3.3) takes another cycle. These coupled with 7 cycles for encoding give rise to a gamma period of 15 clock cycles.

In summary, a synapse and its weight are implemented with a counter FSM that can 1) increment, saturating at w_{max}; 2) decrement, saturating at 0; and 3) wrap-around decrementing, emitting an output of 1 prior to wrapping around and then a 0 thereafter. Note that this synapse design preserves the original weight value while doing RNL readout, which eliminates the need of a separate SRAM for weight storage and access.

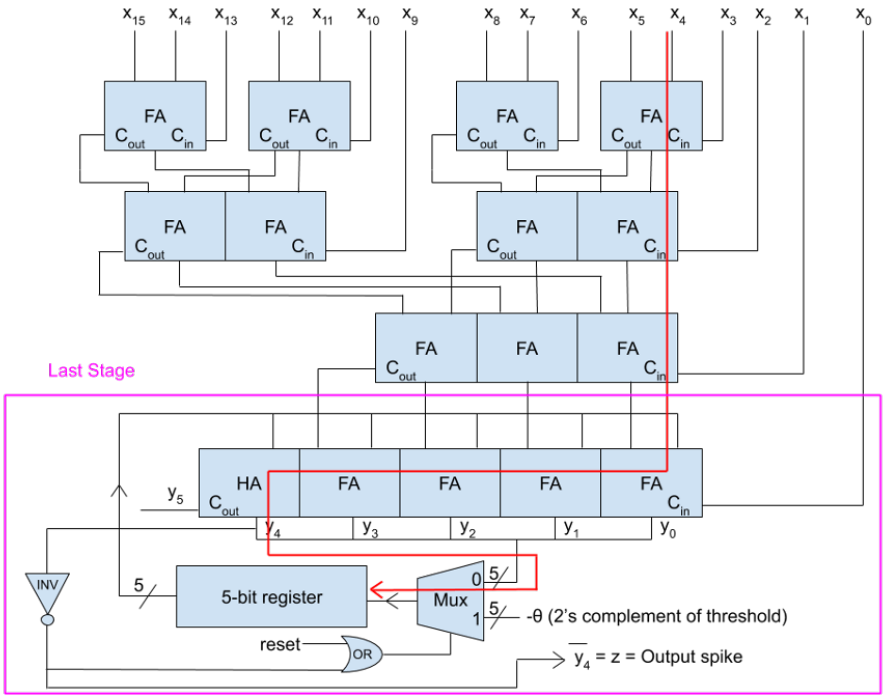


Figure 3.5: Neuron Body with 16 Synapses

3.2.3 Neuron Body

As described in Chapter 2, the neuron body is implemented as a parallel counter that adds the thermometer coded weights coming from the synapses, cycle by cycle, thereby accumulating the membrane potential as a sum of RNL response functions. When (and if) the parallel counter output reaches the threshold θ, an output spike is emitted during that cycle.

Based on Parhami [93], the membrane potential accumulator can be efficiently implemented using ripple carry adders by integrating a (p-1)-input parallel combinational counter and a ($log_2 p + 1$)-bit adder into one design. Fig. 3.5 shows the design for a 16-input accumulator, with integrated output spike generation. For a p-input accumulator, p-1 inputs are accumulated into a ($log_2 p$)-bit output, which is then added to the previous stored ($log_2 p + 1$)-bit value from the register with the one remaining input bit acting as carry-in. Note that the configuration in Fig. 3.5 allows all adder inputs to be efficiently utilized and is most optimal when p is a power of 2. The accumulating register is initialized with (signed 2's complement) -θ at every gamma cycle, which eliminates the need for any comparator for output spike generation. The $(log_2 p + 1)^{th}$ bit of the output indicates if the accumulated body potential has crossed the threshold and triggers

Table 3.1: STDP Update Rules

Input Conditions		Weight Update
$x(t) \neq \infty$; $z(t) \neq \infty$	$x(t) \leq z(t)$	$\Delta w = +B(\mu_{capture}) * max(F(w), B(\mu_{min}))$
	$x(t) > z(t)$	$\Delta w = -B(\mu_{backoff}) * max(F(w), B(\mu_{min}))$
$x(t) \neq \infty; z(t) = \infty$		$\Delta w = +B(\mu_{search})$
$x(t) = \infty; z(t) \neq \infty$		$\Delta w = -B(\mu_{backoff}) * max(F(w), B(\mu_{min}))$
$x(t) = \infty; z(t) = \infty$		$\Delta w = 0$

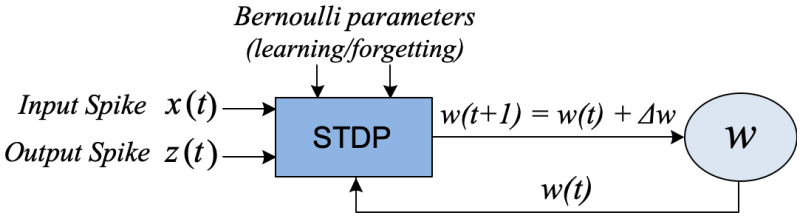


Figure 3.6: Local STDP Update Process

a 3-bit counter to generates an 8-cycle wide pulse (output spike).

3.3 STDP & R-STDP Implementation

STDP is a distinctive feature of TNNs. STDP learning is unsupervised and local to each synapse. It can perform inference and online continual learning at the same time. In this work, we propose an STDP design that is both effective in learning and implementable using standard CMOS.

3.3.1 Proposed STDP Update Rules

Our learning method is a customized version of the classic Spike Timing Dependent Plasticity (STDP) [101]. STDP is implemented locally at each synapse as shown in Fig. 3.6. The proposed STDP learning rules are summarized in Table 3.1. Here, x(t) and z(t) represent input and output spiketimes respectively. Δw denotes change in weight and B(μ) represents a Bernoulli random variable with probability μ.

STDP update rules are divided into four major cases, corresponding to the four combinations of input and output spikes (represented by x(t) and z(t) respectively) being present ($\neq \infty$) or absent ($= \infty$) . When both are present, two sub-cases are formed based on the relative timing of the input and output spikes in the classical STDP manner [101]. In effect, a synaptic weight

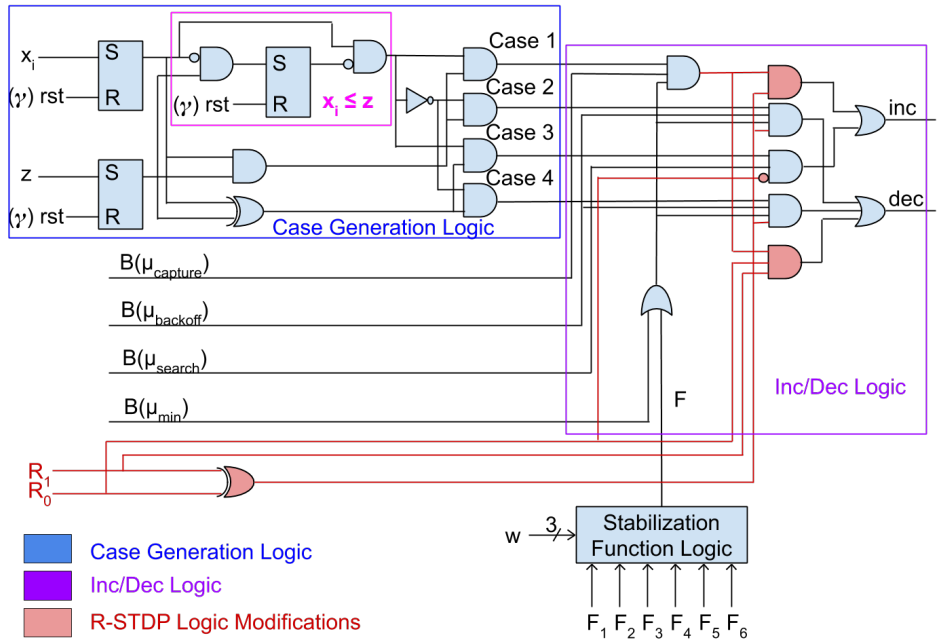


Figure 3.7: STDP and R-STDP Logic Implementation

is incremented (strengthened) if there is an input spike and it either contributed (Case 1) or can potentially contribute (Case 3) to the output spike; else it is decremented.

The STDP update function either increments the weight by Δw (up to a maximum of w_{max} = 7), decrements the weight by Δw (down to a minimum of 0), or leaves the weight un-changed. The Δ values (1, 0 or -1) are defined using Bernoulli random variables (BRVs) with parameterized learning probabilities denoted as B(μ) with a descriptive subscript. $F(w)$ is a stabilization function (=$B((w/w_{max})(1 - w/w_{max}))$) which makes the weights "sticky" at both ends (0 and 7) [102, 103].

3.3.2 Proposed STDP Implementation

The proposed STDP logic implementation is shown in Fig. 3.7. It generates 2 control signals (increment/decrement) at the output that feed into the synaptic weight counters described in Fig. 3.4. Note that STDP updates (and the associated resets) are performed at the end of a computational cycle (or onset of next gamma clock); inputs for the new computational cycle begin a unit clock cycle later. The proposed STDP logic implementation can be partitioned into three components.

- Case Generation Logic: The per-synapse case generation logic compares the synapse's input spiketime (x_i) with its post-synaptic neuron's output spiketime (z) and generates 4 control signals corresponding to the 4 cases in Table 3.1. Case 5 is implicitly invoked when none of the other 4 cases is a 1. The logic equations implemented for the 4 STDP cases are given below. Note that $((x_i \leq z))$ is implemented here using a much simpler *temporal* comparator as opposed to a binary comparator. If z arrives prior to x, the output is 0; else x is allowed to pass.
  - Case 1: $(x_i \leq z).(x_i).(z)$
  - Case 2: $(\overline{x_i \leq z}).(x_i).(z)$
  - Case 3: $(x_i \leq z).(x_i \oplus z)$
  - Case 4: $(\overline{x_i \leq z}).(x_i \oplus z)$
- Stabilization Function Logic: This logic selects 1 BRV from a set of finite BRVs generated by $F(w)$, based on the synaptic weight. For $w_{max} = 7$, there are 6 non-zero BRVs to choose from. The output bit is generated by an 8-to-1 multiplexer controlled by 3-bit weight.
- Inc/Dec Logic: The inc/dec logic assumes 4 BRV inputs from the LFSR network corresponding to the four STDP cases. The *max* operation in Table 3.1 is simply implemented by 'OR'ing 'F' with *min* BRV input. The output of the stabilization logic is used along with the cases from case generation logic to generate *inc* and *dec* outputs.

3.3.3 Proposed R-STDP Implementation

This subsection introduces a variation of the proposed STDP method capable of *reinforcement learning* (R-STDP) [98] that uses an external *reward* signal to drive its learning process in a desired direction. It involves three forms of reinforcement:

- When the column's (non-null) output matches the desired action, *reward* = '1'. It operates as per Table 3.1; except case 3 results in no synaptic weight update.

- When the column's (non-null) output does not match the desired action, *reward* = '-1'. Only Cases 1 and 3 are performed; for Case 1, weight is actually decremented instead of incremented.

- When the column produces no output, i.e., no neuron spikes, *reward* = '0' and only Case 3 operates.

In effect, desired behavior is reinforced and undesirable behavior is repressed using a single *global* reward signal. Note that R-STDP is still applied locally to each neuron and is typically deployed in the final layer of a TNN. The logic modifications for R-STDP are minimal and straightforward as highlighted in Fig. 3.7. *reward* is a 2-bit signal (which encodes '-1', '0' and '1' as '11', '00' and '01' respectively). Unsupervised STDP is invoked when *reward* is '10'.

The implemented STDP/R-STDP learning rules are capable of performing extremely efficient online incremental learning (see Section 3.5.3). To the best of our knowledge, such gate-level hardware-efficient implementations of STDP/R-STDP rules for TNNs have not been presented or published before.

3.4 (Mini)Column Implementation

A column is a fundamental functional unit in TNNs [15, 99], much like ALUs in von-Neumann computers. As shown in Fig. 3.1.2, a $p \times q$ column contains q excitatory neurons and a synaptic crossbar connecting the p inputs to the q neurons via $p \times q$ synapses. A column supports unsupervised learning via STDP or supervised learning via R-STDP, followed by WTA lateral inhibition to assist in weight convergence. A single column supported by STDP/R-STDP and WTA becomes a fully operational TNN, capable of performing online continual learning and inferencing. Columns can be used to create larger TNNs by stacking multiple columns to form a multi-column layer, and by cascading multiple layers into a multi-layer TNN.

Winner-take-all (WTA) inhibition is a distinctive feature of a column that selects the first spiking neuron and allows its output spike to pass through intact, while nullifying other neurons' outputs. Fig. 3.8 shows the logic diagram for 1-WTA inhibition across q neurons in a column. The inhibition operation is performed by a latch-based less-than-or-equal temporal comparison

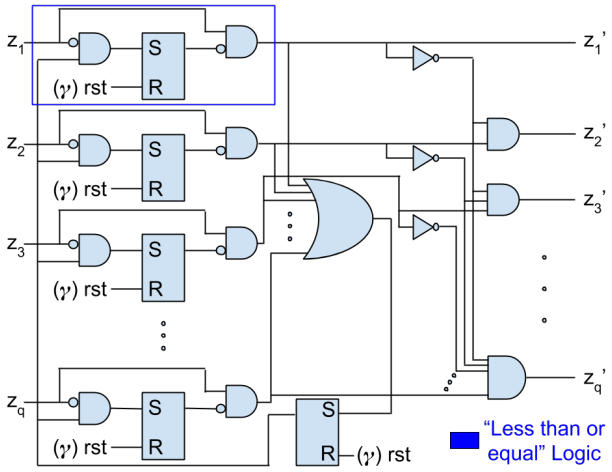


Figure 3.8: WTA Inhibition for a Column of q Neurons

unit. The first spike is found through a large 'OR' gate, or a tree of small OR gates, (performing a temporal 'min' function) and is fed back through a latch which holds the signal at 1 until the next gamma cycle. Any input pulse coming to the latch after this signal is blocked, so only the first spikes are passed. Tie breaking is implemented by selecting the spiking neuron with the lowest index.

3.5 Results and Discussion

Scalable neuron and column designs are implemented in System Verilog; synthesis results are generated using open-source 45nm Nangate standard cell library [94] and Synopsys tools. Hardware design is evaluated in terms of area (A), critical path delay (D), computation time (T) and power (P). T is the time taken to process one input (one *gamma* cycle). Power, performance, area (PPA) results for multilayer feedforward TNN designs will be discussed in Parts III and IV of this dissertation.

3.5.1 Gate-Level Characteristic Scaling Equations

We derive characteristic scaling equations (Table 3.2) for A, D (neuron), T (column) and P based on gate count ('AND' equivalents) and number of signal transitions, parameterized in terms of number of neurons (q) and number of synapses per neuron (p). The procedure is as follows: 1) Gate count is used as a surrogate for area and static power. 2) Number of gates in the critical path

Table 3.2: Characteristic scaling equations for A, D/T and P for a neuron with p synapses and a $p \times q$ column.

Metrics	Neuron	Column
A	$102p + 8log_2p + 36$	$102pq + 8qlog_2p + 44q + q^2$
D / T	$6log_2p + 4$	$90log_2p + 60$
P_{static}	$102p + 8log_2p + 36$	$102pq + 8qlog_2p + 44q + q^2$
$P_{dynamic}$	$204p + 185log_2p + 241$	$204pq + 185qlog_2p + 257q + 2q^2$

Table 3.3: A, T and P (in 45 nm CMOS) for three column sizes of 64×8, 128×10, 1024×16, with STDP and R-STDP.

	Synapses x Neurons	**Gate Count**	**Area** [mm^2]	**Comp. Time** [ns]	**Power** [mW]
STDP	64 × 8	51,824	0.05	28.95	0.25
	128 × 10	128,658	0.13	32.40	0.62
	1024 × 16	1,639,020	1.65	42.30	7.96
R-STDP	64 × 8	54,384	0.05	28.95	0.26
	128 × 10	135,058	0.14	32.40	0.65
	1024 × 16	1,720,940	1.75	42.30	8.36

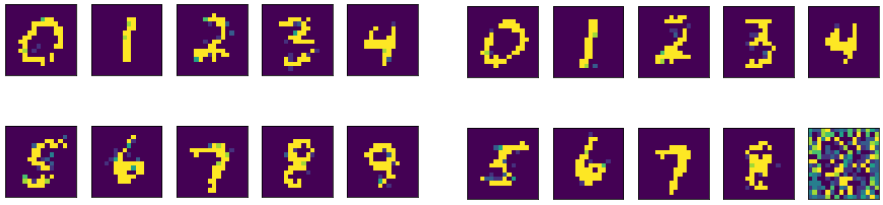
is used for D; T is derived using the *gamma* period, $T = 15 * D$. 3) Number of gate transitions is used for dynamic power. These equations can serve as a powerful tool for design space exploration, as they can help estimate the hardware complexity of arbitrary TNN designs.

From our gate-level analysis for a single neuron, synapses (including STDP) constitute almost 90% (50% synaptic FSM and 40% STDP logic) of the entire neuron complexity while the neuron body accounts for the remaining 10%. In a single column, neurons constitute almost the entirety of column complexity; WTA incurs negligible cost (less than 1%).

3.5.2 45nm Post-synthesis Evaluation of Column Designs

Area, power and critical path delay are obtained directly from Design Compiler, and computation time is derived as earlier. We use the low power process corner for synthesis with operating frequency of 100 kHz and voltage of 0.95 V.

Table 3.3 presents 45 nm post-synthesis results for three column configurations for STDP and R-STDP learning rules: 1) a small 64×8 column; 2) a medium 128×10 column; and 3) a large 1024×16 column. The *gamma* cycle for the large 1024×16 column with around 1.7M gates is 42.3 ns (23.64 MHz). It has an area and power footprint of 1.65 mm^2 and 7.96 mW with STDP in 45nm, less than 1% of the area and power budget of typical mobile SoCs. Note that the overhead



(a) Trained for Digits 0 - 9 (b) Trained for Digits 0 - 8

Figure 3.9: Synaptic weight matrices converge to image centers resembling MNIST digits in 10,000 samples.

Figure 3.10: Online Incremental Learning: STDP learns a previously unseen input number '9' within 500 examples.

for R-STDP is minimal; it increases die area and power by only 5% relative to STDP while adding supervision to learning.

3.5.3 Online Incremental Learning

In contrast to the typical epoch-based training with global back-propagation, STDP is an online local learning method that processes inputs in a streaming manner targeting online real-time applications. This section uses a subset of MNIST, with images resized from 28×28 to 16×16, to illustrate online incremental learning for TNNs. Because our focus is on online learning, the standard MNIST benchmark protocol for offline training/testing does not apply. From our experiments, using just a single (256×10) column and resized MNIST, several interesting capabilities

of TNNs can be observed.

1. *Online Classification via Centroid Formation*: Fig. 3.5.3 shows the synaptic weights converged to the 10 class centroids via R-STDP, which resemble the corresponding digits. This shows the efficacy of R-STDP in driving the weights towards class centroids.

2. *Fast Training Convergence*: The synaptic weights in Fig. 3.5.3 and Fig. 3.5.3 converged after approximately 10,000 training samples, which implies that TNNs can learn very quickly and can generalize from small datasets.

3. *Online Incremental Learning*: In this experiment, supervised R-STDP training is first performed with only 9 classes (0 to 8) by hiding the digit '9', resulting in the converged weights shown in Fig. 3.5.3. Then the digit '9' is introduced in the input sequence without labels to illustrate the ability to dynamically learn a previously unseen class in an unsupervised fashion. As shown in Fig. 3.10, the synaptic weights converge to the digit '9' after only about 500 testing samples via STDP.

Thus, online incremental learning enables a TNN to adapt to new input data not seen before during the original (offline) training. Continual learning allows a TNN to keep learning and improving its performance concurrently with inference.

Chapter 4

Cortical Column with Reference Frame

As described earlier, Hawkins' theory suggests that cortical columns (CCs) are the fundamental neocortical processing units that embody intelligence. CCs learn information through movement (sensorimotor learning), i.e., as the associated sensing agent moves through an environment. The environment could be physical (e.g., a room, an object such as mug, car, etc.) or metaphysical (e.g., abstract concepts like mathematics, democracy, etc.). CCs model this sensory information and knowledge in structured *Reference Frames* (RFs), and learn models/maps through feedback. While navigating an environment, a CC is continuously predicting what feature it may observe next based on the sensor location. It then compares its prediction against the actual sensed feature to update its model stored in the RF. Each cortical column is computationally powerful and can model complete objects in its RF, regardless of its modality or hierarchical abstraction (see Figure 1.4 in Chapter 1). Multiple CCs targeting different sensory modalities can interact with each other and seek consensus on the output via voting within and across sensory modalities. In contrast to current DNNs that separate training and inference, CCs that store and process information in RFs can support online, concurrent, and continuous learning and can dynamically adapt to sensory input changes.

The continuous feedback loop of predict-sense-update within a reference frame effectively introduces an additional dimension of "memory" that remembers past observations and patterns. This "sequential" behavior is missing in feed-forward TNNs. Hence, TNN mini-columns combined with feedback mechanism implementing the predict-sense-update loop can be used

to build cortical columns. Further, the TNN hierarchy is enhanced here with more biologically plausible active dendrite neuron model.

This dissertation proposes *Cortical Columns Computing Systems (C3S)* as a new genre of computing systems based on cortical columns and reference frames that are:

- Capable of brain-like sensory processing
- Extremely energy-efficient when implemented in standard digital CMOS technology
- Truly "intelligent"
  - Learn rapidly with little data
  - Learn continuously
  - Learn with structure

4.1 Cortical Column: Agent and Reference Frame

This dissertation defines each *Cortical Column*, irrespective of its sensory modality, as composed of two major components: 1) a *Reference Frame* that maintains a "map" of the sensory information, and 2) an *Agent* that achieves goal-oriented behavior based on information from the reference frame and the input signals. This microarchitecture is shown in Figure 4.1.

- Agent comprises of three functional blocks:
  - **Input Encoder**: Acts as a temporal encoder that converts input sensory modalities to spike times.
  - **Compute**: Made up of multiple clustering TNN minicolumns that processes the input to be sent to the reference frame. This is typically similar to the feedforward TNNs in Chapter 3.
  - **Output Decoder**: Performs classification, voting, etc. based on the spiking output of the Compute block. It can also be used to generate movement information for the agent to navigate through the environment.

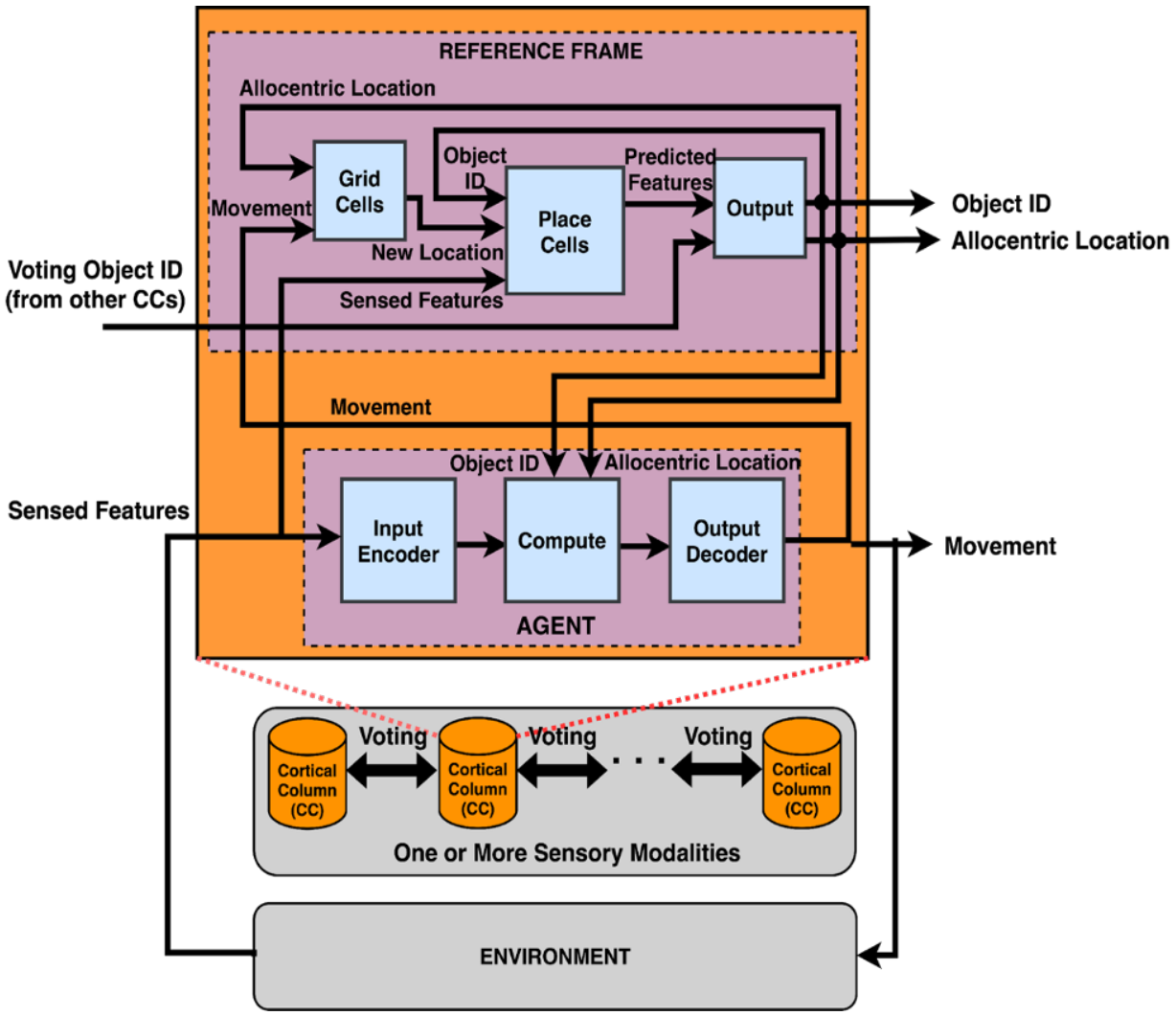


Figure 4.1: Cortical Columns Computing System (C3S) architecture consisting of multiple CCs targeting multiple sensory modalities interacting with each other to form a consensus on the output via voting. Each CC broadly consists of five TNN-style mini-columns: *Where*, *What* and *Output* mini-columns that together implement the Reference Frame (RF), and *unsupervised* and *supervised* mini-columns comprising the agent. For visual object recognition, the respective functionalities of the three RF mini-columns are: derive locations of sensor on the object, map features to locations, and derive the object ID based on the feature map. In contrast to feedforward TNNs, each CC learns through feedback from output and possesses a form of "memory" in the learning process.

- Reference Frame consists of the following three functional blocks:
  - **Grid Cells**: Determines next location of the agent by performing path integration, i.e., accumulating movement information over the course of navigation. It can be

implemented using adders and shifters.

- **Place Cells**: Stores the sensory map as tuples of feature-location pairs for each environment class/ID. It can be used to predict the next feature based on location or vice versa as well as perform inference of environment ID through sequential sensory observations.
- **Output**: Integrates voting information across multiple CCs and determines environment/object ID together with Place Cells.

The *Grid Cells* and *Place Cells* take the outputs from the *Output* as feedback information. The *Grid Cells* block generates location of sensor on the object based on this feedback and the latest movement information from the agent. The *Place Cells* block predicts object features based on the result from the *Grid Cells* block and updates its model based on the actual sensory input and the feedback from the *Output* block. This dissertation focuses primarily on the compute (Chapters 3, 5, 6) within Agent and Place Cells within Reference Frame (Chapter 7), as they are both the dominant contributors to hardware complexity in CCs. These components are implemented using biologically plausible neuronal hierarchy composed of active dendrites (Chapters 8, 9).

4.2 Active Dendrite Hierarchy

Neurons, unlike in Chapter 3, are now composed of active dendrites that perform pattern clustering themselves. Each dendrite comprises of multiple segments, that each acts as individual pattern recognising units. A dendrite selects the output of a segment that achieves the best match. Multiple such dendrites form a neuron and such neurons are used to build C3S minicolumns. Based on this hierarchy, such minicolumns are used to build Place Cells. This hierarchy is described in detail in [68] and illustrated in Figure 4.2, which is an extension of Figure 1.3 from Chapter 1. Despite the shift in hierarchy compared to Chapter 3, the same microarchitecture framework can be used to compose CCs. Microarchitecture implementation results for cortical columns with active dendrite hierarchy are provided in Chapter 9.

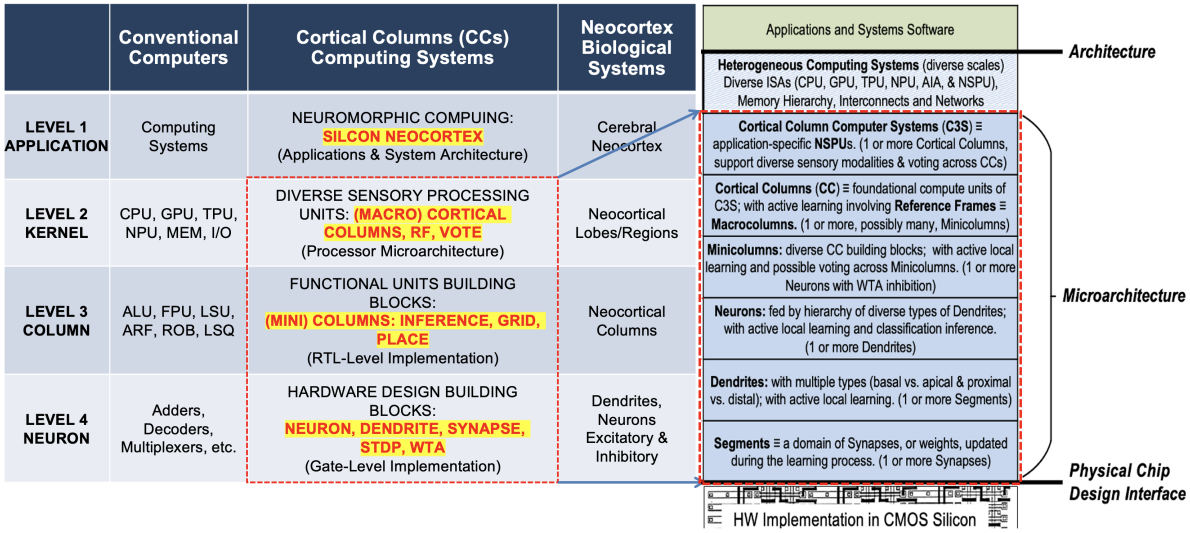


Figure 4.2: Active Dendrite Hierarchy: Neurons, instead of passive synapses, are now composed of active dendrites that perform some amount of pattern clustering themselves. Each dendrite comprises of multiple segments, that each acts as individual pattern recognising units. A dendrite selects the output of a segment that achieves the best match.

Part III

Functional Building Blocks

Chapter 5

TNN7: Custom Macros for TNN Design

This chapter builds on and goes beyond Chapter 3, exploring the potential for creating a customized cell library that utilizes inherent TNN principles to improve the power, performance and area (PPA) of TNN designs. Furthermore, this work serves as the first step towards creating a scalable design framework and toolsuite for building TNN-based neuromorphic processors. We make four key contributions: 1) the TNN design process, including gate-level implementations, is replicated in 7nm predictive CMOS using the ASAP7 Process Design Kit (PDK) [104], and post-synthesis Power-Performance-Area (PPA) results are reported; 2) a set of nine new highly-optimized custom macro extensions to ASAP7, called *TNN7*, that can be used for implementing highly energy-efficient parameterizable TNNs is proposed; 3) significantly improved scaling of PPA as well as synthesis runtime for larger design sizes, achieved by TNN7, is demonstrated; and 4) the hardware complexities of TNN prototypes in [15] for image classification, and [80] for unsupervised time-series clustering, are evaluated and shown to achieve significant improvements using the custom macro extensions, demonstrating the potential of TNNs for energy-efficient sensory processing with online learning.

The proposed nine macros have been designed to target and optimize the primary TNN building blocks, or TNN columns. Fig. 5.1 illustrates the custom macros in a typical pxq column (key TNN building block) with p synapses per neuron and q such neurons, followed by winner-take-all (1-WTA) lateral inhibition. As majority of the TNN computation occurs in the synaptic crossbar, five macros are dedicated for synapses (two for synaptic inference or response

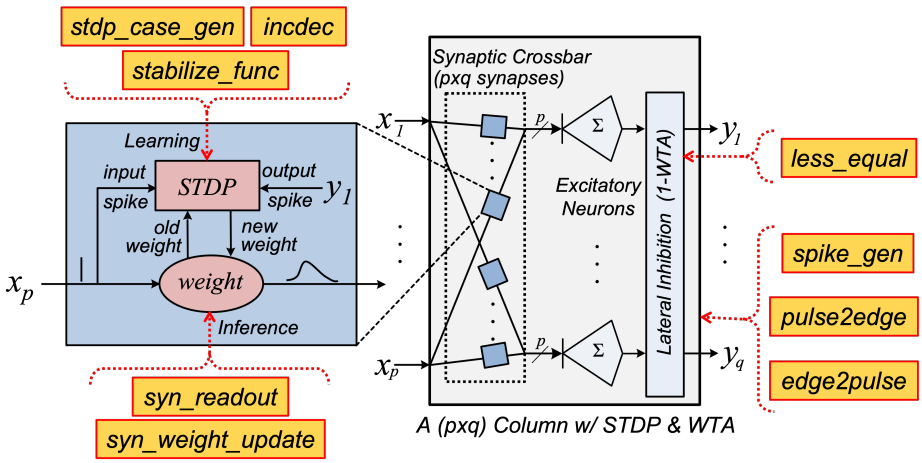


Figure 5.1: Functional components of a *pxq* TNN Column and associated custom macros (highlighted in yellow)

Table 5.1: Proposed Custom Macros

TNN Units	Proposed Macros	Function Description	Figure Label
Synaptic Response	*syn_readout*	Perform RNL readout	Fig. 5.2
	syn_weight_update	Perform weight update	Fig. 5.3
WTA	*less_equal*	Perform temporal inhibit	Fig. 5.4
STDP	*stdp_case_gen*	Control STDP cases	Fig. 5.5
	incdec	Control update direction	Fig. 5.6
	stabilize_func	Stabilize weights bimodally	Fig. 5.7
Utility	*spike_gen*	Perform spike encoding	Fig. 5.8
	pulse2edge	Convert from pulse to edge	Fig. 5.9
	edge2pulse	Convert from edge to pulse	Fig. 5.10

function generation and three for STDP local learning). These synapses then feed into corresponding neuron bodies which perform response function summation through adder trees. Another macro enables the key comparison operation in WTA and the remaining three macros serve more generic utility purposes (e.g. spike encoding).

These macros (elaborated in Section 5.2) are summarized in Table 5.1 along with their functional descriptions and schematic figure labels. It should be noted, these custom macros can be generalized beyond use in the microarchitecture model in [79], and can serve as the foundation for building generic temporal functions based on *space-time* algebra [14].

5.1 Methodology

The proposed TNN7 custom cells are developed as hard macros using an open-source 7nm predictive Process Design Kit (PDK), called ASAP7 [104]. This section describes the ASAP7 library and the CAD design flow used in this work.

5.1.1 Framework

ASAP7 [104] is an academically certified, foundry agnostic, predictive PDK based on 7nm finFET technology. This involves a standard cell library and a collection of rule-sets for physical verification - design rule checks, layout vs. schematic, and parasitic extraction. The electrical activity of the transistor models is scaled from the BSIM-CMG SPICE models [105], which captures advanced trends in the finFET industry. ASAP7 offers transistor device models at four threshold voltages (SLVT, LVT, RVT and SRAM), and three process corners, typical-typical (TT), slow-slow (SS) and fast-fast (FF).

In this work, following selections are used for the design of custom macros: 1) RVT device models with nominal operating conditions at TT corner (0.7V supply voltage and 25°C operating temperature), 2) composite current source (CCS) modeling for timing files, and 3) Cadence/Mentor Graphics toolchain for logic synthesis, schematic, layout and characterization.

5.1.2 Methodology

In developing the custom macros, Cadence tool suite is used as follows: 1) *Genus* for register-transfer level (RTL) logic synthesis, 2) *Virtuoso* for schematics and layouts, 3) *Liberate* for characterization of the macros and generating Liberty (.lib) timing files, and 4) *Abstract* for generating Liberty Exchange Format (.lef) files of the macros. Layout verification, including Layout Versus Schematic (LVS) and Design Rule Check (DRC), is performed using Mentor Calibre and the resulting LVS & DRC-clean Graphic Data Stream (GDS) files are imported to *Abstract*. Moreover, Calibre Parasitic Extraction (PEX) tool reads the layout and generates the extracted netlist which is then used for Spectre simulations in *Liberate*.

In order to report the optimization gains presented in Section 5.3, following steps are adopted: 1) *Genus* is used to synthesize the original functional modules from [79] with the ASAP7 standard

cell library and establish the baseline values; 2) TNN7 macro equivalent of the original modules are designed by either (i) structurally optimizing at the microarchitectural level, or (ii) creating mixed-signal circuits from scratch in *Virtuoso*; 3) *Genus* is used to resynthesize the modules by replacing the ASAP7 standard cells with the TNN7 .lib and .lef files (obtained from *Liberate* and *Abstract*), to obtain post-synthesis area, power and delay. These values are then compared against the ASAP7-based post-synthesis values to compute the corresponding improvements.

5.2 TNN7 Custom Macro Cells

This section describes the circuit-level design of the proposed nine macros and their functionalities in detail. The macros are segregated into *TNN functionality cells*, that perform exclusive TNN functions, and *utility cells*, that perform generic functions like spike encoding.

5.2.1 TNN Functionality Cells

This subsection describes the six macros implementing synaptic response, WTA and STDP. The following notations are used for the two hardware clocks introduced in Chapter 3 - *aclk* for the unit clock and *gclk* for the gamma clock.

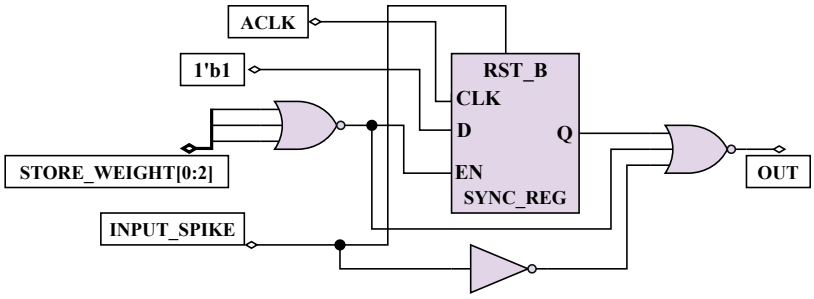


Figure 5.2: *syn_readout* macro

syn_readout and syn_weight_update

As noted in [79], synapses constitute majority of the hardware complexity in TNNs. Hence, in this work, main synaptic functions are identified, optimized and modularized into custom macros. The two key synaptic functions of response function generation and weight update

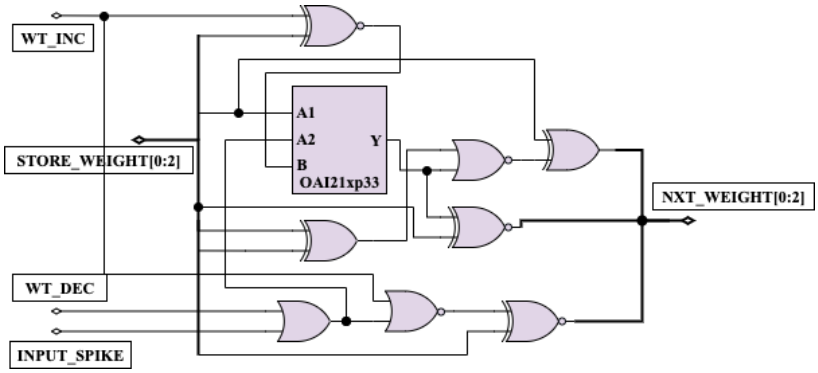


Figure 5.3: *syn_weight_update* macro

are implemented as *syn_readout* (Fig. 5.2) and *syn_weight_update* (Fig. 5.3) macros respectively. When an input spike pulse arrives, the synaptic weight undergoes a unit decrement every cycle, until it wraps around to the original value. During this process, the *syn_readout* macro takes in the weight value every cycle and asserts the output until the weight reaches zero, and then deasserts it. This parallels the unary-coded ramp-no-leak (RNL) response function in [79]. The *syn_weight_update* macro controls the weight decrementing process during readout, and updates the synaptic weight during "learning", via the STDP-based control signals (*WT_INC* and *WT_DEC*). Only one of the control signals is active at a time and performs either unit increment or decrement. Note that the *syn_weight_update* macro merely updates the weight based on external control signals; the control signals are generated by input spike (inference) and the three STDP macros (learning).

This modular approach to designing synapses provides flexibility to implementing TNN frameworks. For example, the response function can be changed by modifying *syn_readout* while keeping the other macros intact. This flexibility adds to the diversification of TNN models for diverse applications.

less_equal

The *less_equal* macro (Fig. 5.4) models the temporal inhibitor and functions as the basic unit for WTA inhibition. More generally, it implements the temporal operation of "less_equal" from space-time algebra [14] and hence is widely used in the TNN design framework. The input data (DATA_IN) value is propagated to the output if and only if it arrives earlier or at the same

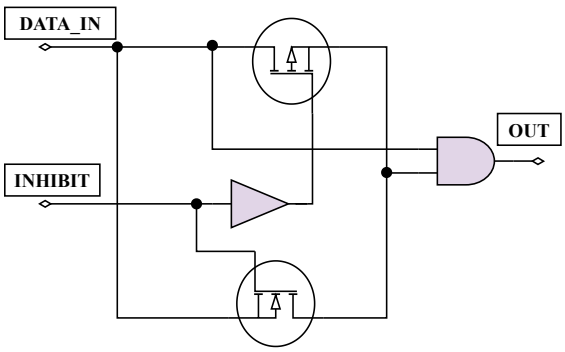


Figure 5.4: *less_equal* macro

time as INHIBIT; else, it is suppressed. This module's functionality can be achieved by using a single transistor [106]. However, to mitigate the high leakage current observed during the cell's characterization, a pair of NMOS and PMOS transistors is employed.

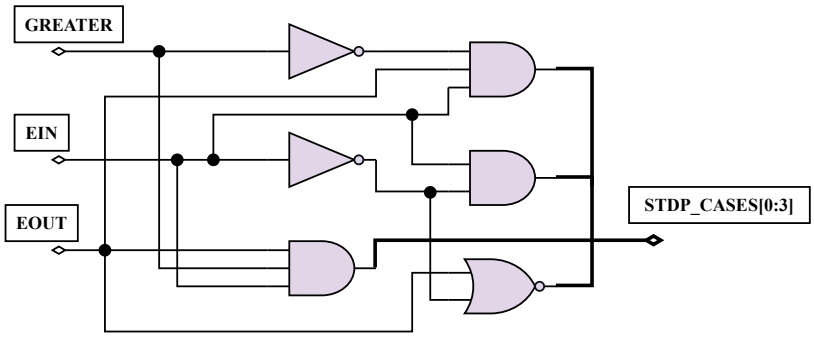


Figure 5.5: *stdp_case_gen* macro

stdp_case_gen

The *stdp_case_gen* macro generates the essential control signal outputs, corresponding to the four STDP cases from Table I in [79]. As shown in Fig. 5.5, it takes in the negated output of *less_equal* (GREATER) and input/output spikes represented as edge transitions (EIN/EOUT), and generates a one-hot encoded output for the STDP cases. When both input and output spikes are absent, the output is zero, resulting in no weight update during STDP.

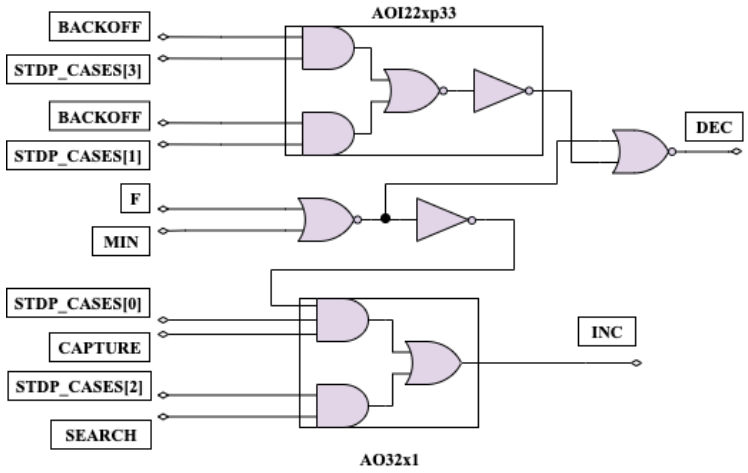


Figure 5.6: *incdec* macro

incdec

The *incdec* (Fig. 5.6) macro takes in the STDP cases and Bernoulli random variables (BRVs) as inputs (as in [79]), and generates control signals for driving the local synaptic weight update process. It consists of AND-OR-INVERT (AOI) cells that activates INC for STDP cases 0 and 2, and activates DEC for cases 1 and 3, if the BRV is one. It is important to note that the modularity in STDP logic (due to *stdp_case_gen* and *incdec*) allows for easy modification of STDP rules.

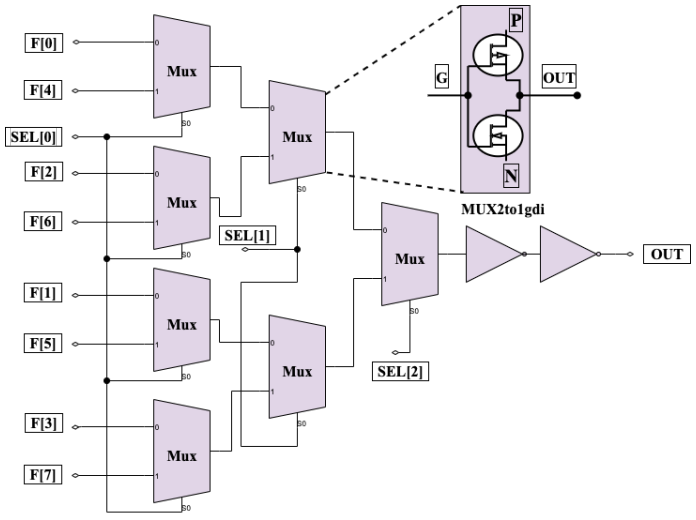


Figure 5.7: *stabilize_func* macro

stabilize_func

This macro (Fig. 5.7) is responsible for selecting the appropriate BRVs as per the stabilization function in [79], and plays a key role in establishing weight convergence. It is architected as an 8:1 multiplexer module with a hierarchy of Gate Diffusion Input (GDI) cells [107], each acting as a 2:1 multiplexer. The 2:1 GDI multiplexers utilize just two transistors, however suffer from degraded output levels. This is corrected by applying level restorers at the output, making the final design both robust and highly efficient.

5.2.2 Utility Cells

The remaining three macros are utility cells generalized to perform broader functions within the TNN framework such as spike encoding, synchronization, etc. They are detailed below.

spike_gen

The *spike_gen* macro (Fig. 5.8) plays a key role in spike encoding. It implements the combinational logic associated with a 3-bit counter used to convert input pulses of any width to an 8 cycle-wide output pulse (for 3-bit synaptic weights). As demonstrated in [79], this spike encoding is central to the ramp-no-leak (RNL) functionality of the compact synapse design used in the TNN framework, and can be easily extended to generalize for arbitrary pulse widths.

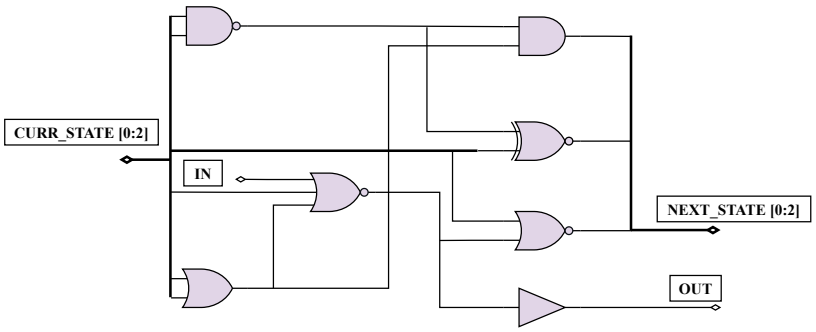


Figure 5.8: *spike_gen* macro

pulse2edge and edge2pulse

TNN implementations utilize edge-encoded signals (i.e., encoded as edge transitions from 0→1) for performing various temporal operations. The *pulse2edge* macro (Fig. 5.9) transforms an incoming pulse signal into an edge signal (lasting until the end of current *gclk* cycle), and is used extensively across the TNN framework. On the contrary, *edge2pulse* macro (Fig. 5.10) outputs a pulse lasting one *aclk* cycle as soon as an edge signal arrives at its input. It is typically used to produce internal reset pulses from *gclk* to synchronize the sequential blocks in the datapath.

All nine macros have been carefully designed to use minimal number of gates and transistors to achieve their corresponding functionalities. In order to further reduce cell area, we perform diffusion layer overlapping during manual layout. Table 5.2 reports their respective PPA metrics. In order to demonstrate their benefits, these macros are used to build various TNN prototypes as discussed in the next section.

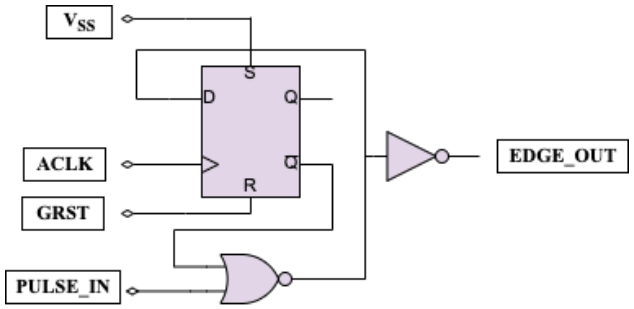


Figure 5.9: *pulse2edge* macro

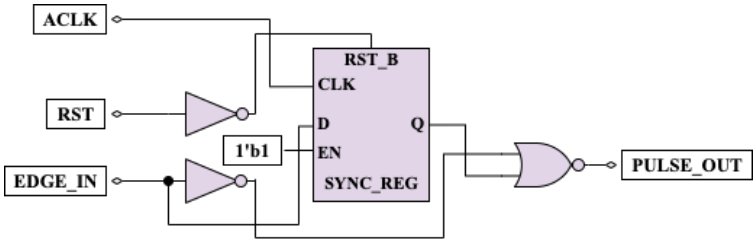


Figure 5.10: *edge2pulse* macro

Table 5.2: 7nm PPA for proposed custom macros

Custom Macro Name	Leakage Power (nW)	Delay (ps)	Cell Area (μm^2)
syn_readout	0.43	32	0.50
syn_weight_update	1.22	190	1.24
less_equal	0.17	30	0.17
stdp_case_gen	0.34	66	0.60
incdec	0.26	56	0.34
stabilize_func	0.12	158	0.36
spike_gen	1.46	28	1.55
pulse2edge	0.44	22	0.44
edge2pulse	0.49	58	0.61

5.3 Results and Discussion

This section presents 7nm post-synthesis power, performance, area (PPA) results for application-specific TNN prototypes. Performance is measured in terms of computation time (time taken to process one input), and is derived from the critical path delay and the gamma period as in [79]. Area is the total cell and net area, while power includes dynamic (calculated using Cadence *Joules*) as well as leakage power.

In order to demonstrate the efficacy of the TNN7 macros, we perform benchmarking for two groups of TNN prototype designs targeting two application domains: 1) 36 single-column TNN designs for unsupervised time-series clustering on 36 UCR datasets from [80], with total synapse counts ranging from 130 to 6750; and 2) three much larger multi-layer TNN designs for MNIST digit recognition, namely, *2-layer*, *3-layer* and *4-layer* TNNs (from [15]) with total synapse counts of 389K, 1,310K and 3,096K, respectively. Following [79], an operating frequency of 100 kHz is chosen for *aclk* based on real-time operation requirement. We observe linear scaling of dynamic power with frequency and omit those results here for brevity.

5.3.1 UCR Time-Series Clustering

As shown in [80], TNN designs outperform or are competitive to state-of-the-art algorithms for unsupervised time-series clustering, averaging across the 36 UCR benchmark datasets. A specific column configuration is used for each of the 36 UCR datasets depending on the corresponding input size and number of clusters. While the hardware complexity analysis in [80] uses standard

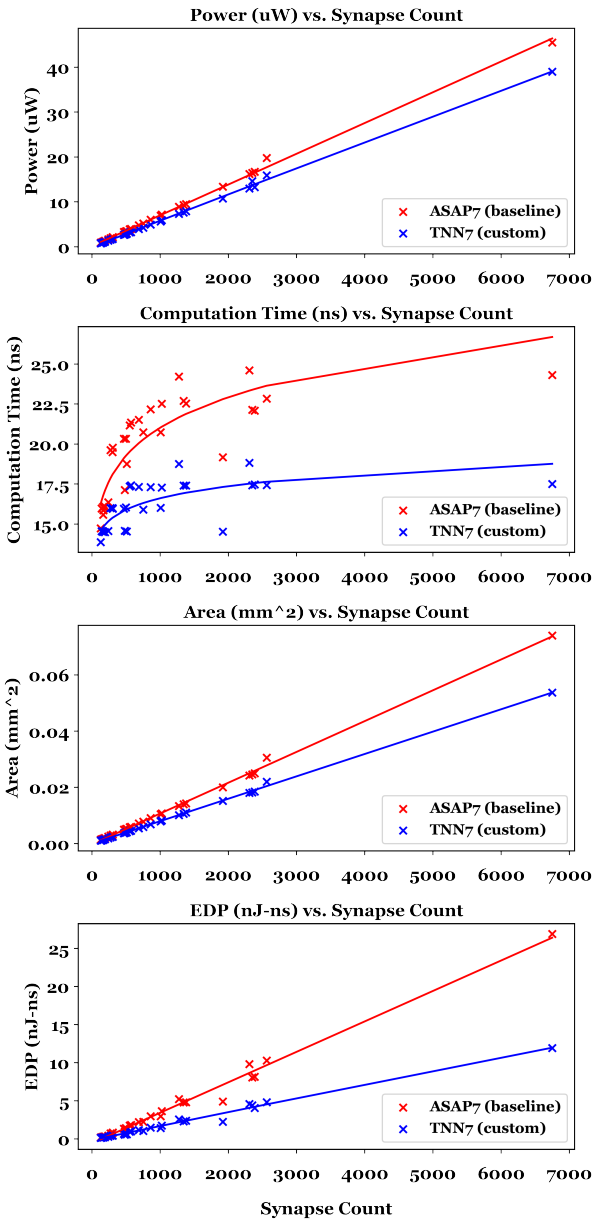


Figure 5.11: ASAP7 vs. TNN7 7nm PPA scaling across synapse counts for the 36 single column TNN designs as used in [80]

technology scaling to estimate the 7nm results from 45nm post-synthesis results, we present direct post-synthesis 7nm PPA results for all 36 TNN designs and further optimize them with our custom macros.

To assess the range of PPA complexities for time-series clustering TNNs, we plot area, power, computation time, and energy-delay product (EDP), for the 36 single-column designs in Fig.

5.11. EDP is used here to gauge both energy-efficiency and performance. Three key results can be observed here:

1. *PPA synaptic scaling*: Area and power scale linearly with total synapse counts for both ASAP7 baseline and TNN7 custom designs, whereas computation time scales logarithmically with synapses per neuron (p). This corroborates with the characteristic scaling equations in [79]. Note that x-axis is monotonic in $p*q$ (not p), making computation time data points non-monotonic in Fig. 5.11.

2. *PPA improvements with TNN7*: TNN7 designs consume about 18% less power and 25% less area compared to baseline designs, and are about 18% faster. EDP improves by more than 45%, which clearly shows TNN7 designs are significantly more energy-efficient and are also faster. The gap between the two designs grows with increasing synapse count, which implies, as TNN designs grow larger, they reap even more benefits from custom macros.

3. *Potential for low-power edge-native sensory processors*: With custom macros, even the *largest* TNN column with 6,750 synapses consumes just 0.054 mm^2 area and 39 μW power. Note that this also accounts for on-chip learning via STDP, highlighting the value of proposed macros for highly energy-efficient TNN sensory processing units capable of online continuous learning.

5.3.2 MNIST Digit Recognition

Here, we move to much larger TNN designs and evaluate three multi-layer TNN prototypes for MNIST digit recognition, with different design points in the error rate vs. hardware complexity tradeoffs. The three designs are as follows: 1) 2-layer TNN (389K synapses and 7% error) derived from ECVT in [15]; 2) 3-layer TNN (1.31M synapses and 3% error) derived from ECCVT in [15]; and 3) 4-layer TNN (3.096M synapses and 1% error) derived from ECCCVT in [15]. Table 5.3 provides 7nm PPA for these designs, derived using synaptic count scaling as in [79]. Note that "C" layers above consist of TNN7 columns, however the "VT" layers [15], that are a simpler form of TNN columns, are currently not supported within TNN7. Hence, the synaptic scaling here treats all network layers as "C", thereby providing an upper limit on the PPA complexity.

Table 5.3: ASAP7 vs. TNN7 7nm PPA comparison for three TNN prototype designs for MNIST from [15]

TNN Design	Synapse Count	Error Rate	Cell Library	Power (mW)	Comp. Time (ns)	Area (mm^2)
2-Layer	389K	7%	ASAP7	2.62	49.00	4.27
			TNN7	2.25	41.38	3.09
3-Layer	1,310K	3%	ASAP7	8.83	78.37	14.37
			TNN7	7.57	66.16	10.42
4-Layer	3,096K	1%	ASAP7	20.86	108.46	33.95
			TNN7	17.89	91.58	24.63

From Table 5.3, similar PPA improvements with custom macros can be observed for these complex multi-layer TNNs (14%, 16%, and 28% improvements on power, performance and area, respectively). The 4-layer TNN with 3M synaptic weights and 99% MNIST accuracy consumes only 17.89 mW power and 24.63 mm^2 area. This TNN represents an edge-native real-time sensory processing unit that is capable of both online (MNIST-like) image-based classification and continuous learning, while consuming less than 20 mW power.

Using the survey of MNIST neural networks from [108], it can be observed that for similar accuracies, TNN-based processing units that consume a few tens of mW power are about 1000x more efficient comparing to GPUs, 100x comparing to FPGAs and 10x comparing to many state-of-the-art ASICs that consume a few hundreds of mW power. Furthermore, TNN7 enhances this scalability as it offers a lower-cost trajectory in the accuracy vs. hardware complexity tradeoff.

5.3.3 Synthesis Runtime Evaluation

A further advantage to using a custom cell library is significantly faster design netlist generation. As the macro design instances are preserved and not manipulated during synthesis, it enables the synthesis tool to realize a design hierarchy by directly mapping the hard macros, thereby mitigating the combinatorial search space complexity for the optimization tool. To evaluate this benefit for TNN7, we use the following setup: Genus v19.1 is run on a server comprising of 48 Intel(R) Xeon(R) E5-2680 CPU cores with the maximum number of CPUs utilized set to 8. Synthesis was performed on the same column configurations from Section 5.3.1, with the TNN7 custom macros as well as without them (ASAP7 baseline).

Fig. 5.12 depicts the runtimes for both standard ASAP7-based and corresponding TNN7-

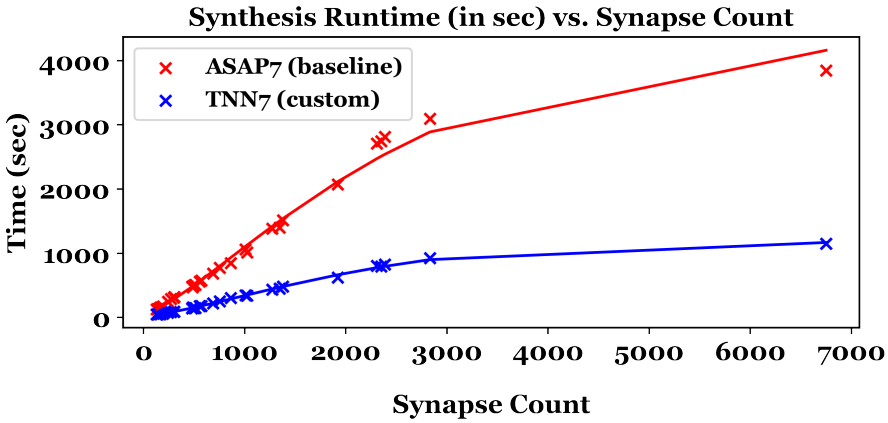


Figure 5.12: ASAP7 vs. TNN7 synthesis runtime comparison

based designs.

On average, TNN7 speeds up the netlist generation (including mapping and optimization) by 3.17x with respect to the baseline ASAP7-based designs. Using TNN7, the largest column with 6750 synapses is synthesized in 926 seconds (∼15 minutes), as opposed to 3849 seconds (∼1 hour) for the baseline design. Fig. 5.12 illustrates increasing runtime benefits for TNN7 as the designs grow larger. This trend can be extrapolated beyond single columns to multi-layered networks based on synapse counts, demonstrating the scalability of TNN7 to realize deep TNNs, that would have otherwise suffered from long runtimes.

Chapter 6

SRAM-Based Synaptic Array Design

Chapters 3 and 5 proposed a microarchitecture framework [79] and a set of custom macro building blocks [81] for efficient hardware implementation of TNNs respectively. However, these works rely on expensive flip-flops to implement the synaptic crossbar. This chapter is a first attempt at implementing the entire synaptic crossbar array in SRAM, to significantly reduce the hardware complexity. This serves as the first step towards compute-in-memory (CIM) implementation of TNNs, hence named *TNN-CIM*. SRAM is chosen in this work due to its compatibility with standard CMOS logic and maturity [109, 110] as compared to other emerging approaches such as Resistive RAM [111].

Our SRAM design follows the very recent work titled FAST [113], which proposed a 10T SRAM cell with two intra-cell NMOS switches and one inter-cell transmission gate switch. In contrast to prior SRAM-based CIM works [114–119] that are limited by serial row-by-row access, FAST enables concurrent row compute through multi-phased shifting. Here, we re-purpose the FAST SRAM cell to store sequential states across switches and use two-phased switching to ripple states through a series of cells, thus implementing a finite state machine (FSM). We then leverage this FSM to perform *in-situ* synaptic inference and STDP learning rules. Key contributions are:

- We present *TNN-CIM*, the first work towards compute-in-memory implementation of TNNs, wherein the synaptic crossbar with STDP is fully implemented in SRAM.
- As part of synaptic inference, TNN-CIM proposes a novel *Ripple-Flip Counter* to implement binary counting within SRAM, with arbitrary powers-of-2 count values. We believe such

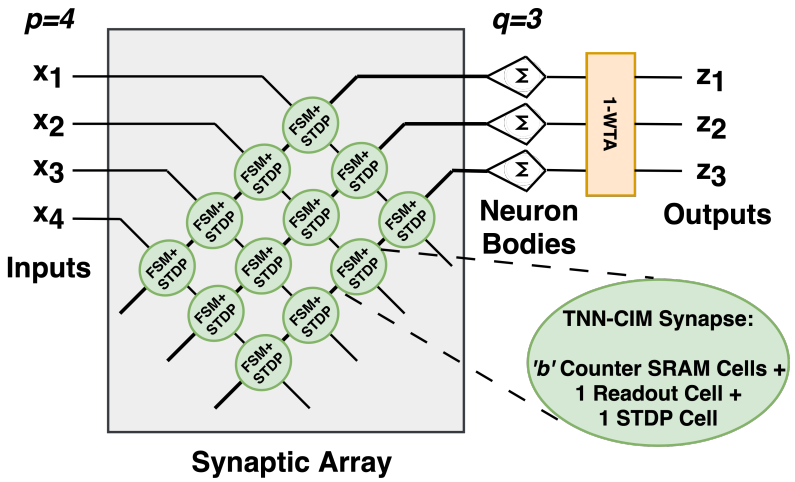


Figure 6.1: A 4x3 TNN column containing: 4 synaptic inputs per neuron and 3 neurons with its 4x3 synaptic crossbar, and 1-WTA lateral inhibition [*adapted from [112]*]. Each of the 12 synapses in the crossbar stores a b-bit weight value, performs local inference via counter-based FSM, and updates its weight locally via STDP learning, concurrently every compute cycle.

sequential counting implemented in SRAM without any adder is novel.

- Along with synaptic inference compute, TNN-CIM implements *in-situ* STDP-based unsupervised learning with which all the synapses in the crossbar can be simultaneously and locally updated in every compute cycle.
- Compared to previous TNN implementations [79, 81], TNN-CIM provides improvements of 1.3x, 1.4x and 1.7x, in power, performance and area, respectively.
- Parameterized equations to assess transistor count complexity scaling of TNN-CIM synaptic arrays are provided.

Section 6.1 describes circuit-level design of the key components of the proposed SRAM-based TNN-CIM, followed by hardware complexity analysis in Section 6.2.

6.1 TNN-CIM: SRAM Synapse Implementation

This section presents the key components of TNN-CIM, starting with the Ripple-Flip Counter design, followed by implementation of synaptic inference utilizing this counter and finally synaptic

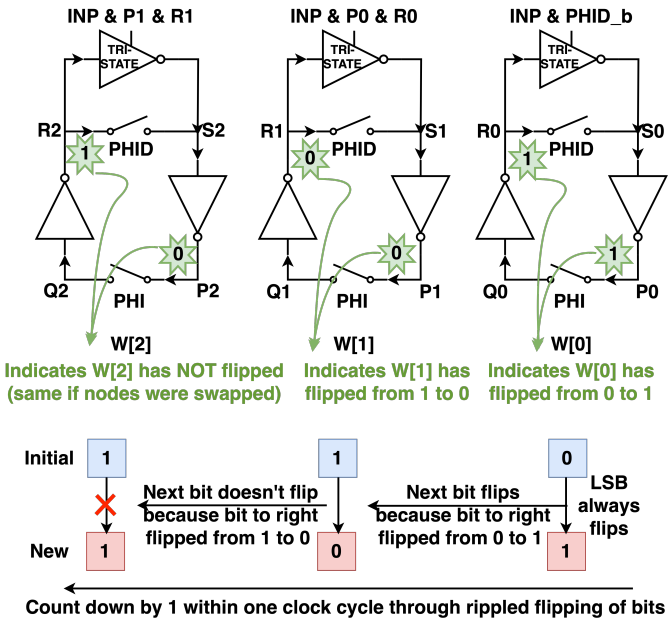


Figure 6.2: Ripple-Flip Counter: 3-bit weight is shown where each bit is implemented in an SRAM cell consisting of additional two transmission gate switches controlled by PHI/PHID signals and a tri-state inverter that flips the cell value based on the "flip" status of the cell to the right. Example shows weight counting down from 6 to 5 through rippled flipping of bits.

learning algorithm (STDP) implementation. Minor circuit details are omitted from schematics for brevity.

6.1.1 Ripple-Flip Counter FSM

Prior SRAM CIM works have focused exclusively on implementing combinational logic such as search, shift, add, and multiply. Here, we propose an SRAM design that leverages FAST [113] switches to store different states and implement sequential counting (Figure 6.2). FAST SRAM uses a peripheral 1-bit adder to perform addition in N clock cycles (where N is the bitwidth of the stored value). While this can be used to implement counting, it takes multiple cycles to update its count by 1. In contrast, the key mechanism in TNN-CIM is to perform upcount (or downcount) by rippling bit flips from LSB to MSB *within a single clock cycle*. For example, downcount by 1 simply flips all '0's to '1's from LSB to MSB until a '1' is reached at which point that '1' is flipped to '0' and no further rippling occurs (see Figure 6.2). Similarly, upcount by 1 entails flipping all '1's to '0's from LSB to MSB until a '0' is reached. Note that this mechanism easily enables upcount/downcount by arbitrary powers-of-2 by just changing the starting bit for

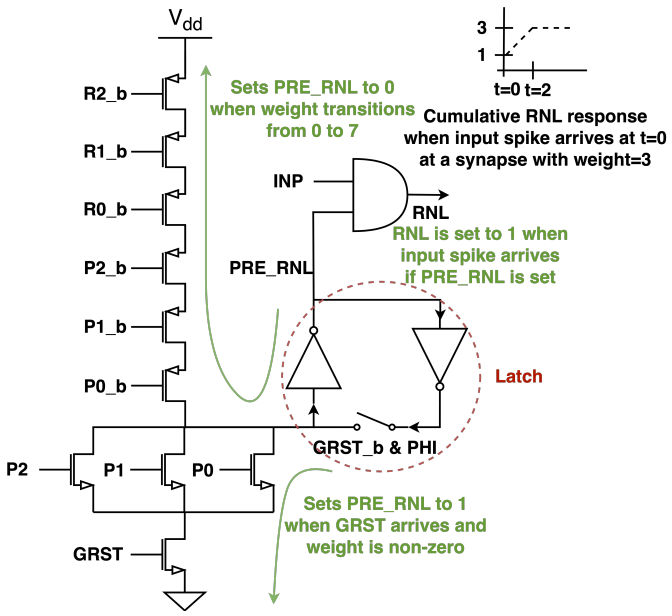


Figure 6.3: RNL Readout Cell: Pull-down circuit sets *PRE_RNL* to 1 at the beginning of gamma period for non-zero synaptic weight. Pull-up network resets *PRE_RNL* to 0 when weight transitions from 0 to 7 by leveraging the decoupled states across all 3 counter cells. *PRE_RNL* is latched using a custom SRAM-type cell with one switch. Final *RNL* output is set to 1 if *PRE_RNL* is high and input spike is asserted.

rippling (count by 2 starts from the second bit, i.e., bit to the left of LSB and so on). To implement this in SRAM, a bit has to know whether its neighbor to the right has been flipped or not. This can be done by carefully re-purposing the decoupled states across the switches within each cell, as explained next.

Figure 6.2 shows a 3-bit weight counter where each counter cell incorporates two transmission gate switches and a tri-state inverter in addition to the traditional 6T configuration. In contrast to FAST that uses three phases, we use only two phases wherein the transmission gate switches are controlled by *PHI* and delayed *PHID* (ϕ_2 and ϕ_{2d} as in FAST), which are asserted in the first half of clock cycle and de-asserted in the second half. The switches are opened before performing the operation ("shift" in case of FAST; "downcount" in our case), and once finished, *PHI* switch closes first followed by the *PHID* switch. To downcount by 1, the LSB cell W[0] is flipped via the tri-state inverter when an input spike *INP* arrives and *PHID* switch is open (*PHID_b* is set). This triggers a chain reaction that ripples to the left, all within the same clock cycle.

Now, W[1] only needs to flip if W[0] flipped from 0 to 1. Note, if the bit to the right didn't

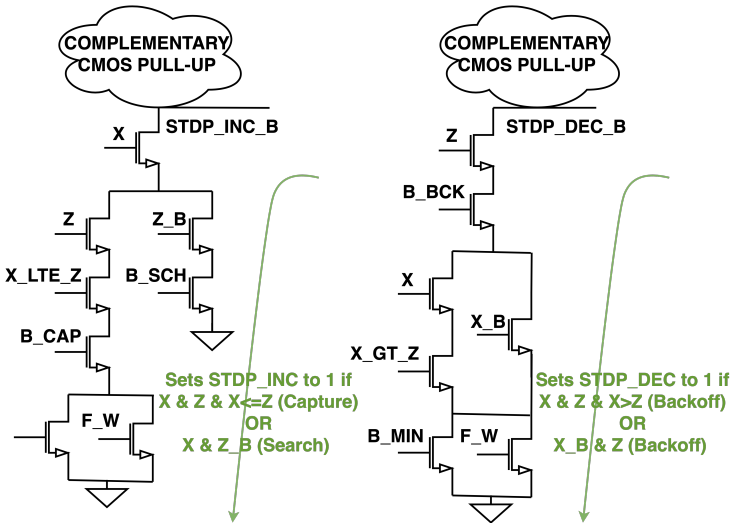


Figure 6.4: STDP: Custom CMOS gates to increment/decrement as per STDP cases 1 (capture), 2 (backoff), 3 (search), 4 (backoff) in Table 6.1. *B_CAP*, *B_SCH*, *B_BCK*, *B_MIN* are Bernoulli random variables to regulate corresponding stochastic updates. *F_W* is stabilization function [79]. X/Z are input/output spikes.

flip or flipped from 1 to 0, it implies the end of rippling. This "flip" status is indicated by both *P0* and *R0* nodes of W[0] simultaneously having a value of 1. Also, this is possible only if the switches are open. This is depicted in the control signals of W[1]'s tri-state inverter. A similar mechanism applies for all the bits to the left. Note that this counter naturally wraps around to 7 from 0, as is needed for RNL response function generation.

6.1.2 Synaptic Inference (Response Function Readout)

As proposed in [79], synaptic inference generates a unary readout of ramp-no-leak (RNL) response function based on the weight counter state (i.e., the weight value). RNL output is set to 1 when an input spike arrives at a synapse with non-zero weight (at which point weight counter also starts decrementing), and it then gets reset to 0 when the weight counter wraps around to 7 from 0 (for 3-bit weight). This sets the RNL output to 1 for as many cycles as the synaptic weight value. We implement this as a "readout" cell as follows.

For every 3-bit synaptic weight consisting of 3 counter cells (Figure 6.2), there exists one readout cell (Figure 6.3). As in [79], a *gamma* period denotes the duration of one synaptic learning plus inference compute. *GRST* is a signal that is asserted only for one cycle at the beginning of

Case	Input Conditions		Weight Update
1. Capture	$X \neq \infty$;	$X \leq Z$	$\Delta w = +B\_CAP * max(F\_W, B\_MIN)$
2. Backoff	$Z \neq \infty$	$X > Z$	$\Delta w = -B\_BCK * max(F\_W, B\_MIN)$
3. Search	$X \neq \infty$; $Z = \infty$		$\Delta w = +B\_SCH$
4. Backoff	$X = \infty$; $Z \neq \infty$		$\Delta w = -B\_BCK * max(F\_W, B\_MIN)$
-	$X = \infty$; $Z = \infty$		$\Delta w = 0$

Table 6.1: STDP update rules (from [79]) directly implemented in TNN-CIM, consisting of four cases depending on the presence/absence/relative timings of input (X)/output (Z) spikes.

every gamma period. As shown in Figure 6.3, it consists of a custom tri-state circuit that sets or resets *PRE_RNL* which is latched in a set of two cross-coupled inverters with a transmission gate switch. *PRE_RNL* is set to 1 at the beginning of gamma period only if the weight value is non-zero (indicated by *P2*, *P1* and *P0*), else it remains at zero. In the presence of an input spike, *RNL* output is set to 1 if *PRE_RNL* is asserted. *PRE_RNL* and thereby *RNL* are both de-asserted when weight counter transitions from 0 to 7 which is implemented by the pull-up circuit, leveraging the decoupled states across the switches within the 3 counter cells. Once *PRE_RNL* is reset to 0, it can only be set in the next gamma period. This ensures RNL output is asserted only until weight counter counts down to 0 in presence of an input spike.

6.1.3 Synaptic Learning (STDP)

The STDP unsupervised learning logic (Figure 6.4) generates two control signals, *STDP_INC* and *STDP_DEC*, to increment or decrement the weight counter by 1. TNN-CIM directly implements the four STDP cases listed in Table 6.1 as noted by the green text in Figure 6.4 (fifth case is implicit). Compared to previous TNN STDP implementations, our STDP implementation incorporates a key optimization. Each counter cell in Figure 6.2 that natively performs decrement needs additional transistors to support increment due to STDP. This overhead is avoided by ensuring the counter wraps around to its (original value + 1) during RNL readout (i.e., downcount for only 7 instead of 8 cycles for 3-bit weight). As a result, during the STDP cycle, high *STDP_INC* implies no change, high *STDP_DEC* implies decrement by 2, and neither set high implies decrement by 1 (this restores original weight from previous gamma period before current gamma's RNL compute begins). Thus, we avoid having to perform any explicit increment. Note that both STDP_INC and STDP_DEC cannot be set high simultaneously. In addition to the circuit shown

Table 6.2: TNN-CIM Transistor Count (TC) evaluation of a single synapse with bitwidth b based on its key components.

Component	b-bit Synapse	pxq Synaptic Array
Ripple-Flip Counter	$18b + 6$	$(18b + 6) * pq$
RNL Readout	$3b + 13$	$(3b + 13) * pq$
STDP	$2^{b+2} + 2b + 76$	$(2^{b+2} + 2b + 76) * pq$
Total	$2^{b+2} + 23b + 95$	$(2^{b+2} + 23b + 95) * pq$

in Figure 6.4, we also use custom latches (highlighted in red in Figure 6.3) to generate signals such as X_LTE_Z ($X \leq Z$) and X_GT_Z ($X > Z$). Thus, this STDP logic implementation localized for every set of 3 counter cells (representing 3-bit weight) enables TNN-CIM to perform *in-situ online continuous learning within the SRAM array*.

6.2 Results and Discussion

This section evaluates the proposed TNN-CIM SRAM-based synaptic array against the baseline flipflop-based microarchitecture in [79]. Two types of evaluation are presented: 1) transistor count analysis and parameterized equations to assess hardware complexity scaling of TNN-CIM, and 2) 45 nm CMOS power-performance-area (PPA) for three synaptic array sizes - two sizes 64x8, 128x10 are adopted from [79] and 96x2 from [78] targeting unsupervised ECG clustering.

6.2.1 Transistor Count Evaluation

We derive transistor count equations based on the bitwidth b of each synapse, and the synaptic array size pxq with total synapse count $p * q$ (Figure 6.1). Table 6.2 provides the parameterized equations for a single synapse and a pxq synaptic array, segregated into the three key components. These equations can be used to assess transistor count, and thereby area and leakage power, for arbitrarily configured TNN-CIM array.

Figures 6.5 illustrates the breakdown of transistor count for a single synapse across varying bitwidths (1 to 8 bits). Note that prior TNN works only present hardware complexity for 3-bit synapses. Each 3-bit synapse in TNN-CIM consumes 196 transistors (less than half of the transistor count in [79]). It can be seen from Figure 6.5 that STDP scales exponentially with bitwidth and experiences a sharp increase for 7 and 8 bits, thereby suggesting a reasonable

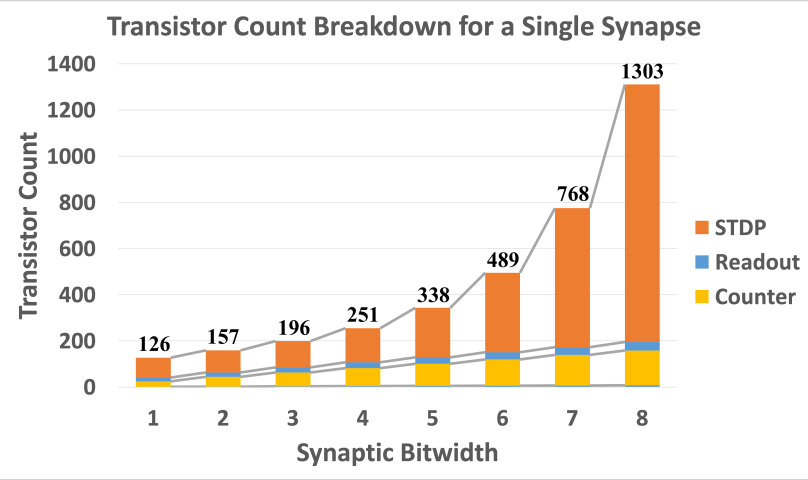


Figure 6.5: Transistor Count Breakdown for a Synapse: STDP (60%), counter (30%), and readout (10%). Total transistor count is shown at the top of the bar for each bitwidth.

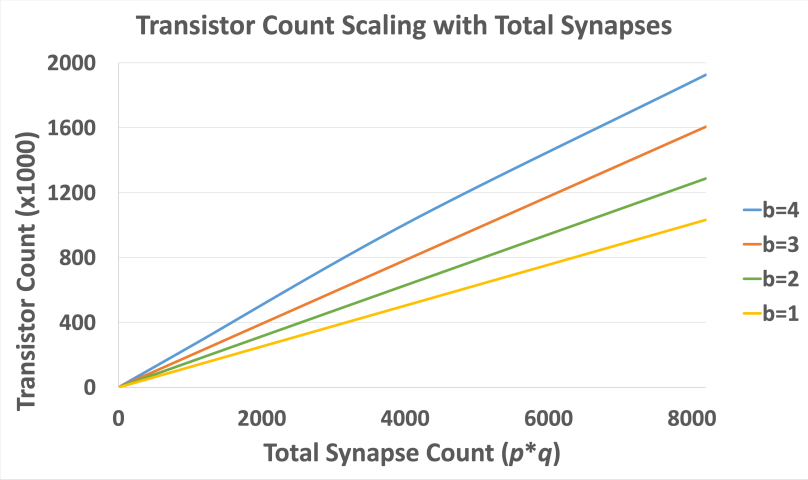


Figure 6.6: Transistor Count scaling relative to total synapse count (pxq) and bitwidth (b) of synaptic weights.

maximum bitwidth of 6. In contrast, counter and readout scale much better (linear) throughout 1 to 8 bits. Figure 6.6 illustrates the linear scaling of transistor count with increasing total synapse count across 1 to 4 bits (from neuroscience, only 3-4 bits are needed).

Key Takeaway: STDP consumes majority (60%) of synaptic complexity, followed by counter (30%) and readout (10%). STDP's overhead (esp., dynamic power) can be substantially mitigated once weights converge (due to infrequent updates).

Table 6.3: 45nm PPA Comparison of TNN-CIM vs. Baseline [79] for three synaptic array sizes (b=3 bits), including an application-specific configuration for ECG signal clustering.

	Synapses x Neurons	**Total Synapses**	**Area** [mm^2]	**Comp. Time** [ns]	**Power** [mW]	**Energy** [nJ]
TNN-CIM	64 × 8	512	0.026	22.5	0.177	40.71
	128 × 10	1280	0.066	22.5	0.443	101.89
	96 × 2 (ECG)	192	0.010	22.5	0.066	15.18
Baseline[79]	64 × 8	512	0.045	28.95	0.225	54.00
	128 × 10	1280	0.117	32.40	0.558	133.92
	96 × 2 (ECG)	192	0.017	30.6	0.084	20.16

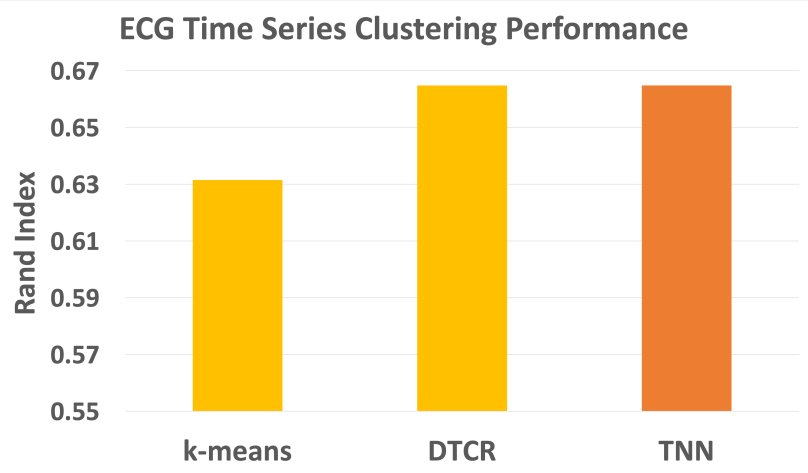


Figure 6.7: ECG Clustering Performance (rand index) of TNN with 96x2 synaptic array vs. k-means and state-of-the-art DTCR.

6.2.2 PPA Evaluation and ECG Signal Clustering Performance

Table 6.3 provides 45nm PPA for three TNN-CIM synaptic arrays (b=3 bits) and compares them against corresponding flipflop-based baselines in [79]. Baseline area, power values are scaled to reflect just the synaptic complexity alone. Computation time is not scaled as neuron body incurs the critical path in baseline. Baseline 96x2 values are derived using characteristic scaling equations from [79]. TNN-CIM schematics and layouts are designed with 1V supply voltage and 100kHz clock using Cadence Virtuoso with SPICE simulations to get PPA values.

Table 6.3 shows that TNN-CIM reduces area and power by 1.7x and 1.3x respectively. In contrast to baseline implementations, TNN-CIM incurs majority of the critical path in synaptic rippling compute (body accumulation can be simply implemented as analog addition over shared bitline [120]), and hence it stays constant with fixed bitwidth. Computation time is improved by

1.4x. Area and power for TNN-CIM and baseline scale linearly with p^*q. Figure 6.7 shows PyTorch [121] results (clustering rand index) on ECG200 dataset [122] for a TNN column with $96x2$ synaptic array, illustrating its efficacy over baseline k-means and more complex DTCR [123].

Key Takeaway: TNN-CIM significantly improves all three PPA metrics compared to flipflop-based synaptic arrays. Compared to complex ML algorithms running on CPUs and GPUs consuming 10's-100'sW power, TNN-CIM incurs just 66 μW, enabling competitive ECG clustering within sub-mW power.

Chapter 7

CAM-Based Reference Frame Design

Cortical columns in the neocortex model sensory information and knowledge in structured *Reference Frames* (RFs) and continuously predict and update the stored models as the sensing agent moves in an environment. As mentioned earlier, such a Reference Frame, inspired from biological grid cells in the entorhinal cortex [63] and place cells in the hippocampus [64], implements the common universal algorithm for achieving intelligence within each cortical column, irrespective of the sensory modality feeding that particular cortical column.

Following this, Hawkins' colleagues from Numenta successfully demonstrated visual object recognition on MNIST using a grid cell-based network, called GridCellNet [65], that learns a digit by associating features at certain locations within an implicit RF and classifies digits by obtaining sequential input features through movement in this RF. Prior works [60, 62] have illustrated the important roles of RFs in cortical columns, and the potential applications of this new theory based on software simulation results. However, currently no prior work has investigated the feasibility and the potential of direct implementation of Reference Frames in conventional off-the-shelf digital CMOS technology.

As defined in Chapter 4, a Cortical Column is modeled as consisting of an Agent and a Reference Frame (RF), wherein the RF maintains a map of the sensory information collected by the agent and the agent then achieves goal-oriented behaviors using input sensory features along with information from the RF. In this model, intelligence arises as a result of "movement" in the brain, wherein the system learns to associate "features" (semi-unique identifiers for tangible

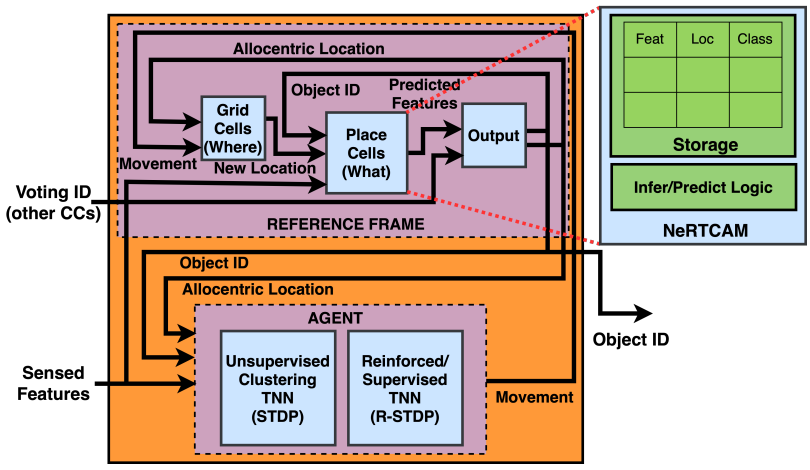


Figure 7.1: NeRTCAM as part of Reference Frame (RF) within a Cortical Column (*adapted from [124]*). It is used as a building block for RF's key component that holds "sensory map", i.e., the *Place Cells*. NeRTCAM implements storage as well as inference/prediction logic consistent with the models used in [65, 68].

structures on an object, or abstract concepts forming a whole idea in a cognitive map) with "locations" in an RF. This model of intelligence necessitates having a form of associative lookup for features given locations, and vice versa. Hence, in this chapter, we propose a content addressable memory (CAM) based microarchitecture as a step towards direct CMOS implementation of the RF. Specifically, we propose the Neuromorphic Reverse Ternary Content Addressable Memory (NeRTCAM) targeting inference using RFs.

This chapter is organized as follows. Description of NeRTCAM system architecture is provided in Section 7.1. Section 7.2 discusses specific implementation details for each of the NeRTCAM system components, followed by post-synthesis PPA evaluation and results in Section 7.3.

7.1 NeRTCAM Architecture

In this section, we present the proposed NeRTCAM (Neuromorphic Reverse Ternary Content Addressable Memory) architecture and describe its design and operation. First, a system overview is provided followed by description of input commands (i.e., the macro-operations queried to NeRTCAM by the agent). The system components and internal micro-operations will be discussed in detail later in Section 7.2.

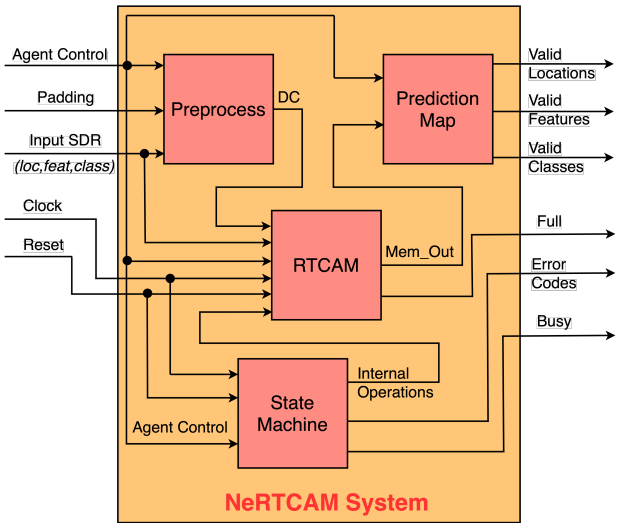


Figure 7.2: NeRTCAM System consists of four main components: 1) Preprocess, 2) RTCAM, 3) State Machine, and 4) Prediction Map. Preprocess and prediction map are combinational, whereas RTCAM and state machine are sequential. The system as a whole takes in input SDR and control commands from agent and outputs valid set of feature-location-class triplets.

7.1.1 System Overview

NeRTCAM (depicted in Figure 7.2) is designed to serve as a fundamental building block for implementing Reference Frame (specifically the most important "*Place Cells*" [124]), storing and retrieving information in the form of {feature, location, class} triplets. It is to be noted that it can also be used as a separate structure to which the "Place Cells" assign learned information for storage and fast retrieval during inference. NeRTCAM performs three main functions within the RF: 1) *Predict* feature and class given location of the sensor, 2) *Predict* sensor location and class given input feature, or 3) *Infer* class given location of the sensor and the sensed feature at that location. Each of these operations can output multiple matching entries. Further, NeRTCAM supports fuzzy matching as will be explained later in Section 7.2.1.

In order to achieve this, NeRTCAM is designed with four major components as shown in Figure 7.2, namely, *Preprocess*, *RTCAM*, *State Machine*, and *Prediction Map*. NeRTCAM takes in instructions from agent in the form of control commands (*Agent Control*) and fuzziness amount (*Padding*). It also receives an input SDR (sparse distributed representation as in [65]) that is used to query matching entries based on the content stored in NeRTCAM. At the output, it generates three different vectors indicating valid set of locations, features and classes consistent

with the information collected by the agent as it navigates an environment. Alongside, three status signals, namely, *full*, *error* and *busy* are also generated. High-level functionalities of the four components are described next.

The preprocess block is responsible for creating the "don't-care" (*DC*) mask pertaining to the operation the agent wishes to perform, as well as modifying it to reflect the desired fuzziness for matching. The output of this block (the DC mask) is fed to the RTCAM along with input SDR. The input SDR and DC mask are both vectors of same length with certain number of bits reserved for representing feature, location and class. Feature, location and class sections within the SDR are assumed to be 1-hot. RTCAM is the storage element possessing all the learned triplets by the agent and it outputs all matching valid outputs, i.e., class and feature/location predictions (*Mem_Out*) corresponding to the input SDR and DC mask. The state machine converts control commands from agent (macro-operations) to internal micro-operations for RTCAM that performs memory-native functions such as lookup. Lastly, the prediction map is used to condense the multiple matching 1-hot memory output into k-hot binary vectors that represent the current set of valid features, locations, or classes to be informed to the agent.

7.1.2 Agent Control Commands

NeRTCAM supports six control commands: CLEAR, RESET, STORE, DELETE, INFER, and PREDICT. These commands form a concise view of how an RF-based system can support an agent in the cortical columns computing paradigm. A flowchart describing the steps involved in the four operational commands (STORE, DELETE, INFER, PREDICT) is presented in Figure 7.3. The two simple commands (CLEAR, RESET) are omitted from the figure for brevity. All six commands are explained below.

CLEAR and RESET

CLEAR and RESET are simple housekeeping operations for NeRTCAM. The input SDR provided by agent is irrelevant for both these operations and hence is ignored by the system. When performing a CLEAR, the system clears all of its memory contents. This is typically used during initialization. It can also be used by the agent whenever it wishes to start from scratch with

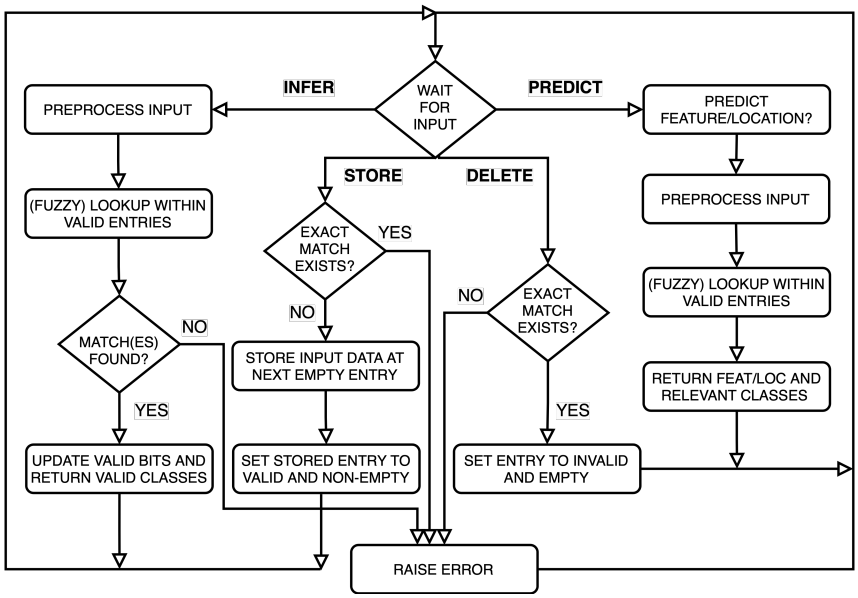


Figure 7.3: Agent Control Commands Flowchart: The process of executing four key commands from the agent (STORE, DELETE, INFER, PREDICT) is illustrated in terms of the different steps involved. STORE and DELETE are performed with fully specified input SDR with feature, location and class. INFER and PREDICT are performed with partially specified input SDR along with corresponding preprocessed don't-care (DC) mask.

an empty memory (e.g., migration to a completely new environment or task that's unrelated to previously learned information).

RESET is an operation that the agent can use to force the system to 'forget' the sequential information it collected while performing an identification on a certain object. Note that an object is identified by sequentially collecting sensor location-feature information and remembering the valid set of classes (object IDs) that are consistent with all location-feature pairs observed so far. Thus, RESET is intended to be used either after a successful identification as it prepares the system to aid in the agent's observation of a different object, or after a failed identification so that the agent can either try again or move on. It can also be used whenever the agent so pleases. Note that RESET does not reset or clear the stored entries (i.e., feature-location-class triplets).

STORE and DELETE

STORE and DELETE are slightly more complex operations compared to RESET and CLEAR. STORE is used to add new information the agent will use for identification, into the system. When performing a store, the agent must specify exactly what feature at what location belongs

to what class in the input SDR. This is done only when the agent is fully confident about its orientation within an environment. Relatively, during identification, agent's observations are more general (partial), and the system's primary goal is to match general observations with specific, remembered information. Moreover, to avoid redundancy, when performing a store, the system will check if the desired information has already been stored. If an exact match exists, the agent is notified via an error message, or else the information is stored normally. DELETE is very similar; the only difference is that it sends an error when an exact match is not present; or else that match is deleted normally.

INFER and PREDICT

These are the two main operations that assist the agent in identifying an environment or object. As part of INFER, the agent specifies a location and feature but no class. This is to replicate the act of the agent seeing a feature at a location, and then trying to remember which classes said pair belongs to. When beginning the identification process, a lookup is done on all information that is stored in the system. Any classes that contain that relevant pair are remembered as valid, and in turn, data pertaining to those valid classes will be the only data that will be used for the next lookup. During the next lookup, the classes that contain the new location-feature pair is determined out of the remaining valid classes. This continues until there is only 1 class left (successful identification) or something is seen that does not pertain to any of the relevant classes (error notification to agent).

The PREDICT operation has two variations that essentially perform the same desired functionality, as mentioned in Section 7.1.1. This operation is used when the agent is unable to perform a full lookup but it has some partial information, either a location or feature, and wishes to see if said information is present in any of the valid entries. So a lookup is performed with either just feature or just location. The lookup returns which of the current valid classes contain said feature or location as well as what location or feature that input refers to (given a location, a list of features is provided and visa versa).

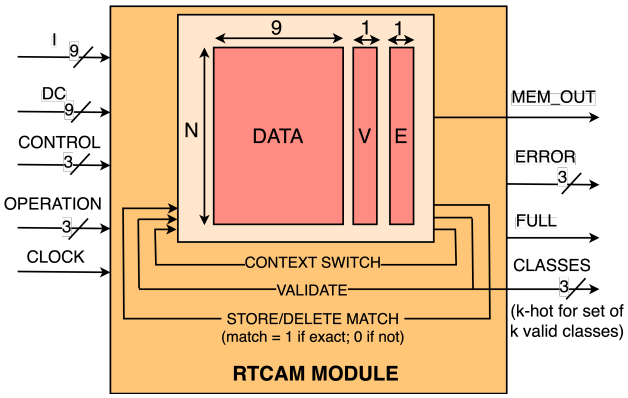


Figure 7.4: RTCAM Module: This is the memory array that stores SDRs in the form of triplets - location, feature, class, along with two additional bits for valid (V) and empty (E). Assuming 'N' number of entries with 3 bits each for location, feature and class, each entry hold 11 bits in total (indicated by Nx11). It takes in input SDR (I) and agent control commands (control) from the agent, DC mask (DC) from preprocess, and internal operation (Operation) from the state machine. It generates valid location-feature-class triplets (Mem_out) and set of k valid classes as k-hot vector (classes).

7.2 NeRTCAM Implementation

We now present each of the four components of the NeRTCAM system in further detail below.

7.2.1 Preprocess

The preprocess block is a combinational component of the system, that takes in agent control commands and padding input to generate DC vector, as shown earlier in Figure 7.2. For illustration simplicity, we use a 9 bit SDR (3 bits each for feature, location, and class). Hereafter, '*I*' refers to the input SDR from the agent, and '*DC*' refers to the generated 'don't-care' vector from the preprocess block.

- RESET and CLEAR: I and DC are completely ignored, and are expected to be held low.
- STORE and DELETE: *I* = {feature, location, class} with 3-bit non-zero 1-hot vector for each; *DC* = 9'b0. Note that for both these operations, I is completely specified and DC is ignored.
- INFER: *I* = {feature, location, 000}; *DC* = 000000111. Note, this assumes a padding value of 0; padding for fuzziness is explained in the next paragraph. Here, the agent is viewing a feature at a location and does not have any class data to match, so it is ignored.

- PREDICT Feature: I = {000, location, 000}; DC = 111000111. For this operation, the agent provides a location (with again a padding of 0). The feature and class bits are ignored as no data was provided for a match.
- PREDICT Location: I = {feature, 000, 000}; DC = 000111111. In similar fashion, the agent provides a feature, so the other two elements of the SDR are ignored.

Additionally, the system allows adding fuzziness to location search via padding. In order to obtain features corresponding to all nearby locations during *PREDICT Feature*, DC vector is padded with 1's around the '1' in the location part of the SDR. This is equivalent to introducing fuzziness to predict close-by features. Further, more precisely placed padding can mitigate the spatial information loss due to conversion from 2D grid to 1D vector. Note that padding in DC is only done for location part of the SDR as fuzziness is not meant for feature or class.

7.2.2 RTCAM

Figure 7.4 illustrates the RTCAM module with its inputs and outputs. It consists of four sub-modules as described below:

Memory Module

This is currently implemented using flip-flops (registers). Each entry in the memory module consists of (f+l+c+2) bits where 'f', 'l', and 'c' refer to the number of bits for feature, location and class respectively. The additional 2 bits denote valid (valid class) and empty (empty entry). Note that the corresponding input SDR and DC vector will have (f+l+c) bits. As shown in Figure 7.4, it has two internal outputs, *mem_out* and *valid_entry*. The latter is used by the state machine to make decisions, and the former is all the data sent to the prediction map and must be condensed during a prediction. The other output is *infer_class_out* which is a k-hot vector of the valid classes after a successful inference. The valid bits are automatically updated each cycle with a lookup.

Validation Module

Validation is crucial to this system as it keeps track of the valid classes throughout multiple identification cycles in sequence. This step is performed after each successful lookup portion of

INFER which only marks entries valid with matching feature-location pair. For this operation, after a lookup, a bit representation of all valid classes is created, 1 bit high for each class's place in the 1-hot bit structure. We then pad it with 0s until it's the same size as an input for I and pad it with 1s for DC. The validation bits are then reset and another 'lookup' is performed. The specified input is used to perform a search that ignores all bits except those that would be high to represent the valid classes. As classes are stored as 1-hot, with all 0s not valid, this means entries with classes having a 1 bit in any of the marked bits, will be marked as valid. Between each successful INFER, we output the condensed class vector so that the agent knows that classes are currently valid after this operation.

Context Switch Module

The context switch feedback loop is used to denote the system level functionality of being able to determine when the agent has started viewing a new object without performing a reset. In this case, a normal lookup fails, and rather than saying the entire inference fails, we reset the valid bits, try again, and see if there are any valid entries. If yes, then we know that the agent has switched to viewing a new object and can validate accordingly. If not, then we know that the inference simply failed, as the agent tried to perform an inference with unlearned data.

Store/Delete Match Module

When storing a new value, we need to ensure the data requested to be stored is not already in the RTCAM. For a store, we first perform a lookup of the store itself, and if we get any exact matches (1s in the valid bits of the RTCAM), then we simply reset the valid bits and await the next input. If we get no exact matches, then we find the next empty entry in the RTCAM and store the data there, setting its empty bit to 0 (now indicating not empty). We then reset the valid bits and wait for the next input. If there are no empty entries in the RTCAM, then we fail the store and notify the agent. Delete operates similarly, except finding an exact match results in the desired operation (i.e., delete) and no match results in failure of the operation and resultant error notification to the agent.

7.2.3 State Machine

The agent operations vary widely in terms of complexity. To maximize clock frequency, some of them are performed across multiple clock cycles. Some agent operations may require multiple clock cycles involving a sequence of internal "micro-operations". A set of internal operations ("micro-ops") is developed to aid the implementation of agent operations ("macro-ops") that get executed via a sequence of single-cycle internal operations. These internal operations (micro-ops) are generated by a state machine (Figure 7.5) and are listed below:

- clear: clears all the information stored in the RTCAM and sets all the empty bits high.
- reset: resets all valid bits back to 1, in turn making the system 'forget' what the agent has observed in this identification cycle/the previous lookup.
- store: adds an entry to the RTCAM and sets its empty bit low and valid bit high.
- delete: removes an entry from the RTCAM and sets the empty bit high.
- lookup: performs a query into the RTCAM; valid bits are updated and stored accordingly.
- validate: given a previously successful lookup, mark all entries that contain the valid classes as valid and output said classes.

The state machine is also responsible for error generation. It generates four types of errors to be sent to the agent: 1) *Delete_Failed* - no matching entry, 2) *Store_Failed*: data already present, 3) *Infer_Failed* - the system cannot continue as a specified lookup returned with no valid entries remaining in the RTCAM, and 4) *Context_Switch* - the system realizes that the agent has moved onto looking at a new, learned object and has adjusted accordingly.

Figure 7.5 shows the state transition diagram. The state machine consists of four states: *Starting State (SS)*, *First Lookup (FL)*, *Internal Reset (IR)*, and *Second Lookup (SL)*. Each transition is marked by the input agent control command (blue capitalized), followed by the generated internal operation (pink) to be executed by the RTCAM memory. RTCAM, in turn, provides valid (V) or not valid (NV) signals for its memory output, that the state machine uses to determine its transitions. Green lines indicate successful control commands and red lines indicate failure of those commands. *SS* is the default state and the only state in which the NeRTCAM accepts

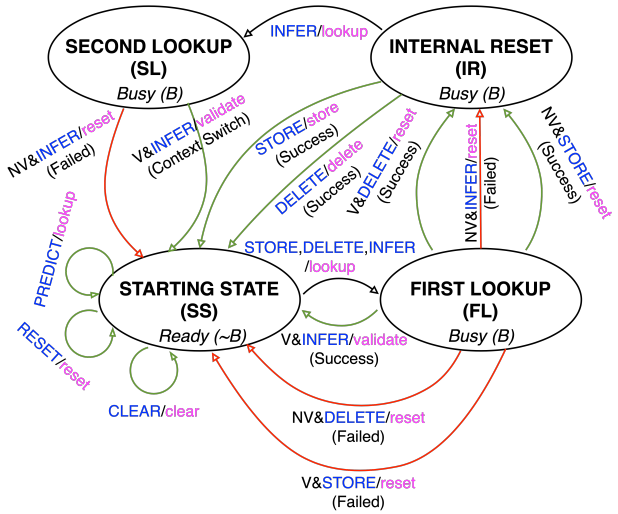


Figure 7.5: State machine: It consists of four states (SS, FL, IR, and SL). Each transition shows agent control command in blue followed by internal operation in pink, along with valid (V) and not valid (NV) bits from memory output. Status of the control command such as Success, Fail or Context Switch is provided in parentheses. Green lines indicate success and red lines indicate failure of the command.

new input from an agent; system is considered busy (B) in other states. CLEAR, RESET, and PREDICT commands execute in one clock cycle as denoted by the self-loops in *SS*. STORE and DELETE require a lookup followed by a reset of the valid bits, taking two cycles in the event of failure, and an additional cycle for the actual store/delete operation if successful. INFER is executed through a lookup followed by validate, thus finishing in two cycles if successful. On failure, it performs a reset instead of validate after lookup, and takes two additional cycles - one cycle for a second lookup and another cycle for returning with failure or success (resulting in context switch as discussed in 7.2.2). Figure 7.5 clearly depicts these transitions for every case.

7.2.4 Prediction Map

The prediction map (as illustrated earlier in Figure 7.2) is another combinational block of the system. Its primary purpose is to output the desired information specified by the control. It condenses the data given by the *mem_out* signal in the same way that the validate system condenses valid bits. However, it has the capability to do this for all 3 sections of the SDRs. The idea is to predict the set of valid features/locations/classes, with a one-hot class output indicating a strong prediction for one particular class that the agent can choose to use (i.e. a classification).

Otherwise, it just indicates the set of valid classes remaining at this point in the prediction. The same form of compaction is needed to select only the valid entries from the previous stage and condense it into a singular k-hot vector, that could then be used by the agent to distinguish which features/locations are expected to be seen next.

7.3 Results and Discussion

7.3.1 Experimental Setup & Methodology

Parameterized RTL designs for NeRTCAM and all its components are implemented in SystemVerilog. Functional RTL simulation is performed using Cadence Xcelium and logic synthesis using Cadence Genus with open-source 7nm predictive ASAP7 PDK [104]. Clock frequency used is 100 kHz to achieve real time operation similar to biological scales.

For correctness testing, rigorous testbenches are created to test every component including every state transition for the state machine. Post-synthesis power measurement is done using switching activity file generated through functional RTL simulation. The test vectors provided for power measurement are sampled from MNIST SDRs as used in [65]. Authors in [65] generate a 128-bit feature vector on a 5x5 grid using a CNN, for each MNIST digit. Aligning with this approach, we use 25 bits for location, 128 bits for feature, and 10 bits for class (for 10 handwritten MNIST digits). This implies a total SDR length of 163 bits. This SDR configuration is used to generate the post-synthesis results (not 3 bits each, which was used as an illustrative example in the earlier sections). Lastly, note that the feature vectors in [65] are not 1-hot but are 19-hot, however our system can be extended to support k-hot feature vectors as long as locations and classes are still 1-hot.

7.3.2 7nm PPA Evaluation

NeRTCAM is able to successfully demonstrate sequential inference on MNIST as presented in [65]. The SDR bit lengths for post-synthesis evaluation are determined using the same MNIST example from [65] as mentioned earlier. Accounting for the two additional valid and empty bits, each entry consists of 165 bits in total. The number of entries is varied from 64 to 1024, totaling five design configurations.

Table 7.1: 7nm CMOS Post-synthesis PPA results for NeRTCAM with the number of entries scaled from 64 to 1024. Each entry stores 165 bits = 163-bit SDR (128 bits for feature, 25 bits for location, 10 bits for class) + 1 valid bit + 1 empty bit (based on MNIST benchmark requirements from [65]).

Number of Entries	Power (mW)	Latency (μs)	Area (μm^2)
64	26.30	2.06	9,468.73
128	53.02	2.26	18,912.53
256	106.54	3.08	38,961.76
512	213.39	6.91	77,441.56
1024	400.23	9.18	153,914.94

Table 7.1 presents the post-synthesis 7nm power, performance (latency) and area results for varying storage sizes (number of entries) of NeRTCAM. Area and power scale almost linearly with respect to the number of entries, almost doubling with every 2x increase in size, whereas critical path latency scales much more gradually. The smallest assessed NeRTCAM with 64 entries incurs less than 0.01 mm^2 area and 26.3 mW power. Even NeRTCAM storing 1024 entries consumes just 0.15 mm^2 area, 400.23 mW power, and 9.18 μs latency which is very amenable for real-time operation. This area footprint is about 0.1% of typical mobile SoCs where memory dominates die area, thus demonstrating the potential and feasibility of implementing multiple NeRTCAM modules in parallel or in cascaded fashion to achieve highly scalable compute fabric.

Authors in [65] illustrate a few key desirable features for their proposed neural network (GridCellNet) inspired from Grid Cells and Reference Frame: 1) *Learning with Arbitrary Sequences*: GridCellNet significantly outperforms similar LSTM and KNN implementations when MNIST inference is performed using arbitrary sequences of feature-location inputs. In fact, in this challenging biologically motivated scenario, KNN is unable to learn well and reaches only $\sim$30% accuracy, whereas LSTM and GridCellNet achieve $\sim$65% and $\sim$80% respectively. 2) *Few-shot learning*: With arbitrarily sequenced inputs, GridCellNet achieves above 70% accuracy with only 5 training samples per class and reaches 80% with 20 samples per class, while consistently outperforming LSTM and KNN implementations. 3) *Rapid inference with partial sequences*: GridCellNet can successfully classify majority of the MNIST digits in under 10 sensations, i.e., it doesn't require all 25 sensations (at 25 locations) to infer the digit. These results from [65] indicate that

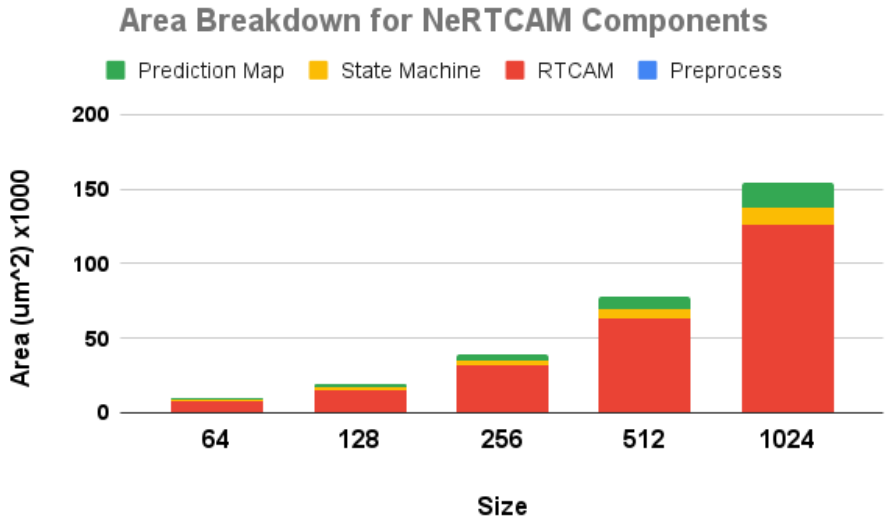


Figure 7.6: Hardware Complexity (Area) Breakdown for NeRTCAM in terms of its four components: Preprocess, RTCAM, State Machine, and Prediction Map. RTCAM consumes 82.2% of the area, followed by 10.5% for Prediction Map and 7.3% for State Machine. Preprocess incurs negligible complexity.

NeRTCAM with 500 entries (10 sensations per sample, 5 samples per class, and 10 classes) can successfully achieve >70% accuracy for arbitrary sequences of inputs outperforming LSTM and KNN implementations, while consuming just above 200 mW power and 0.08 mm^2 area. Note that GridCellNet's largest configuration (25 sensations per sample, 20 samples per class, and 10 classes) results in 5000 entries for NeRTCAM, which is an overspecification and unnecessary for successful MNIST inference.

Lastly, Figure 7.6 illustrates the hardware complexity breakdown in terms of area, for the four major components of NeRTCAM. RTCAM memory consumes majority (82.2%) of the area as expected, followed by 10.5% for Prediction Map and 7.3% for State Machine. These three components scale almost linearly with the number of entries. In contrast, Preprocess barely scales with number of entries and incurs negligible complexity. The dominant contribution to hardware complexity from RTCAM memory module implies replacement of flip-flops with custom TCAM cells in RTCAM can potentially provide significant improvements in PPA.

Part IV

Design Framework and Tools

Chapter 8

TNNGen: Design Framework for TNN

Building on Chapters 3 and 5, this chapter serves as the first attempt at realizing a design framework that can automate the design of TNNs for online sensory processing applications. The tool presented here, *TNNGen*, specifically focuses on the automated design of application-specific single-layer TNNs for online clustering of time-series sensory signals. As shown in Fig. 8.1, it leverages PyTorch [121], PyVerilog [125], Python, Cadence toolchain, and open-source FreePDK45 [126], ASAP7, and TNN7 libraries [81]. Starting from high-level modeling of TNNs in PyTorch, it enables generation of post-layout netlists of the models along with hardware metrics in a single automated flow. It facilitates the development of optimized energy-efficient designs without expert involvement, integrating previously segregated software-only and hardware-only TNN developments. Further, we enable users without EDA access to obtain key hardware results without running the actual process flow, via *forecasting*.

Other similar design frameworks have been proposed for DNNs [127, 128]. ANNA [127] performs application-specific neural architecture search (NAS) followed by translation to High-Level Synthesis (HLS) kernels utilizing pre-defined architectural templates (e.g., convolution units). SODA [128] utilizes multi-level intermediate representation (MLIR) constructs for mapping algorithmic Python models to low-level LLVM intermediate representation, which is then converted to Verilog RTL using HLS. However, both frameworks only support automation until the logic synthesis stage and are targeted towards DNN accelerators. In contrast, our proposed TNNGen generates specialized highly efficient post-layout netlists of neuromorphic TNN implementations

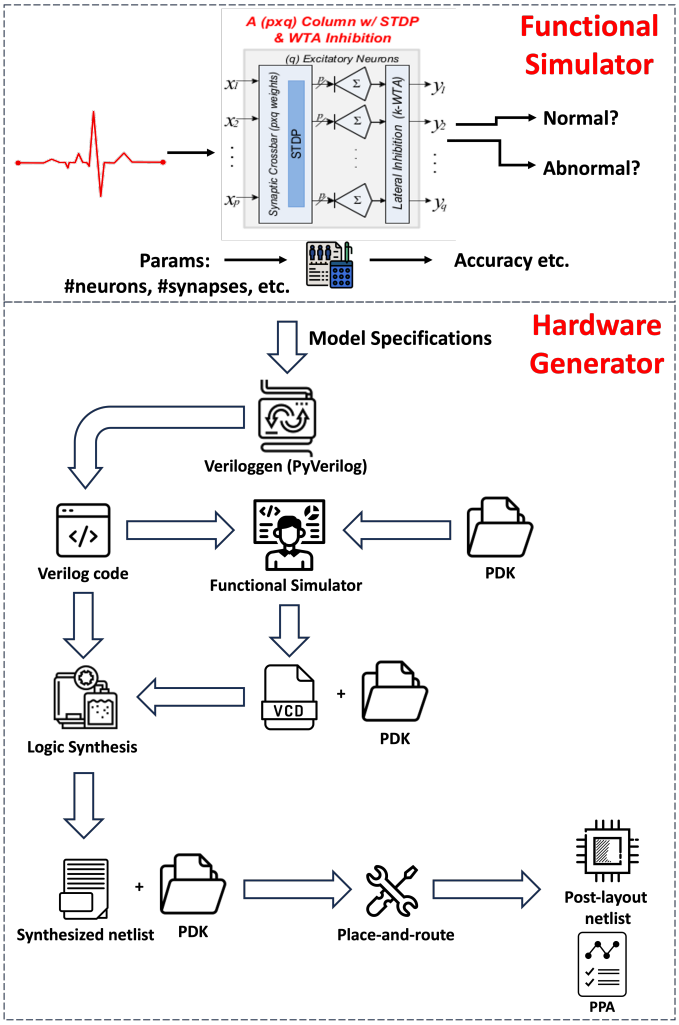


Figure 8.1: TNNGen framework for designing TNN-based neuromorphic sensory processing units. It comprises a PyTorch functional simulator and a hardware generator, and automates the entire design flow from PyTorch to chip layout.

starting with TNN functional models in PyTorch.

The next two sections details the TNNGen framework and its key components: PyTorch-based Functional Simulator (Section 8.1) and PyVerilog-based Hardware Generator (Section 8.2). Section 8.3 describes our experimental setup and evaluation results, highlighting the optimized TNN designs generated and design flow runtimes, along with *forecasting*.

8.1 PyTorch-Based Functional Simulator

A novel PyTorch-based functional simulator is developed as part of TNNGen for swift design space exploration and precise evaluation of application-specific metrics (e.g., classification accuracy, clustering rand index, F1 score, etc.). The simulator is flexible, offering users the ability to quickly explore a vast design space and use the resulting insights to develop optimized TNN models. Some key design space configurations include: (i) single-column TNNs with an arbitrary number of neurons (q) and synapses per neuron (p), and (ii) large multi-layer TNNs with an arbitrary number of layers and columns per layer with configurable inter-layer connectivity. It supports various neuron response function models (including step-no-leak, ramp-no-leak, leaky-integrate-and-fire), winner-take-all inhibition (with customizable winner count and tie-breaking options), and spike timing dependent plasticity (STDP) learning in both supervised and unsupervised modes. Pytorch's tensor operations are utilized to implement all the TNN functionalities for high simulation speed. Further, it also supports GPU acceleration via PyTorch's CUDA API.

TNNGen simulator models time precisely, aligning with the direct implementation methodology in [79] wherein spike times are dictated by precise hardware clock cycles. It performs cycle-accurate temporal modeling for time windows around onset of spikes, and dynamically switches to an event-driven approach in windows where spikes are absent to speed up the simulation. TNNGen employs a modular and parameterized approach, leveraging key functional blocks of TNNs.

8.2 PyVerilog-Based Hardware Generator

TNNGen automates the hardware design process flow by facilitating automated RTL generation, RTL simulation, logic synthesis, and place-and-route while ensuring a smooth design flow. It leverages Veriloggen package built on top of PyVerilog [125] to provide a Python interface to the user for converting PyTorch model specifications to Verilog RTL codes.

Table 8.1 specifies the process flows within TNNGen, the Cadence tools utilized during each flow, and the various libraries currently supported by the framework. Cadence EDA tools are specifically chosen as the ASAP7 and TNN7 libraries are primarily supported in the Cadence

toolchain. However, TNNGen is built with huge focus on flexibility and modularity to enable easy integration of other toolchains and libraries. We plan to open-source TNNGen for the research community to not only leverage it for their custom TNN design flow but also enhance it with additional capabilities.

In the TNNGen backend, to enable PyTorch-to-RTL conversion, we implemented all the TNN functionalities in PyVerilog, ensuring the generated RTL is highly optimized and aligns with the microarchitecture in [79]. TNN7 custom macros are incorporated to help accelerate runtime. [81] reports a 3x speedup for logic synthesis; we go a step further and report the place-and-route speedup in Section 8.3. Further, TNNGen contains a library of specifically tailored TCL scripts and templates for automating the various design flows and PDKs, while providing end-user with complete freedom to configure the flow as needed. Note that TNNGen does not cover DRC & LVS checks for signoff as it requires expert intervention.

Table 8.1: Industry EDA tools supported by TNNGen.

Design Flow Stage	**Cadence Tool**
RTL simulation	Xcelium
Logic synthesis	Genus
Place-and-route	Innovus
Library support	FreePDK45, ASAP7, TNN7

8.3 Results and Discussion

8.3.1 Experimental Setup

We adopt the same seven single-column designs in [78], targeting different sensory modalities within the time series archive from University of California, Riverside (UCR) [129], as representative benchmarks to showcase TNN designs. As shown in Table 8.2 they include: 1) *SonyAIBORobotSurface2* - classify two types of walking surfaces based on accelerometer data; 2) *ECG200* - classify normal heartbeat vs. myocardial infarction using ECG signals; 3) *Wafer* - classify normal vs. abnormal silicon wafers based on fabrication process control sensor signals; 4) *ToeSegmentation2* - classify normal vs. abnormal walking using motion sensor data, and 5) *Lightning2* - detect presence or absence of lightning based on power-density series derived from optical and RF sen-

Table 8.2: Seven different TNN configurations for various sensory modality applications used for the experimental setup. Clustering performance (rand index) for TNNs vs state-of-the-art DTCR, normalized to *k*-means is provided.

Column Size (*pxq*)	UCR Benchmark Name	Sensory Modality	DTCR Rand Index	TNN Rand Index
65x2	SonyAIBORobotSurface2	Accelerometer	0.8354	0.6066
96x2	ECG200	ECG	0.6648	0.6648
152x2	Wafer	Fabrication process	0.7338	0.555
343x2	ToeSegmentation2	Motion sensor	0.8286	0.6683
637x2	Lightning2	Optical + RF sensor	0.5913	0.577
470x5	Beef	Food spectrograph	0.8046	0.731
270x25	WordSynonyms	1D word outlines	0.8984	0.8473

sor spectrogram; 6) *Beef* - classify adulteration levels into five classes using food spectrograph data; and 7) *WordSynonyms* - classify 25 different words based on 1D series from word outlines. The designs are evaluated using three approaches:

- Develop PyTorch TNN models as per p, q parameters in Table 8.2 and report corresponding clustering performance.
- Use TNNGen to generate post-layout hardware metrics across multiple cell libraries and technology nodes.
- Predict the hardware metrics without actually running any of the process flows, using TNNGen's *forecasting* feature.

This research extends beyond previous post-synthesis studies on TNN implementations by presenting post-layout results using multiple PDK cell libraries. Further, we provide layout runtime comparisons and assess the *forecasting* feature of TNNGen that can predict post-layout hardware metrics without time-consuming EDA runs. Our runtime simulations are run on a server with 8 Intel Xeon(R) E5-2680 CPU cores.

8.3.2 TNNGen Design Performance and Hardware Complexity

TNNGen simulator is used for modeling and rapidly simulating different single-column TNN designs targeting seven different sensory modalities. To evaluate the unsupervised clustering performance, *rand index* is utilized, following the method outlined in [80]. Table 8.2 shows the

Table 8.3: Post-place-and-route die area results for the seven TNN columns targeting the seven UCR benchmarks, using three PDK cell libraries: FreePDK45, ASAP7, TNN7.

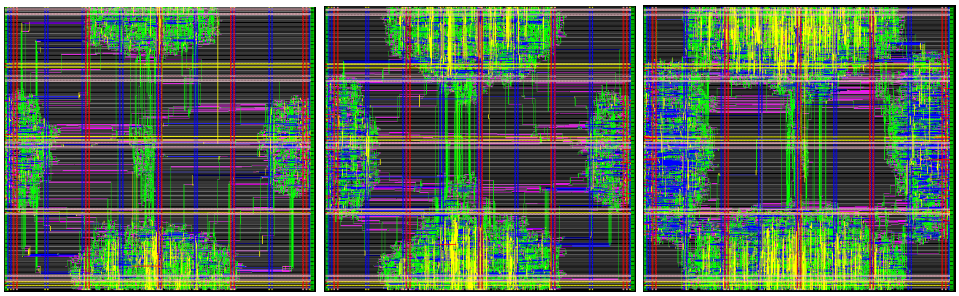
UCR Benchmark Name	Synapse Count	FreePDK45 Area (μm^2)	ASAP7 Area (μm^2)	TNN7 Area (μm^2)
SonyAIBORobotSurface2	130	14284.466	1028.67	692.06
ECG200	192	21036.08	1513.05	1015.8
Wafer	304	33868.98	2394.01	1608.52
ToeSegmentation2	686	75654.82	5388.72	3682.63
Lightning2	1274	140,502.84	10184.45	6860.68
Beef	2350	259,167.4	18298.1	12634.83
WordSynonyms	6750	744,422.4	51158.20	35303.88

Table 8.4: Post-place-and-route leakage power results for the seven TNN columns targeting the seven UCR benchmarks, using three PDK cell libraries: FreePDK45, ASAP7, TNN7.

UCR Benchmark Name	Synapse Count	FreePDK45 Leakage (mW)	ASAP7 Leakage (μW)	TNN7 Leakage (μW)
SonyAIBORobotSurface2	130	0.299	0.961	0.57
ECG200	192	0.442	1.41	0.84
Wafer	304	0.717	2.26	1.34
ToeSegmentation2	686	1.59	5.09	3.14
Lightning2	1274	2.95	9.81	5.84
Beef	2350	5.452	17.4	11.06
WordSynonyms	6750	15.66	46.69	31.13

rand index results normalized to k-means for TNN and a state-of-the-art deep learning algorithm called Deep Temporal Clustering Representation (DTCR) [130]. It can be seen that a single TNN column performs nearly as well as DTCR for four of the seven benchmarks but underperforms for the remaining three. The performance on those three benchmarks can be potentially improved by adopting more complex TNNs with multiple columns and layers. On average, DTCR outperforms TNNs by nearly 12%, aligning with the results in [80]. However, it is essential to note that in contrast to integer-based single-column TNN models, DTCR employs a significantly more complex multi-layer RNN model performing high dimensional floating point tensor algebra, rendering it impractical for deployment in mobile and edge devices due to its computational demands. Thus, the simulator results presented demonstrate the efficacy of small TNN designs for time-series clustering.

TNNGen hardware generator translates the above software models to layouts. We employ three open cell libraries, namely, 45nm CMOS FreePDK45 [126], 7nm CMOS predictive ASAP7



(a) 65x2 (79.2 ns) (b) 96x2 (93.36 ns) (c) 152x2 (98.4 ns)

Figure 8.2: Layouts of three generated column configurations fitted onto the same floorplan size. $p \times q$ column configurations are provided with computation latencies inside parentheses.

PDK [104], and TNN-tailored custom TNN7 library with nine macros as proposed in [81]. The resulting hardware metrics are reported in Tables 8.4 and 8.3.

With TNN7, there is a 32.1% and 38.6% decrease in area and leakage power compared to ASAP7, respectively, aligning with the findings in [81]. We report only leakage power here as total power requires fine-tuned physical rules specific to each design, including clock tree synthesis. Nevertheless, we do report the total power specifically for the largest column (6750 synapses) with TNN7 library for evaluation purposes.

Using the TNN7 macros, the largest column results in just 0.035 mm^2 area and consumes only 0.067 mW total (leakage + dynamic) power after layout, going beyond the post-synthesis area and total power reported in [81]. The corresponding FreePDK45 and ASAP7 area/leakage values are 0.744 mm^2/15.66 mW and 0.051 mm^2/0.047 mW respectively. The advantage of 7nm designs vs. 45nm is clear. We also see from Tables 8.4 and 8.3 that TNN7 (with custom macro cells) achieves better area and leakage than ASAP7.

For computation latency (i.e., per sample inference) evaluation, we first consider three smaller columns (65×2, 96×2, 152×2) fitted for the same floorplan size, as shown in the layouts in Fig. 8.2. The resulting computation times are 79.2 ns, 93.36 ns, and 98.4 ns, respectively. For the largest 270×25 column, the resulting latency is 180 ns. We can see that these TNN designs are extremely fast in performing inference and thereby ideal for low power real-time edge AI deployment.

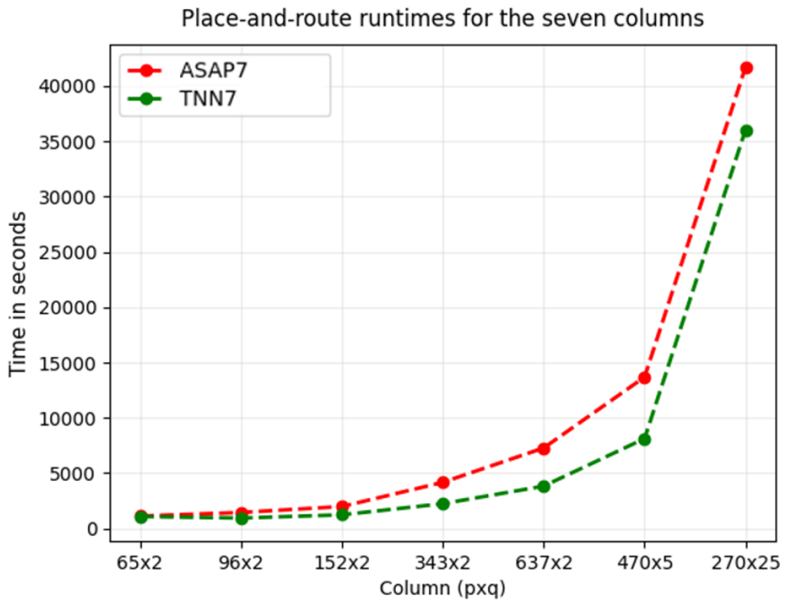


Figure 8.3: Innvous place-and-route runtime (in seconds) for baseline (ASAP7) and TNNGen (TNN7) column designs.

8.3.3 Runtime Evaluation

Authors in [81] report an approximate 3x speedup during logic synthesis due to the use of TNN7 macros during the mapping and optimization phases. We further validate and extend their empirical results by taking a step further and evaluating runtime speedup during place-and-route using Innovus.

Fig. 8.3 illustrates the runtime for place-and-route for increasing column sizes using only ASAP7 cells vs. TNN7 macros. As depicted, runtime scales with increasing column sizes, but TNN7 macros yield a slower trend. On average, the layout runtime using TNN7 in Innovus place-and-route is roughly 32% better than ASAP7. For the largest column, the entire hardware process flow (synthesis + place-and-route runtime) is reduced by almost 47%, indicating larger designs benefit more in runtime speedup with TNN7's custom macros.

8.3.4 Area and Leakage Power Forecasting

Hardware development is typically time-consuming. Many researchers may not have access to commercial EDA tools. Hence, we integrate a *forecasting* feature for predicting silicon die area based on TNN synapse count without actually running the TNNGen hardware flow. This feature

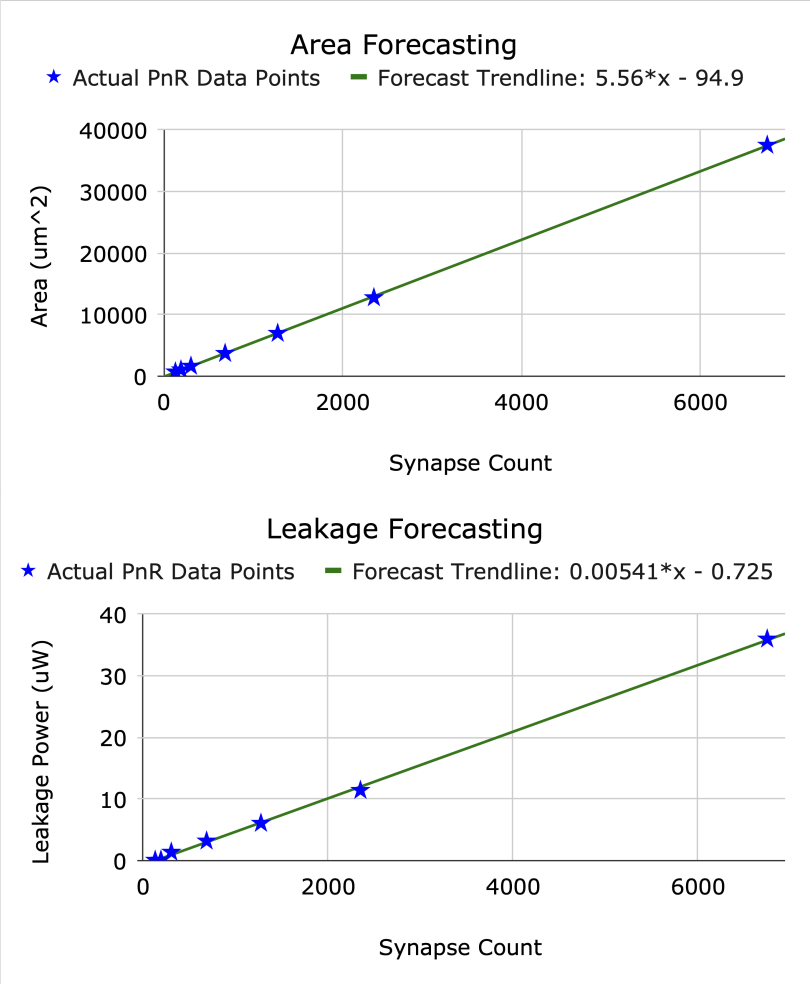


Figure 8.4: Area and Leakage power forecasting illustrating actual data points ('stars') and the forecasting trendline equations.

leverages the linear trends of area and leakage power with respect to total synapse count to build a linear regression model, which is trained on many TNNGen runs with varying TNN sizes.

The corresponding regression models for area and leakage power follow the equations:

$$Area = 5.56 \times SynapseCount - 94.9 \tag{8.1}$$

$$Leakage = 0.00541 \times SynapseCount - 0.725 \tag{8.2}$$

Fig. 8.4 along with Table 8.5 report the forecasting (FC) results for area and leakage power, along with their forecasting errors. It can be seen that area can be predicted very accurately within 1% of the original values for large designs. Leakage power, although inaccurate for small designs (omitted for the two smallest designs), is also highly accurate for large designs (the largest design

Table 8.5: Forecasted (FC) post-place-and-route 7nm PPA for seven representative UCR column designs using TNNGen.

UCR Benchmark Name	Syn. Count	FC Area (μm^2)	FC Area Error	FC Leakage (μW)	FC Leakage Error
SonyAIBORobot...	130	627.9	+10.36%	-	-
ECG200	192	972.62	+6.07%	-	-
Wafer	304	1595.34	+2.25%	0.92	+32.9%
ToeSegmentation2	686	3719.26	-0.33%	2.98	+6.14%
Lightning2	1274	6988.54	-0.25%	6.16	-1.72%
Beef	2350	12971.1	-1.7%	11.98	-5.1%
WordSynonyms	6750	37435.1	+0.2%	35.77	+0.52%

only incurs 0.52% error). Fig. 8.4 illustrates the efficacy of the linear trendline. The forecasting regression model is part of the TNNGen framework and can be continually refined with more actual design data points.

Chapter 9

C3SGen: Design Framework for C3S

Designing effective Cortical Columns Computing System (C3S) architectures requires expertise in the fundamental tenets of TNNs and CCs (e.g., temporal processing) along with the architectural design insights and microarchitectural optimizations from prior works mentioned earlier. Even if one were to achieve a deep understanding of these models and successfully map them to specific applications, pushing the design through synthesis and place-and-route requires yet another level of hardware design expertise. This underscores the need and convenience of an automated framework for design of application-specific neuromorphic processors based on TNNs and CCs. First attempt at such a framework was made in the previous chapter (Chapter 8), which proposes TNNGen as a novel framework consisting of a functional simulator and hardware generator targeting single-layer TNNs with point neurons. TNNGen enables application-specific model development in PyTorch and performs subsequent automated translation to post-layout netlists. While TNNGen demonstrates the feasibility of such a framework, it is very limited in its scope for building scalable neuromorphic processors, owing primarily due to lack of support for multi-layer TNNs and C3S architectures (cortical columns with reference frames).

Building on the TNNGen framework established in Chapter 8 and Cortical Column microarchitecture in Chapter 4, this chapter presents C3SGen as an end-to-end design framework and toolsuite that extends far beyond TNNGen in its capabilities and supports automated PyTorch-to-layout translation of significantly more complex and capable application-specific C3S architectures. C3SGen is a generalizable framework that extends beyond TNNGen to model not only

multi-layer feedforward TNNs but also adds support for active dendrites and cortical column models with reference frames.

In this work, the *clustering voter (CV)* layer serves as an example of a customized layer type that is based on the generic "TNN" layer, which can be used as a template to create user-specific application layer. The clustering voter (CV) unit [69] describes a neuron with only a single dendrite. The collection of multiple CV units is called a CV group. Each CV group is responsible for a portion of an input (i.e., MNIST image), referred to as a receptive field. The number of CV units within each CV group matches the number of classes or labels. Each CV unit is capable of casting a vote to decide whether the input belongs to the label the CV unit is assigned to. The votes can then be tallied for classification.

The differences between TNNGen and C3SGen are highlighted in Figure 9.1, depicting the much richer hierarchy adopted by C3SGen (based on Chapter 4). With this hierarchy, the C3SGen toolchain can generalize and support different implementations of neuromorphic designs. For example, a single active dendrite neuron can be modeled at the same abstraction as a layer consisting of columns with simple point neurons. Further, an active dendrite neuron incorporates two different types of synapses, distal and proximal, directly inspired from biology and adds significantly enhanced computationally capabilities compared to a point neuron [131, 132]. Traditional feedforward TNNs with point neurons can be modeled using the macrocolumn abstraction, which consists of multiple minicolumns with active dendrite neurons. An example of this is a reference frame, specifically Place Cells, as depicted in [68]. The CV units and CV groups can be viewed as special cases of neurons and minicolumns respectively. As can be seen from the figure, modeling capabilities of C3SGen far exceeds that of TNNGen. We plan to open-source this framework to facilitate further development and broader experimentation by the research community.

9.1 C3SGen Framework

C3SGen augments the existing TNNGen [84] framework with active dendrite and CV abstraction using new submodules implemented in PyVerilog. The overlaps in the required fundamental building blocks (lower left blocks in Figure 9.1) between the two toolchains allows C3SGen to

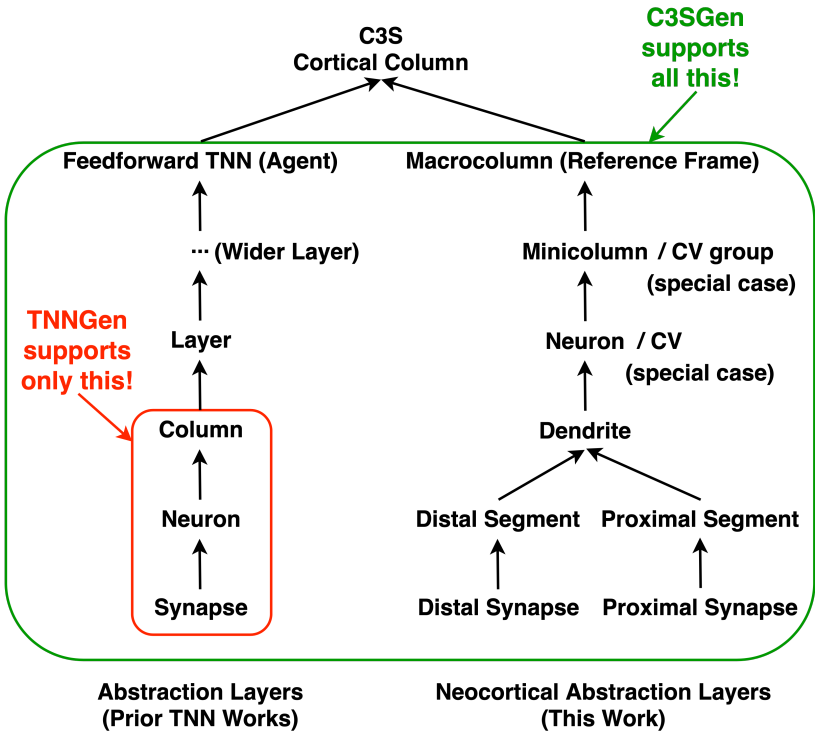


Figure 9.1: Comparison of abstraction layers for implementing hierarchical neuromorphic designs. Left bottom-up flow represents the simpler "point neuron" hierarchy as used in prior TNN works. Right flow depicts the enhanced hierarchy mimicking the neocortical abstraction layers with "active dendrite" neuron which is computationally much more powerful (two layers of abstraction above point neuron). As can be seen from the highlighted rectangles, previously proposed TNNGen [84] supports only a very limited subset, whereas C3SGen proposed in this work is much more generalizable and capable.

easily expand upon TNNGen's single-layer implementation to attain the capability of creating larger temporal compute units and eventually multi-layer networks.

To generalize the terminologies used in the work, we define the combination of multiple neurons with active dendrites as a *minicolumn*, and each TNN or CV layer can have multiple collections of minicolumns or CV groups.

9.1.1 User Configuration

Figure 9.2 provides an overview of the C3SGen workflow. The main control flow of the framework is encapsulated in a configuration text file that the user populates before running any model. One of the required fields, `flow`, controls the extent to which the design is intended to pass through the workflow. There are four core stages involved in the workflow: RTL generation,

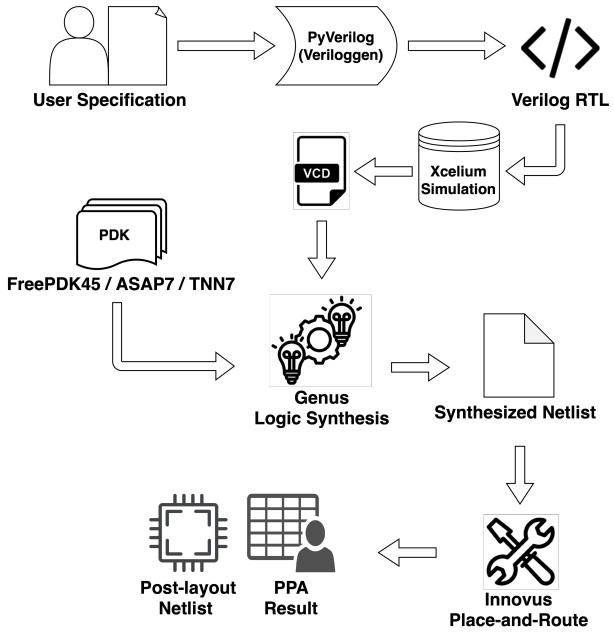


Figure 9.2: The process flow from RTL generation to layout. Veriloggen converts the software model to Verilog HDL, which is subsequently simulated using testbench-driven vectors, producing the value change dump (VCD) file. This is sourced during Genus synthesis, along with relevant standard cells and macros to generate the post-synthesis netlist. Innovus uses this netlist to generate the post-layout netlist and PPA.

cycle-level functional simulation, logic synthesis, and place-and-route.

The user also specifies which set of PDKs to use with the `node` argument. To disambiguate between ASAP7 and TNN7, a node value of 7 is chosen to represent ASAP7 and a value of 77 is used to represent TNN7. Depending on this node value, the synthesis Tcl script will be generated based on the template from Genus. Consequently, a Tcl script suited for each PDK is invoked when the user targets place-and-route.

Finally, the user specifies all the parameters related to the model in PyTorch, such as the layer type and the number of minicolumns. These parameters are automatically imported from PyTorch in to the hardware generator for further automation. In order to support parallel execution of different models, the model names are also parameterized based on configuration and the node type. The framework is designed to maximize runtime parallelism.

9.1.2 Submodules

The distinction between the two frameworks C3SGen and TNNGen begin in the neuron design. To adopt the active dendrite model, C3SGen modifies the ramp-no-leak (RNL) neuron architecture from TNNGen to include proximal spikes along with the corresponding weight update inputs in addition to the existing distal input spikes (previously described as simple input spikes). Adopting the terminology used in [69] and the new abstractions from Figure 9.1, the simple point neuron in TNNGen becomes a "segment" here. C3SGen constructs an abstraction layer on top of segments as dendrite, which is a modified version of TNN column from TNNGen that incorporates proximal inputs. Building on this hierarchical design, a TNN layer encapsulates multiple minicolumns, which contains a collection of neurons with user-configurable multi-segment dendrites.

The concept of CV also follows this hierarchical abstraction, with CV units representing a single-dendrite neuron and CV groups acting as minicolumns. Note that since CV units can only have one dendrite, the connections can be simplified in comparison with the more generic TNN layer. This can be seen in the area per synapse metric in the results section.

All the functional units highlighted in Figure 9.1 are meticulously implemented in PyVerilog at the backend of C3SGen, in a manner that the generated RTL closely matches the golden reference.

9.1.3 Layers

C3SGen includes two types of layers: generic TNN layer, and application-specific simpler CV layer. The layer instantiates all its submodules automatically based on arguments given by the user. Additionally, a kernel with configurable size and stride can be added to the input of either layer to satisfy applications such as the MNIST model in [69].

9.1.4 Model

The Model object sits at the top of the abstraction layers, serving as a wrapper for all user-defined layers. Through the `.add()` primitive, the user can add multiple layers on top of one another. Upon invocation, C3SGen checks if the input bit-width of the new layer matches the bit-width

of the previous layer, ensuring valid layer connection. Once the layer is deemed compatible with the rest of the model, it is instantiated with a unique ID, then appended to the model's list of layers. Upon detection of a mismatch, the tool suggests possible compatible parameters to the user. To visualize the model, the `.summary()` primitive prints out all layers' names and their configurations. Once the user has finished configuring the model, invoking the `.compile()` primitive would generate appropriate input and output ports for the model based on the layers added. It would then instantiate wires to connect a layer's distal input port with the previous layer's output port. The distal input port of the first layer would be connected to the model's own input port, and the output port of the last layer would be connected to the model's output port. Optionally, a kernel layer can be added before a TNN or CV layer for alternative connections.

9.2 Results and Discussion

9.2.1 Hardware Complexity for MNIST Classification

In this section, we evaluate several architectures with varying synapse counts and complexities to characterize the differences between different technology nodes and layer types. Table 9.1 and Table 9.2 summarize the post-layout leakage power and area metrics respectively for the different designs. It can be seen that both area and leakage power scale almost linearly with respect to total synapse count for both TNN-based and CV-based designs.

Computing the average of the area per synapse, we see that the TNN layer employing FreePDK45 has the largest average area-per-synapse of 111.2 μm^2 per synapse, followed by ASAP7 with 7.79 μm^2 per synapse, and TNN7 with 6.23 μm^2 per synapse. Leakage power follows the same trend. Note that the CV layers' average area is around 67.5% of TNN layers, due to the simpler design. This is also reflected on the lower leakage power of CV layers.

The last rows of Tables 9.1 and 9.2 showcase the MNIST network from [69]. It achieves 99% accuracy and incurs just under 6.67 mm^2 area in 45nm CMOS and 0.48 mm^2 area in 7nm CMOS using optimized TNN7 macros.

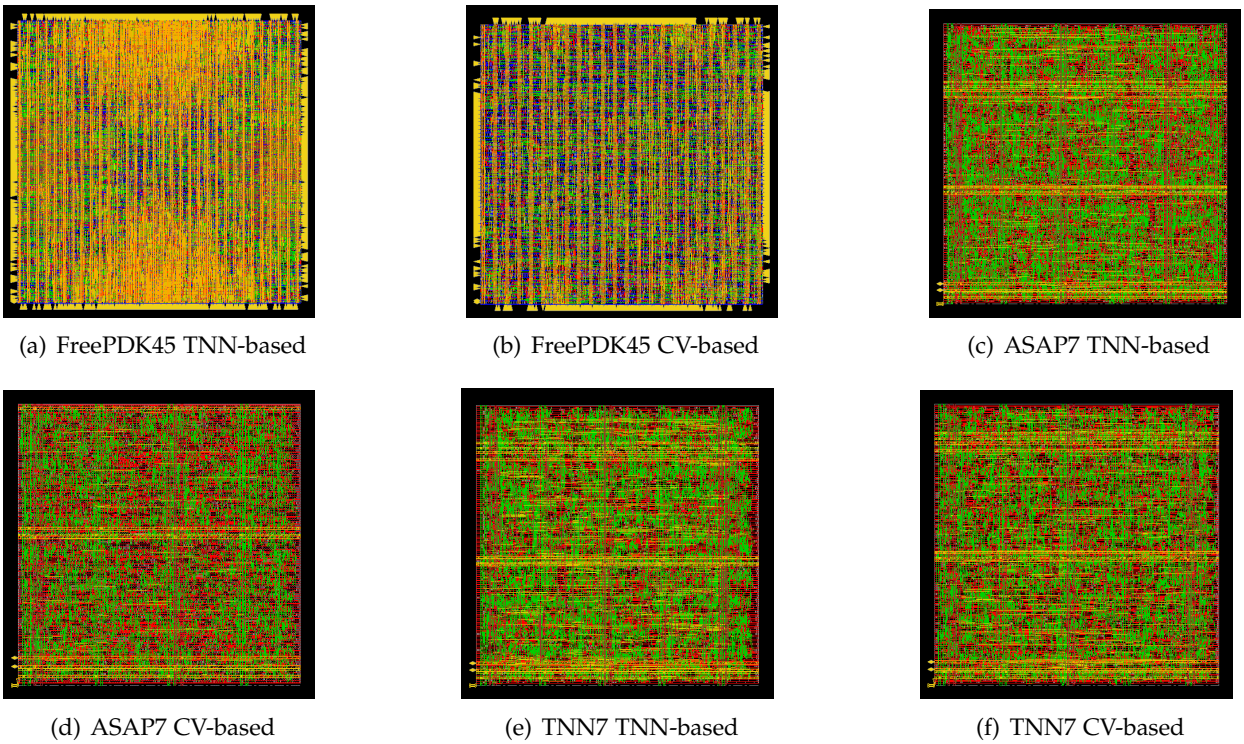
To visually observe the area differences between TNN and CV, we extract layout images of the 6 smallest designs (with 190 synapses) as shown in Figure 9.3. As evident from the figures, the CV-based designs show relatively lesser dense network of cells compared to TNN-based designs.

Table 9.1: Post-layout leakage power results for architectures with generic TNN and specialized simpler CV designs.

Layer Type	**No. of CV Groups**	**Synapse Count**	**FreePDK45 (mW)**	**ASAP7 (μW)**	**TNN7 (μW)**
TNN	1	190	0.421	1.328	1.003
TNN	2	380	0.848	2.659	2.042
TNN	4	760	1.707	5.348	4.056
TNN	8	1520	3.544	10.796	8.209
TNN	10	1900	4.573	13.672	10.267
TNN	20	3800	8.953	26.770	20.253
TNN	50	9500	27.035	71.116	52.523
CV	1	190	0.227	0.927	0.779
CV	2	380	0.458	1.878	1.565
CV	4	760	0.924	3.757	3.243
CV	8	1520	1.860	7.566	6.454
CV	10	1900	2.348	9.427	8.103
CV	20	3800	4.594	19.446	14.495
CV	50	9500	11.514	49.581	36.500
CV	576	109,440	132.641	571.173	420.48

Table 9.2: Post-layout area results for architectures with generic TNN and specialized simpler CV designs.

Layer Type	**No. of CV Groups**	**Synapse Count**	**FreePDK45 (μm^2)**	**ASAP7 (μm^2)**	**TNN7 (μm^2)**
TNN	1	190	19,743.05	1,425.85	1,158.03
TNN	2	380	39,474.40	2,877.26	2,354.48
TNN	4	760	79,807.45	5,779.82	4,688.18
TNN	8	1520	166,095.19	11,725.75	9,490.78
TNN	10	1900	216,659.13	15,246.09	11,983.78
TNN	20	3800	423,371.19	29,301.24	23,916.68
TNN	50	9500	1,243,239.31	79,537.18	59,683.06
CV	1	190	11,001.76	1,071.43	796.10
CV	2	380	22,023.20	2,157.55	1,599.76
CV	4	760	44,529.20	4,318.55	3,340.96
CV	8	1520	89,532.94	8,740.88	6,630.11
CV	10	1900	112,229.39	10,868.59	8,341.32
CV	20	3800	231,098.94	22,797.57	16,499.65
CV	50	9500	578,702.684	60,455.58	41,918.58
CV	576	109,440	6,666,654.920	696,448.282	482,902.042



(a) FreePDK45 TNN-based (b) FreePDK45 CV-based (c) ASAP7 TNN-based

(d) ASAP7 CV-based (e) TNN7 TNN-based (f) TNN7 CV-based

Figure 9.3: Layout images of 190-synapse designs. All configurations are designed to consume approximately 80% of the required die area.

9.2.2 Reference Frame (Macrocolumn) for Mouse-in-the-Dark

In this section, we demonstrate the implementation results of a reference frame using active dendrite neuron-based hierarchy. This is derived from Smith's macrocolumn architecture proposed in [68]. As mentioned earlier, a reference frame within a cortical column stores structured information about environments and objects. This was illustrated for MNIST in Chapter 7 and authors in [65]. The most intuitive benchmark to assess reference frame functionality is learning of physical spatial environments and subsequent informed navigation through them. In [68], Smith uses 40 2D environments, each 30x30 in size, with 10 features randomly placed in the environments. This setup can be used to simulate scenarios where a mouse randomly placed in an environment eventually learns to navigate to the location of a desired feature (e.g., cheese).

Table 9.3 illustrates the leakage power and area for the Place Cells used for the above Mouse-in-the-Dark benchmark in [68] with different technology nodes. A reference frame Place Cells block (implemented using active dendrite model with 324,240 synapses) that stores information

for 40 different environments with the capability for online continuous learning and updates through feedback incurs just about 1.9 mm^2 area and 1.6 mW leakage power in 7nm CMOS. This demonstrates the feasibility for much larger systems with multiple cortical columns and reference frames for biologically plausible intelligent sensory processing tasks.

Table 9.3: Place Cell Total Power & Area

Layer Type	**Library**	**Synapse Count**	**Leakage Power** (mW)	**Area** (mm^2)
TNN	FreePDK45	324240	684.36	31.903
TNN	ASAP7	324240	2.10	2.307
TNN	TNN7	324240	1.60	1.907

9.2.3 Runtime Evaluation

To verify that C3SGen maintains the runtime benefits of TNN7-based over ASAP7-based designs demonstrated in TNNGen [84], we also measure the place-and-route runtimes for some of the configurations explored in the previous section, with synapse counts varying from 190 to 1900. As seen from Figure 9.4, C3SGen shows close to 50% improvement in TNN7 runtimes compared to ASAP7, aligning with the observations from TNNGen in Chapter 8.

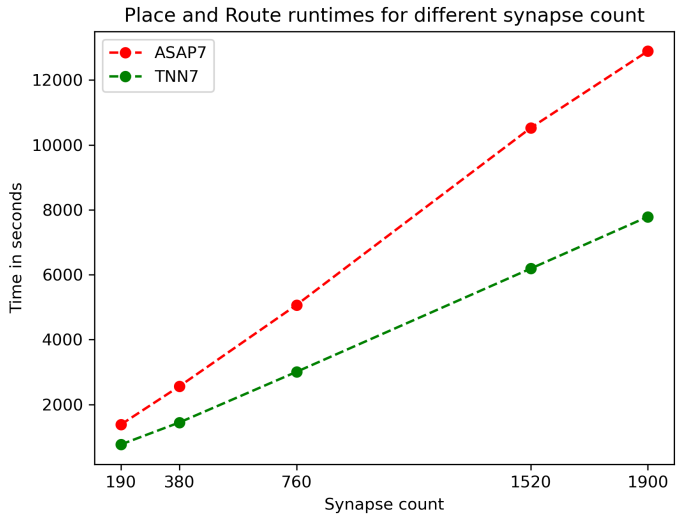


Figure 9.4: Innovus place-and-route runtime differences for ASAP7 and TNN7 for varying synapse counts from 190 to 1900.

9.2.4 Area and Leakage Power Forecasting

As evident from previous section, synthesis and place-and-route consumes significant amount of time in hardware design. Even if runtime is not a concern, not many users would have access to industry EDA tools. Hence, we present a forecasting model based on the linear trend in area and power. Figure 9.5 demonstrates the area and leakage power trends for both ASAP7 and TNN7 TNN designs from Section 9.2.1. The forecasting equations along with the actual post-layout metrics are also shown in the figure. Note that for both metrics, the predicted values align very closely with the post-layout values. Further, TNN7 scales slower than ASAP, thereby underscoring its increasing advantage for larger designs.

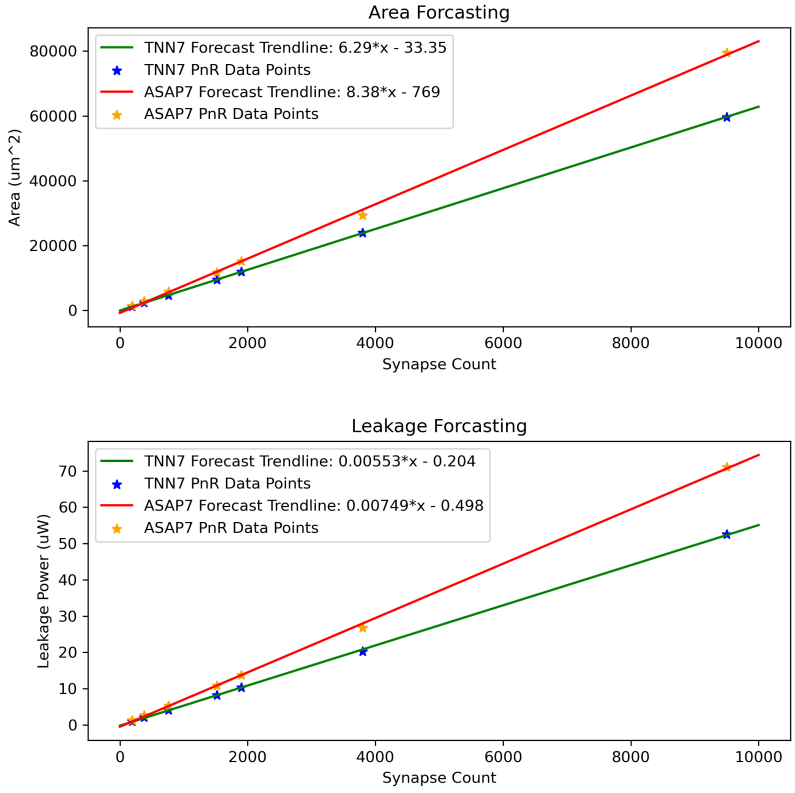


Figure 9.5: C3SGen forecasting model for post-layout area and leakage power.

Part V

Epilog

Chapter 10

Summary and Future Directions

10.1 Summary

This dissertation marks a significant first step towards implementing neocortical intelligence in digital CMOS hardware that adheres to the structural and functional hierarchy as observed in the neocortex along with support for online continuous learning. We envision such an approach as *Cortical Columns Computing Systems (C3S)*. This approach differs from existing deep neural networks (DNNs) and spiking neural networks (SNNs) in the following three major aspects:

- Truly neuromorphic tenets: C3S employs spiking neurons with active dendrites, temporal coding, and biologically plausible online continuous local learning (STDP). Most SNNs and DNNs employ a point neuron model which differs from biological neuron compute.
- Hardware implementation feasibility: In contrast to most SNN works that focus only on software simulation or employ analog hardware approaches, we target digital CMOS implementation for near-future mass market impact. C3S adopts a simple response function model operating on low-precision integers. In contrast to DNNs, no multiplication is performed. Instead, simple operations such as count, add, rank, and adjust, form the basis of C3S compute.
- Adherence to biological structural hierarchy: Most existing digital SNN implementation works implement a sea of neurons that communicate via packetized messages containing spiketime binary values. In contrast, C3S adopts a direct implementation strategy that

strictly adheres to the structural hierarchy of composable building blocks informed by experimental neuroscience and as observed in the neocortex: segments, dendrites, neurons, minicolumns, cortical columns.

This dissertation lays out a CMOS implementation methodology for C3S and employ the Temporal Logic Design (TLD) style, based on signal timings that forms the basis for representation and processing of values. Two categories of TLD are discussed, namely, edge temporal and pulse temporal. This methodology transcends beyond the C3S domain and can be utilized for accelerating traditional GEMM computations in DNNs as described in Appendix (Chapter 11).

Further, C3SGen as proposed in this dissertation makes the first leap towards creating an automated framework that enables application developers to prototype neuromorphic sensory processing units (NSPUs) based on complex cortical columns with active dendrites. The initial set of results highlight the feasibility of neocortical abstraction with “active dendrite" abstraction. The results demonstrate the efficacy of the framework, and illustrate that the newly added modules are not only a good foundation for future C3S implementations, they also provide developers a tool to explore, customize and add design options as per their application requirements.

10.2 Future Directions

C3SGen embodies a tangible collection of the research ideas presented in this dissertation. Though C3SGen provides a richer platform for C3S application developers to easily develop their hardware design, there are a few key areas that can still be improved. Firstly, support for native CAM-based macros can be added, that can be directly instantiated for efficient implementation of reference frames. Secondly, various applications for C3S across diverse sensory modalities needs to be investigated. Specifically, domains such as telecommunication, healthcare, automotive, factory automation, military and aerospace can benefit from low power intelligent edge-based sensory processing. In order to achieve this, variety of new building blocks can be added to C3SGen to suit the increasing number of applications. Especially, diverse input encoders need to be added to support various data types, such as images, audio, videos, and other time series signals. Further, there is also potential for incorporating more TNN and SNN archi-

tectures for developers to explore design choices. Finally, to make the verification of modules more robust, assertion-based testing can be integrated to the set of current testing functionalities.

Going further beyond, specialized processing units (NSPUs) can be created based on cortical columns that can be integrated into existing mobile Systems-on-Chip (SoCs) or as chiplets to achieve biological intelligence. As an even tighter integration, C3S-based functional units can be designed and added within CPUs or GPUs to enhance their efficiency for edge AI workloads.

Part VI

Appendix

Chapter 11

Temporal Logic Designs for GEMM

General matrix multiplication (GEMM) forms the fundamental building block for compute-intensive operations in deep neural networks (DNNs). Its general format is:

$$\mathbf{Y} = \alpha\mathbf{AB} + \beta\mathbf{C} \tag{11.1}$$

where **A**, **B** and **C** are generic MxN, NxP and MxP input matrices, **Y** is the MxP output matrix, and α and β are scaling factors. This work focuses on *non-scaled* GEMM operation, i.e., $\alpha = \beta = 1$.

Traditionally, GEMM was implemented as software libraries for CPUs and GPUs [133, 134]. However, more recently, dedicated GEMM hardware units have been implemented within GPUs and DNN accelerators to improve compute efficiency. The increasing demand for hardware acceleration resulted in companies like Nvidia introducing the tensor cores [135] capable of performing 4x4 matrix multiplication, and Google introducing Tensor Processing Units (TPUs) [136] with Matrix Multiply Units (MXUs). Further, with the latest push towards edge computing and on-device AI [137, 138], focus has shifted towards developing low-footprint GEMM hardware. Towards that goal, Nvidia and Google introduced Jetson Xavier NX [139], and edge TPU [140] respectively, both of which deploy reduced compute on device, albeit at the expense of inference accuracy. In the current landscape, various such lightweight systems have been proposed [141, 142], including deep learning accelerators (DLAs) in modern smartphones [143]. These systems predominantly operate on binary values and trade off inference accuracy to meet the Power-Performance-Area (PPA) constraints for edge devices.

On the other hand, *unary* compute-based implementations offer a promising alternative so-

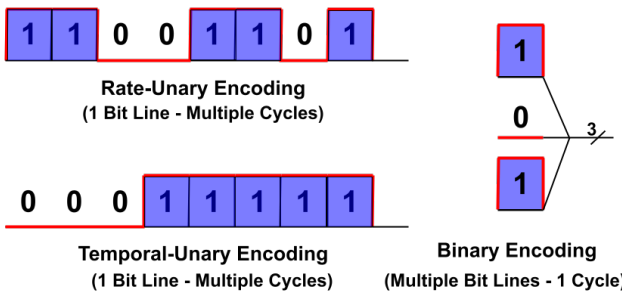


Figure 11.1: Rate-Unary Encoding vs. Temporal-Unary Encoding vs. 3-Bit Binary Encoding for the value '5'

lution to the increasing parallel computation complexity inherent to binary implementations, delivering low area and low power designs. This emerging paradigm replaces the multiple parallel bits with a single serial bit-stream. Unary computing manifests in two major forms, *rate* and *temporal* coding, with rate-based methods being more prevalent. Figure 11.1 illustrates the three data encoding schemes of: 1) rate-based unary, 2) temporal-based unary, and 3) conventional binary. Recently proposed *uGEMM* [144] is a unified rate-and-temporal encoded GEMM architecture that incorporates unary arithmetic units to perform stochastic GEMM operations. It provides significant PPA improvements compared to previous unary designs, while maintaining high accuracy, making it a promising candidate for edge devices. However, being a stochastic approach, it doesn't perform exact compute and falls under the widely researched domain of approximate computing. In contrast, our work focuses on exact, not approximate, GEMM compute based purely on temporal encoding and pulse-based Temporal Logic Design.

Recent works have emphasized the current trend of AI/DL to move towards lower precision. Authors in [145] performed training with 5-bit weights and 4-bit activations, and in [146] with 4-bit weights and 8-bit activations, both with minimal accuracy degradation. More recently, IBM researchers achieved 8-bit precision for training and 4-bit precision for inference across many deep learning datasets [147], followed by the work in [37] that shows both training and inference can be performed with 4-bits with negligible impact on accuracy. Additionally, Akida NSoCs [54] employ 1, 2, 4-bit computations for their weights and activations targeting edge inference. This growing affinity towards low precision influences this work to explore low bit-width implementations for edge AI.

11.1 *tu*GEMM

Here, we propose a novel GEMM architecture, *tuGEMM* (temporal-unary GEMM), based on exact temporal compute, targeting area-power efficiency for low precision edge AI.

11.1.1 Input Encoding

The key idea of unary hardware implementation is encoding values as serial bitstreams on a single bitline. Such an input encoding allows the hardware to be repurposed with significantly less area and power. Unary encoding can be accomplished in two ways: rate and temporal coding. Rate-based systems encode values in the frequency of ones randomly distributed across the bitstream, whereas temporal coding encodes values in the time duration for which a signal is asserted. As a result, a temporally encoded bitstream consists of consecutive ones followed by consecutive zeros, resulting in only two transitions. This naturally leads to improved dynamic power consumption, compared to rate coding with multiple signal transitions due to the distributed occurrence of ones.

Rate coding typically implements stochastically-generated bitstreams using expensive random number generators (RNGs) and suffers from the correlation problem, requiring additional hardware to mitigate it [148–150]. In contrast, temporal encoding uses a single contiguous n-cycle wide logic pulse to represent a value n, analogous to the spike encoding employed in neuromorphic computing [79], and can enable exact deterministic compute in an efficient manner as it does not require RNGs. Our proposed approach distinguishes from previous works by utilizing temporal-unary-encoded exact compute.

11.1.2 tuGEMM Architecture

The *tu*GEMM architecture (Fig. 11.2) consists of an MxP array of *output counter cells* surrounded by peripheral logic that performs unary encoding and co-ordinates the dataflow into the array. The counter array receives unary-encoded input matrices **A**, **B** from the left and top respectively, and implements multiplication in unary fashion. The multiplication compute occurs in N *steps*, where N is the common matrix dimension, which is equal to the number of columns in **A** and number of rows in **B**. Each *step* computes the outer product of i^{th} column from **A** and i^{th} row

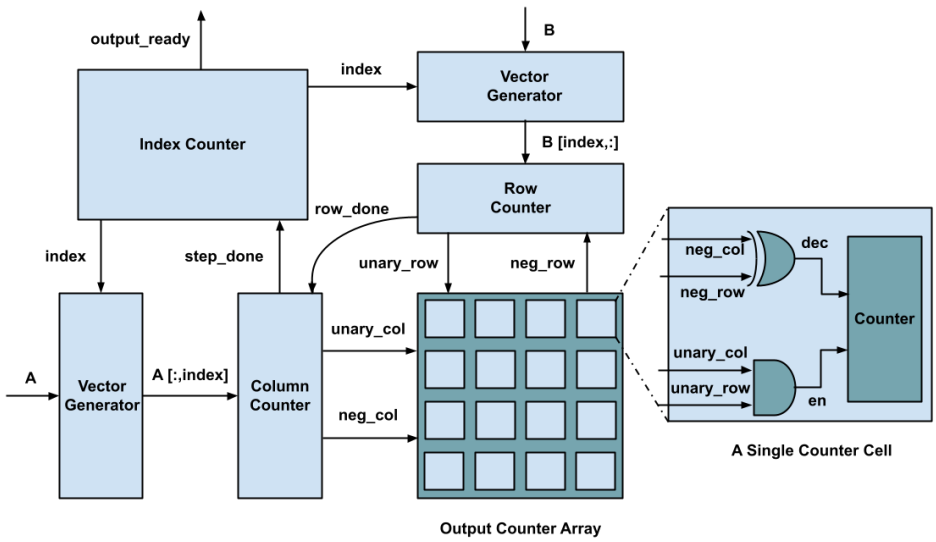


Figure 11.2: Serial tuGEMM architecture for 4x4 GEMM compute

from **B**. During each *column-row* outer product, the *M*x*P* output counters update their counts with the *M*x*P* output values, taking as many cycles as the magnitude of the maximum output value (due to unary multiplication which will be described shortly). Thus, these outer products are accumulated over the *N steps*, at the end of which the final counter values reflect the output matrix **Y**. To eliminate a separate adder, the *M*x*P* counters are initialized with the binary-encoded input matrix **C** (following sections focus only on **A x B** multiplication). The *tu*GEMM architecture performs the *N steps* serially, and is named so. It has four components:

index counter

Each column of **A** is indexed simultaneously with the corresponding row of **B**. This indexing is generated by the *index counter* that counts up from 0 to $N-1$, incrementing each time by one after every *step*, indicated by *step_done* signal. Once its count reaches N, the index counter asserts an *output_ready* signal, implying GEMM has finished.

vector generator

Two vector generators receive index i from the *index counter* and use it to index into the input matrices **A** and **B** to generate the i^{th} column of **A** (M-dimensional vector), and i^{th} row of **B** (P-dimensional vector).

column/row counters

M column counters and *P row counters* convert the binary values from the vector generator to unary signals and co-ordinates the unary multiplication. In every *step*, the column and row values from the *vector generator* are loaded into the counters, which then begin counting towards zero (decrement if the initialized count is positive, increment if negative). The counters operate in a nested fashion such that the row counters are updated by 1 every cycle whereas the column counters only update their values (by 1) once all row counters reach zero. This cycle repeats until all the column counters reach zero thus completing one *step*, eventually triggering the *index counter* for the next *step* via *step_done* signal. During every *step*, the column and row counters assert *M unary_col* and *P unary_row* signals respectively, whenever their corresponding counts are non-zero. These signals represent the converted unary signals derived from the vector generator values, and enable column-row outer product in the *output array* as will be described next. Each counter also asserts a *neg_col/row* signal, if the corresponding initialized count is negative, to determine the direction of update in the output counter cells.

output counter array

It consists of *M*x*P* counter cells (initialized with matrix **C**), where each counter accumulates the unary *column-row* outer product within a *step*, and is enabled when *unary_col/row* are both asserted. If enabled, it increments every cycle if the inputs have the same sign, else it decrements. When the *index counter* asserts *output_ready*, the output counter array holds **AB** + **C**. Note that the final result is binary, which enables direct cascading of multiple tuGEMM units as input values to the vector generators are binary.

11.2 *tub*GEMM

*tub*GEMM (temproal-unary-binary GEMM) attempts to reduce the latency overhead of *tu*GEMM by encoding one input as temporal-unary and the other as conventional binary. Here, we present our proposed *tubGEMM* microarchitecture (Figure 11.3) for matrix-multiply unit design, which consists of an *M*x*P* array of processing elements (PEs) designed to perform: **Y** = **A** x **B** + **C**, where **A**, **B**, **C** are generic *M*x*N*, *N*x*P* and *M*x*P* input matrices, and **Y** is the *M*x*P* output matrix.

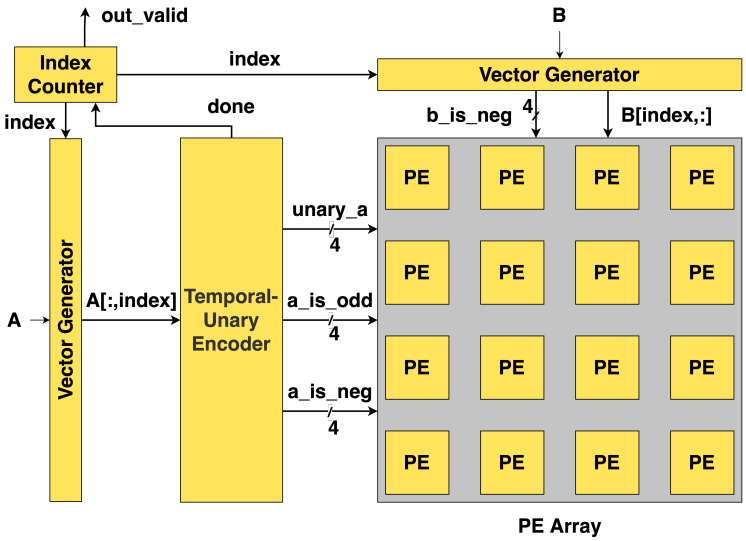


Figure 11.3: 4x4 tubGEMM Architectural Block Diagram

Alongside the tubGEMM microarchitecture, we propose a modified temporal-unary encoding scheme, *Twos-Unary (2-Unary)*, that significantly reduces the latency with minimal hardware overhead. This novel encoding optimization is described in the following subsection.

11.2.1 Twos-Unary Temporal-Unary Encoding

As discussed earlier, the hybrid *tub* approach incurs upto $(2^b - 1)$ cycles for a single multiplication, assuming each unary bit represents magnitude of 1. The encoding latency can be halved if each unary bit represents a magnitude of 2 instead. The only overhead required is a correction mechanism for odd values. This is done with no extra hardware as the existing adder can be reused to add one additional value in the last cycle (described further in the following subsections). With the *2-Unary* temporal encoding, the maximum latency is only (2^{b-1}) cycles, half of the original $(2^b - 1)$ cycle latency.

This encoding can be generalized to arbitrary n (*n-Unary*), where n is a power of 2. However, with higher powers of 2 (e.g., 4, 8), the correction mechanism requires an increasing number of shifters. Our exploration found that n=2 provides a good balance between additional complexity and latency reduction. Next, we describe how the input matrices are streamed into tubGEMM, utilizing the *2-Unary tub* approach.

11.2.2 Input Matrices Dataflow

Figure 11.3 shows the block diagram for the proposed tubGEMM with MxP nodes in the PE array. It takes in temporal-unary **A** input from the left (after the binary to the temporal-unary encoder) and binary **B** input from the top. Input **A** arrives one column (M elements) at a time, and **B** arrives one row (P elements) at a time, in lockstep fashion. Both row and column are indexed identically within their corresponding matrices, i.e., k^{th} column of **A** arrives with k^{th} row of **B**, and their outer product is computed. Thus, the computation occurs in N *steps* where each *step* performs a single *column-row* outer product (**A** has N columns, and **B** has N rows). The MxP PEs keep accumulating the MxP output values from the outer product after each *step*, taking as many cycles as the magnitude of the maximum temporal-unary input value in every *step*. If we initialize the MxP PEs with **C** matrix values, it computes **Y** at the end of N *steps*. Other than the PE array (described in Section 11.2.3), the proposed design has three components:

Index Counter

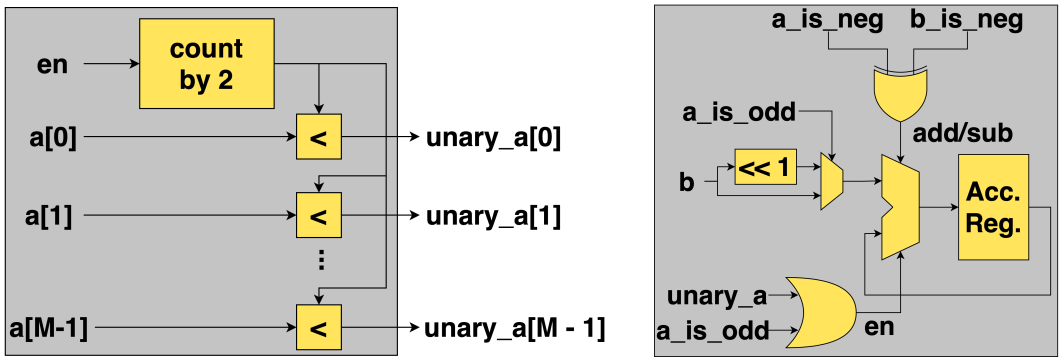
Each column of **A** is indexed simultaneously with the corresponding row of **B**. This is coordinated with the help of an *index counter* that counts from 0 to N, incrementing each time the *done* signal is asserted. Once the count reaches N, it asserts an *output_valid* signal, indicating the end of the matrix multiply computation.

Vector Generator

This component receives the index from the *index counter* and outputs a corresponding M-dimensional column of **A** and P-dimensional row of **B**. Two *vector generators* exist in the design - one each for **A** and **B**.

Temporal-Unary Encoder

As shown in Fig. 11.2.2, it consists of a single *twos*-counter that counts up with increments of 2, and a set of M comparators that convert the M binary values within one column of **A** into M unary signals. Each comparator asserts a high signal at the output for as long as the binary value is greater than the counter value ($\lfloor \frac{n}{2} \rfloor$ cycles for binary value n). Since each unary cycle counts



(a) Temporal-Unary Encoder (b) Processing Element

Figure 11.4: tubGEMM components

up by two, odd values need to have a correction factor, which is facilitated by asserting *a_is_odd* signal in the last cycle, sent to the PEs.

11.2.3 Multiply-Accumulate Processing Element

As shown in Fig. 11.2.2, each node in the PE array is a MAC unit that multiplies a unary signal with a binary signal and adds the resulting binary output to the previously stored binary value. We refer to each element within **A** and **B** matrices as **a** and **b**, respectively. A sequential multiplier is used to accumulate the binary input for as many cycles as the unary input is asserted. We employ bipolar (signed) processing where both inputs **a** and **b** are assumed to be in twos-complement format (b-bits wide), with the maximum magnitude being 2^{b-1}. With the twos-unary encoding, a single multiplication operation can take up to 2^{b-2} cycles.

Additional mechanisms are employed to handle negative and odd inputs. *a_is_odd* signal controls a multiplexer, which outputs **b** when it is enabled; 2***b** otherwise. XOR of the most significant bits from **a** and **b** determines whether to add or subtract the output of the multiplexer to the accumulating register. The sequential multiplication is enabled through the OR of the unary input and *a_is_odd* signal.

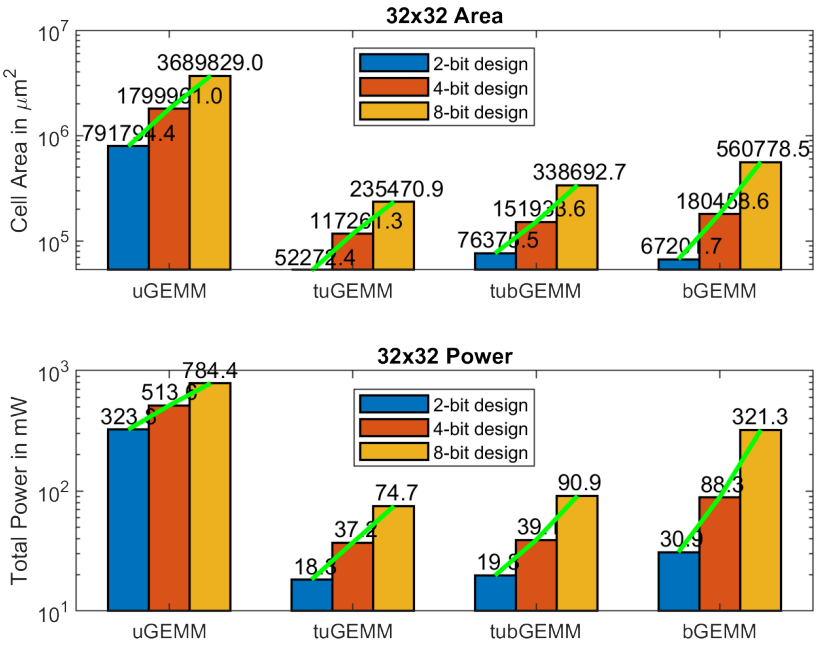


Figure 11.5: 45nm post-synthesis area and power scaling across 2, 4, and 8-bits for 32x32 matrix size. Y-axis is in log scale.

11.3 Results and Discussion

This section presents two types of evaluation for the four GEMM designs (uGEMM, *tu*GEMM, *tub*GEMM, and binary GEMM or bGEMM) across 8-, 4-, and 2-bit precisions and 16x16, 32x32 matrix sizes: 1) 45nm CMOS post-synthesis area-power results and 2) Workload-dependent sparsity and corresponding latency-energy results.

11.3.1 Area-Power Evaluation

Figure 11.5 plots the 45nm post-synthesis cell area and total power for all the designs for comparing their tradeoffs. The lowest (best) and highest (worst) values are marked in green and red, respectively. *tu*GEMM outperforms all other designs in area-power efficiency across all configurations owing to its simplistic counter-based streamlined architecture without the need for any huge adder trees. As it relies predominantly on counters for the temporal accumulation of vector-vector products, it incurs very high latency leading to highest energy consumption among all designs. *tub*GEMM is the next optimal design in area-power efficiency across 4-bits and 8-bits. bGEMM outperforms *tub*GEMM in area at 2-bits but scales considerably worse with increasing

bitwidth. uGEMM, while versatile in supporting both rate-coding and temporal-coding, exhibits lower area-power efficiency across all designs due to its unified unary approach.

More specifically, uGEMM, *tub*GEMM and bGEMM are approximately 7x, 1.5x and 2x worse than *tu*GEMM on average for 16x16 GEMM area. uGEMM scales poorly with matrix size resulting in an increased gap of 15x for 32x32 GEMM, while the gap remains consistent for *tub*GEMM and bGEMM, implying *tu*GEMM, *tub*GEMM and bGEMM scale similarly with matrix sizes. In power, *tu*GEMM and *tub*GEMM are close and consistently outperform uGEMM and bGEMM, with uGEMM consuming the most power (about 10x for 8-bit 32x32 compared to *tu*GEMM). The power efficiency of *tu*GEMM and *tub*GEMM is superior mainly because temporal-unary encoding results in only two signal transitions due to consecutive ones followed by zeros, in contrast to rate-unary and binary encoding with multiple signal transitions.

In terms of bitwidth scaling, all designs scale almost linearly (green trendlines in Figure 11.5. However, the slopes are different for each of the designs (lower slope indicates better scaling). Specifically, for area, *tu*GEMM and *tub*GEMM scale best (2.12 slope), closely followed by uGEMM (2.16) and finally bGEMM (2.90). Similarly, the four designs incur slopes of 2.02, 2.15, 1.56 and 3.25 respectively for power, indicating best scaling for uGEMM. uGEMM's superior bitwidth scaling is likely due to its simpler stochastic single-gate multiplier. In contrast, with increasing matrix sizes, the adder trees in uGEMM become significantly denser resulting in poor scaling.

Key Takeaway: *tu*GEMM has the best area-power efficiency, closely followed by *tub*GEMM and bGEMM. uGEMM is almost 10x worse. *tu*GEMM and *tub*GEMM scale well with bitwidths and matrix sizes. uGEMM scales well with bitwidths but poorly with matrix sizes, and vice versa for bGEMM.

11.3.2 Sparsity-Driven Latency-Energy Evaluation

Weight sparsity profiling results for 8 CNNs and LLaMA2-70B LLM are summarized in Table 11.1. Across most CNN models, there is a consistent word sparsity of approximately 2%, with MobileNetV3 being the outlier with 9.5% of its weights as zeros. For the LLaMA2-70B LLM model, the 8-bit self-attention tokens exhibit comparable weight sparsity, averaging around 2.8% (equivalent to approximately 2B zero weights out of the total 70B weights). 4-bit (36%) and

Table 11.1: Profiled weight sparsities of CNNs and LLaMA2-70B. Word sparsity indicates percentage of zero weights whereas bit sparsity denotes percentage of zero bits within temporal-unary-encoded weights. Higher bit sparsity leads to lower latency and energy.

CNN	**Word (%) 8 bits**	**Bit (%) 8 bits**
MobileNetV2	2.25	44.66
MobileNetV3	9.52	38.59
GoogleNet	1.91	45.91
InceptionV3	1.99	45.61
ShuffleNetV3	1.43	47.18
ResNet18	2.04	45.3
ResNet50	2.45	46.24
ResNeXt101	2.64	44.23
LLaMA2-70B Layer (LLM)	**Word (%) 2/4/8 bits**	**Bit (%) 2/4/8 bits**
Attention FC layer weight	20.7 / 2.85 / 0.0613	50.00 / 12.50 / 0.82
FFN layer weight	20.8 / 3.02 / 0.0524	50.00 / 12.5 / 0.80
Self attention Q	82.8 / 35.0 / 2.71	0.56 / 8.89 / 28.84
Self attention K	85.1 / 37.4 / 2.94	8.19 / 8.58 / 32.52

2-bit (84%) word sparsities are significantly higher. In contrast, the attention FC and FFN layers involve much lower word sparsities (negligible for 8-bits).

Bit sparsity subsumes word sparsity and directly translates to latency and energy improvements for *tu*GEMM and *tub*GEMM. CNNs display significantly better bit sparsity (~43%) compared to LLM (negligible for 8-bit FC/FFN layers and ~30% for tokens). 4- and 2-bit values show varying sparsities for LLM. Using the bit sparsity values from Table 11.1, DL workload-dependent energy values are derived for 8-, 4-, and 2-bits for 32x32 *tu*GEMM and *tub*GEMM. Note, only the two temporal-unary designs can leverage bit sparsity. These values are plotted to the right in Figure 11.6 with worst-case energy values to the left. Three notable improvements for *tub*GEMM upon leveraging sparsity are: 1) Enhanced 2-bit energy efficiency, further increasing the gap with bGEMM. 2) Earlier cross-over point with bGEMM, indicating *tub*GEMM can now outperform or perform on par with bGEMM for 3-bits. 3) More discernable energy gap with uGEMM at 8-bits.

Key Takeaway: Overall, *tub*GEMM stands out as the best design for low-precision AI infer-

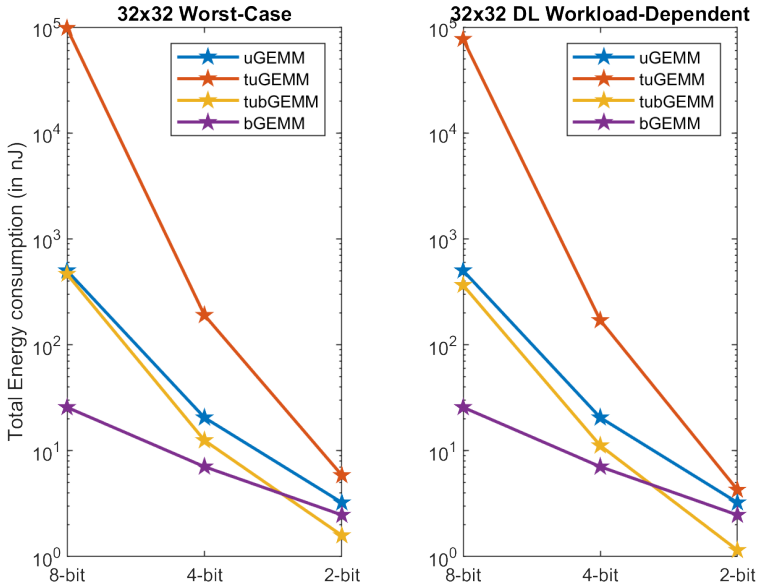


Figure 11.6: Energy consumption for 32x32 GEMM across 8-, 4- and 2-bits is shown. Y-axis is in log scale. The left and right plots display worst-case and DL sparsity-driven values (from Table 11.1) respectively. Note for *tubGEMM*, increased energy efficiency at 2 bits, earlier cross-over point with bGEMM, and larger energy gap to uGEMM at 8 bits, with sparsity.

ence (4 and 2 bits) due to its high area-power-energy efficiency, further enhanced through bit sparsity in DL workloads. *tu*GEMM's low hardware complexity makes it reasonable for applications (especially 2 bits) where area and power are highly constrained but high latency can be tolerated. uGEMM is suitable where the compute infrastructure demands rate-unary inputs, particularly for small matrix sizes. Finally, bGEMM is desirable for low-latency compute, especially for 8-bits and above.

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444 (cit. on pp. 2, 5).

[2] Brandon Reagen, Robert Adolf, Paul Whatmough, Gu-Yeon Wei, and David Brooks. "Deep learning for computer architects". In: *Synthesis Lectures on Computer Architecture* 12.4 (2017), pp. 1–123 (cit. on p. 2).

[3] John Von Neumann. "First Draft of a Report on the EDVAC". In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75 (cit. on pp. 2, 6).

[4] Hans Moravec. *Mind children: The future of robot and human intelligence*. Harvard University Press, 1988 (cit. on p. 2).

[5] Marvin Minsky. *Society of mind*. Simon and Schuster, 1988 (cit. on p. 2).

[6] Rodney A Brooks. "Intelligence without representation". In: *Artificial intelligence* 47.1-3 (1991), pp. 139–159 (cit. on p. 2).

[7] James E Smith. "A Roadmap for Reverse-Architecting the Brain's Neocortex". In: *FCRC Plenary Keynote* (2019) (cit. on p. 3).

[8] Mary J Irwin and John P Shen. "Revitalizing computer architecture research". In: *Computing Research Association* (2005) (cit. on p. 3).

[9] Carver Mead. "Neuromorphic electronic systems". In: *Proceedings of the IEEE* 78.10 (1990), pp. 1629–1636 (cit. on pp. 3, 6).

[10] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 3).

[11] Catherine D Schuman et al. "A survey of neuromorphic computing and neural networks in hardware". In: *arXiv preprint arXiv:1705.06963* (2017) (cit. on pp. 3, 7).

[12] Dennis V Christensen et al. "2022 roadmap on neuromorphic computing and engineering". In: *Neuromorphic Computing and Engineering* 2.2 (2022), p. 022501 (cit. on pp. 3, 7).

[13] James E Smith. "Space-time computing with temporal neural networks". In: *Synthesis Lectures on Computer Architecture* 12.2 (2017), pp. i–215 (cit. on pp. 3, 10, 11).

[14] James Smith. "Space-time algebra: A model for neocortical computation". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 289–300 (cit. on pp. 3, 10, 12, 13, 21, 54, 57).

[15] James E Smith. "A temporal neural network architecture for online learning". In: *arXiv preprint arXiv:2011.13844* (2020) (cit. on pp. 3, 10, 12, 42, 53, 62, 64, 65).

[16] Jeff Hawkins. *A thousand brains: A new theory of intelligence*. Hachette UK, 2021 (cit. on pp. 3, 4, 8).

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012) (cit. on p. 4).

[18] A Vaswani. "Attention is all you need". In: *Advances in Neural Information Processing Systems* (2017) (cit. on p. 4).

[19] Wayne Xin Zhao et al. "A survey of large language models". In: *arXiv preprint arXiv:2303.18223* (2023) (cit. on pp. 4, 5).

[20] John L Hennessy and David A Patterson. "A new golden age for computer architecture". In: *Communications of the ACM* 62.2 (2019), pp. 48–60 (cit. on p. 4).

[21] Kevin Lee, Adi Gangidi, and Mathew Oldham. *Building Meta's GenAI Infrastructure*. 2024. URL: https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/ (visited on 03/12/2024) (cit. on p. 4).

[22] OpenAI. *AI and Compute*. 2018. URL: https://openai.com/blog/ai-and-compute/ (visited on 12/19/2022) (cit. on p. 5).

[23] Samira Pouyanfar et al. "A survey on deep learning: Algorithms, techniques, and applications". In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–36 (cit. on p. 5).

[24] Md Zahangir Alom et al. "A state-of-the-art survey on deep learning theory and architectures". In: *Electronics* 8.3 (2019), p. 292 (cit. on p. 5).

[25] Shi Dong, Ping Wang, and Khushnood Abbas. "A survey on deep learning and its applications". In: *Computer Science Review* 40 (2021), p. 100379 (cit. on p. 5).

[26] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. "The computational limits of deep learning". In: *arXiv preprint arXiv:2007.05558* (2020) (cit. on p. 6).

[27] Andrew Lohn and Micah Musser. "AI and Compute: How Much Longer Can Computing Power Drive Artificial Intelligence Progress". In: *Center for Security and Emerging Technology (CSET)* (2022) (cit. on p. 6).

[28] Numenta Inc. *AI is harming our planet: addressing AI's staggering energy cost*. 2022. URL: https://www.numenta.com/blog/2022/05/24/ai-is-harming-our-planet/ (visited on 12/19/2022) (cit. on p. 6).

[29] Alex Lyzhov. *"AI and Compute" trend isn't predictive of what is happening (Blog Post)*. 2021. URL: https://www.alignmentforum.org (visited on 12/19/2022) (cit. on p. 6).

[30] Jaime Sevilla et al. "Compute trends across three eras of machine learning". In: *arXiv preprint arXiv:2202.05924* (2022) (cit. on p. 6).

[31] David Patterson et al. "The carbon footprint of machine learning training will plateau, then shrink". In: *Computer* 55.7 (2022), pp. 18–28 (cit. on p. 6).

[32] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015) (cit. on p. 6).

[33] Fengfu Li, Bo Zhang, and Bin Liu. "Ternary weight networks". In: *arXiv preprint arXiv:1605.04711* (2016) (cit. on p. 6).

[34] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Quantized neural networks: Training neural networks with low precision weights and activations". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898 (cit. on p. 6).

[35] Yihui He, Xiangyu Zhang, and Jian Sun. "Channel pruning for accelerating very deep neural networks". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 1389–1397 (cit. on p. 6).

[36] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. "Integer quantization for deep learning inference: Principles and empirical evaluation". In: *arXiv preprint arXiv:2004.09602* (2020) (cit. on p. 6).

[37] Xiao Sun et al. "Ultra-low precision 4-bit training of deep neural networks". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1796–1807 (cit. on pp. 6, 118).

[38] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. "Learning structured sparsity in deep neural networks". In: *Advances in neural information processing systems* 29 (2016) (cit. on p. 6).

[39] Trevor Gale, Erich Elsen, and Sara Hooker. "The state of sparsity in deep neural networks". In: *arXiv preprint arXiv:1902.09574* (2019) (cit. on p. 6).

[40] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. "Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks." In: *J. Mach. Learn. Res.* 22.241 (2021), pp. 1–124 (cit. on p. 6).

[41] Shijin Zhang et al. "Cambricon-X: An accelerator for sparse neural networks". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–12 (cit. on p. 6).

[42] Angshuman Parashar et al. "SCNN: An accelerator for compressed-sparse convolutional neural networks". In: *ACM SIGARCH computer architecture news* 45.2 (2017), pp. 27–40 (cit. on p. 6).

[43] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. "SparTen: A sparse tensor accelerator for convolutional neural networks". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 151–165 (cit. on p. 6).

[44] Dingqing Yang et al. "Procrustes: a dataflow and accelerator for sparse deep neural network training". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 711–724 (cit. on p. 6).

[45] Carver Mead and Mohammed Ismail. *Analog VLSI implementation of neural systems*. Vol. 80. Springer Science & Business Media, 1989 (cit. on p. 6).

[46] Rodney Douglas, Misha Mahowald, and Carver Mead. "Neuromorphic analogue VLSI". In: *Annual review of neuroscience* 18 (1995), pp. 255–281 (cit. on p. 6).

[47] Johannes Schemmel, Johannes Fieres, and Karlheinz Meier. "Wafer-scale integration of analog neural networks". In: *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. IEEE. 2008, pp. 431–438 (cit. on p. 7).

[48] Steve B Furber et al. "Overview of the SpiNNaker system architecture". In: *IEEE transactions on computers* 62.12 (2012), pp. 2454–2467 (cit. on p. 7).

[49] Charlotte Frenkel, Martin Lefebvre, Jean-Didier Legat, and David Bol. "A 0.086-mm^2 12.7-pJ/SOP 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm CMOS". In: *IEEE transactions on biomedical circuits and systems* 13.1 (2018), pp. 145–158 (cit. on p. 7).

[50] Charlotte Frenkel, Jean-Didier Legat, and David Bol. "MorphIC: A 65-nm 738k-Synapse/mm^2 quad-core binary-weight digital neuromorphic processor with stochastic spike-driven online learning". In: *IEEE transactions on biomedical circuits and systems* 13.5 (2019), pp. 999–1010 (cit. on p. 7).

[51] Jan Stuijt, Manolis Sifalakis, Amirreza Yousefzadeh, and Federico Corradi. "μBrain: An event-driven and fully synthesizable architecture for spiking neural networks". In: *Frontiers in neuroscience* 15 (2021), p. 538 (cit. on p. 7).

[52] Paul A Merolla et al. "A million spiking-neuron integrated circuit with a scalable communication network and interface". In: *Science* 345.6197 (2014), pp. 668–673 (cit. on pp. 7, 19).

[53] Mike Davies et al. "Loihi: A neuromorphic manycore processor with on-chip learning". In: *Ieee Micro* 38.1 (2018), pp. 82–99 (cit. on pp. 7, 19).

[54] BrainChip Holdings Ltd. *Akida Neuromorphic System-On-Chip*. URL: https://brainchip.com/akida-neural-processor-soc/ (visited on 12/19/2022) (cit. on pp. 7, 118).

[55] Dharmendra S Modha et al. "Neural inference at the frontier of energy, space, and time". In: *Science* 382.6668 (2023), pp. 329–335 (cit. on p. 7).

[56] Ben Varkey Benjamin et al. "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations". In: *Proceedings of the IEEE* 102.5 (2014), pp. 699–716 (cit. on p. 7).

[57] Xuegang Duan et al. "Memristor-Based Neuromorphic Chips". In: *Advanced Materials* 36.14 (2024), p. 2310704 (cit. on p. 7).

[58] Kjell Jørgen Hole and Subutai Ahmad. "A thousand brains: toward biologically constrained ai". In: *SN Applied Sciences* 3.8 (2021), pp. 1–14 (cit. on pp. 8, 9).

[59] Numenta Inc. *The Thousand Brains Theory of Intelligence*. URL: https://www.numenta.com/blog/2019/01/16/the-thousand-brains-theory-of-intelligence/ (visited on 12/19/2022) (cit. on p. 9).

[60] Jeff Hawkins, Subutai Ahmad, and Yuwei Cui. "A theory of how columns in the neocortex enable learning the structure of the world". In: *Frontiers in neural circuits* 11 (2017), p. 81 (cit. on pp. 8, 10, 77).

[61] Jeff Hawkins, Marcus Lewis, Mirko Klukas, Scott Purdy, and Subutai Ahmad. "A framework for intelligence and cortical function based on grid cells in the neocortex". In: *Frontiers in neural circuits* 12 (2019), p. 121 (cit. on p. 8).

[62] Marcus Lewis, Scott Purdy, Subutai Ahmad, and Jeff Hawkins. "Locations in the neocortex: a theory of sensorimotor object recognition using cortical grid cells". In: *Frontiers in neural circuits* 13 (2019), p. 22 (cit. on pp. 8, 10, 77).

[63] Torkel Hafting, Marianne Fyhn, Sturla Molden, May-Britt Moser, and Edvard I Moser. "Microstructure of a spatial map in the entorhinal cortex". In: *Nature* 436.7052 (2005), pp. 801–806 (cit. on pp. 8, 77).

[64] John O'Keefe and Dulcie H Conway. "Hippocampal place units in the freely moving rat: why they fire where they fire". In: *Experimental brain research* 31 (1978), pp. 573–590 (cit. on pp. 8, 77).

[65] Niels Leadholm, Marcus Lewis, and Subutai Ahmad. "Grid cell path integration for movement-based visual object recognition". In: *arXiv preprint arXiv:2102.09076* (2021) (cit. on pp. 10, 77–79, 88, 89, 109).

[66] Douglas Heaven. "Deep trouble for deep learning". In: *Nature* 574.7777 (2019), pp. 163–166 (cit. on p. 10).

[67] James E Smith. "(Newtonian) Space-Time Algebra". In: *arXiv preprint arXiv:2001.04242* (2019) (cit. on pp. 10, 12).

[68] James E Smith. "A Macrocolumn Architecture Implemented with Spiking Neurons". In: *arXiv preprint arXiv:2207.05081* (2022) (cit. on pp. 10, 12, 50, 78, 103, 109).

[69] James E Smith. "Neuromorphic Online Clustering and Classification". In: *arXiv preprint arXiv:2310.17797* (2023) (cit. on pp. 10, 103, 106, 107).

[70] Werner Kistler, Wulfram Gerstner, and J Hemmen. "Reduction of the Hodgkin-Huxley equations to a single-variable threshold model". In: *Neural computation* 9 (1997) (cit. on pp. 11, 36).

[71] Simon J Thorpe and Michel Imbert. "Biological constraints on connectionist modelling". In: *Connectionism in perspective* (1989), pp. 63–92 (cit. on pp. 11, 19).

[72] Simon J Thorpe. "Spike arrival times: A highly efficient coding scheme for neural networks". In: *Parallel processing in neural systems* (1990), pp. 91–94 (cit. on pp. 11, 19).

[73] Rufin VanRullen, Rudy Guyonneau, and Simon J Thorpe. "Spike times make sense". In: *Trends in neurosciences* 28.1 (2005), pp. 1–4 (cit. on pp. 11, 19).

[74] DO Hebb. "Organization of behavior". In: *J. Clin. Psychol* 6 (1949) (cit. on p. 11).

[75] WB Levy and O Steward. "Temporal contiguity requirements for long-term associative potentiation/depression in the hippocampus". In: *Neuroscience* 8 (1983) (cit. on p. 11).

[76] James E Smith. "Temporal Computer Organization". In: *arXiv preprint arXiv:2201.07742* (2022) (cit. on p. 12).

[77] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. "Race logic: A hardware acceleration for dynamic programming algorithms". In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 517–528 (cit. on p. 12).

[78] John Paul Shen and Harideep Nair. "Cortical Columns Computing Systems: Microarchitecture Model, Functional Building Blocks, and Design Tools". In: *Neuromorphic Computing*. IntechOpen, 2023. Chap. 8. DOI: 10.5772/intechopen.110252. URL: https://doi.org/10.5772/intechopen.110252 (cit. on pp. 13, 14, 16, 73, 95).

[79] Harideep Nair, John Paul Shen, and James E Smith. "A microarchitecture implementation framework for online learning with temporal neural networks". In: *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2021, pp. 266–271 (cit. on pp. 14, 16, 54–60, 62, 64, 67, 68, 71–73, 75, 94, 95, 119).

[80] Shreyas Chaudhari, Harideep Nair, José MF Moura, and John Paul Shen. "Unsupervised clustering of time series signals using neuromorphic energy-efficient temporal neural networks". In: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2021, pp. 7873–7877 (cit. on pp. 14, 16, 53, 62, 63, 96, 97).

[81] Harideep Nair, Prabhu Vellaisamy, Santha Bhasuthkar, and John Paul Shen. "TNN7: A Custom Macro Suite for Implementing Highly Optimized Designs of Neuromorphic TNNs". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2022, pp. 152–157 (cit. on pp. 14, 16, 67, 68, 92, 95, 98, 99).

[82] Harideep Nair, David Barajas-Jasso, Quinn Jacobson, and John Paul Shen. "TNN-CIM: An In-SRAM CMOS Implementation of TNN-Based Synaptic Arrays with STDP Learning". In: *2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS)*. IEEE. 2024, pp. 189–193 (cit. on pp. 15, 16).

[83] Harideep Nair, William Leyman, Agastya Sampath, Quinn Jacobson, and John Paul Shen. "NeRTCAM: CAM-Based CMOS Implementation of Reference Frames for Neuromorphic Processors". In: *2024 Neuro Inspired Computational Elements Conference (NICE)*. IEEE. 2024, pp. 1–9 (cit. on p. 15).

[84] Prabhu Vellaisamy, Harideep Nair, Vamsikrishna Ratnakaram, Dhruv Gupta, and John Paul Shen. "TNNGen: Automated Design of Neuromorphic Sensory Processing Units for Time-Series Clustering". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2024) (cit. on pp. 15, 103, 104, 110).

[85] Harideep Nair et al. "tuGEMM: Area-Power-Efficient Temporal Unary GEMM Architecture for Low-Precision Edge AI". In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2023, pp. 1–5 (cit. on pp. 15, 17).

[86] Prabhu Vellaisamy et al. "tubGEMM: Energy-Efficient and Sparsity-Effective Temporal-Unary-Binary Based Matrix Multiply Unit". In: *2023 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2023, pp. 1–6 (cit. on pp. 15, 16).

[87] Prabhu Vellaisamy, Harideep Nair, Di Wu, Shawn Blanton, and John Paul Shen. "Exploration of Unary Arithmetic-Based Matrix Multiply Units for Low Precision DL Accelerators". In: *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2024, pp. 661–665 (cit. on pp. 15, 16).

[88] Shanmuga Venkatachalam et al. "Realtime Person Identification via Gait Analysis". In: *arXiv preprint arXiv:2404.15312* (2024) (cit. on p. 16).

[89] Shanmuga Venkatachalam et al. "SemNet: Learning semantic attributes for human activity recognition with deep belief networks". In: *Frontiers in big Data* 5 (2022), p. 879389 (cit. on p. 17).

[90] Harideep Nair, Cathy Tan, Ming Zeng, Ole J Mengshoel, and John Paul Shen. "AttriNet: Learning mid-level features for human activity recognition with deep belief networks". In: *Adjunct proceedings of the 2019 acm international joint conference on pervasive and ubiquitous computing and proceedings of the 2019 ACM international symposium on wearable computers*. 2019, pp. 510–517 (cit. on p. 17).

[91] Rudy Guyonneau, Rufin VanRullen, and Simon J Thorpe. "Neurons tune to the earliest spikes through STDP". In: *Neural Computation* 17 (2005) (cit. on pp. 21, 36).

[92] Timothée Masquelier and Simon J Thorpe. "Unsupervised learning of visual features through spike timing dependent plasticity". In: *PLoS computational biology* 3.2 (2007) (cit. on pp. 21, 36).

[93] Behraoz Parhami and Chi-Hsiang Yeh. "Accumulative parallel counters". In: *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*. Vol. 2. IEEE. 1995, pp. 966–970 (cit. on pp. 25, 38).

[94] Jesper Knudsen. "Nangate 45nm open cell library". In: *CDNLive, EMEA* (2008) (cit. on pp. 30, 43).

[95] Charles M Gray, Peter König, Andreas K Engel, and Wolf Singer. "Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties". In: *Nature* 338.6213 (1989), pp. 334–337 (cit. on p. 33).

[96] Daniel A Butts et al. "Temporal precision in the neural code and the timescales of natural vision". In: *Nature* 449.7158 (2007), pp. 92–95 (cit. on p. 33).

[97] Zachary F Mainen and Terrence J Sejnowski. "Reliability of spike timing in neocortical neurons". In: *Science* 268.5216 (1995), pp. 1503–1506 (cit. on p. 33).

[98] Milad Mozafari, Mohammad Ganjtabesh, Abbas Nowzari-Dalini, Simon J Thorpe, and Timothée Masquelier. "Bio-inspired digit recognition using reward-modulated spike-timing-dependent plasticity in deep convolutional networks". In: *Pattern Recognition* 94 (2019) (cit. on pp. 35, 41).

[99] James E Smith. "A Neuromorphic Paradigm for Online Unsupervised Clustering". In: *arXiv preprint arXiv:2005.04170* (2020) (cit. on pp. 36, 42).

[100] Wolfgang Maass. "Networks of spiking neurons: the third generation of neural network models". In: *Neural networks* 10.9 (1997), pp. 1659–1671 (cit. on p. 36).

[101] Guo Bi and Mu Poo. "Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type". In: *Journal of neuroscience* 18 (1998) (cit. on p. 39).

[102] Saeed Reza Kheradpisheh, Mohammad Ganjtabesh, and Timothée Masquelier. "Bio-inspired unsupervised learning of visual features leads to robust invariant object recognition". In: *Neurocomputing* 205 (2016), pp. 382–392 (cit. on p. 40).

[103] Saeed Reza Kheradpisheh, Mohammad Ganjtabesh, Simon J Thorpe, and Timothée Masquelier. "STDP-based spiking deep convolutional neural networks for object recognition". In: *Neural Networks* 99 (2018), pp. 56–67 (cit. on p. 40).

[104] Lawrence T Clark et al. "ASAP7: A 7-nm finFET predictive process design kit". In: *Microelectronics Journal* 53 (2016) (cit. on pp. 53, 55, 88, 98).

[105] Juan P Duarte et al. "BSIM-CMG: Standard FinFET compact model for advanced circuit design". In: *41st European Solid-State Circuits Conference (ESSCIRC)*. IEEE. 2015 (cit. on p. 55).

[106] Georgios Tzimpragos, Advait Madhavan, Dilip Vasudevan, Dmitri Strukov, and Timothy Sherwood. "Boosted race trees for low energy classification". In: *Architectural Support for Prog. Languages and Operating Systems (ASPLOS)*. 2019 (cit. on p. 58).

[107] Arkadiy Morgenshtein, Alexander Fish, and A Wagner. "Gate-diffusion input (GDI)-a novel power efficient method for digital circuits: a design methodology". In: *Int'l ASIC/SOC Conference*. IEEE. 2001 (cit. on p. 60).

[108] Brandon Reagen et al. "Minerva: Enabling low-power, highly-accurate deep neural network accelerators". In: *Int'l Symposium on Computer Architecture (ISCA)*. 2016 (cit. on p. 65).

[109] Chuan-Jia Jhang, Cheng-Xin Xue, Je-Min Hung, Fu-Chun Chang, and Meng-Fan Chang. "Challenges and trends of SRAM-based computing-in-memory for AI edge devices". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.5 (2021), pp. 1773–1786 (cit. on p. 67).

[110] Sparsh Mittal, Gaurav Verma, Brajesh Kaushik, and Farooq A Khanday. "A survey of SRAM-based in-memory computing techniques and applications". In: *Journal of Systems Architecture* 119 (2021), p. 102276 (cit. on p. 67).

[111] Shimeng Yu, Wonbo Shim, Xiaochen Peng, and Yandong Luo. "RRAM for compute-in-memory: From inference to training". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.7 (2021), pp. 2753–2765 (cit. on p. 67).

[112] Harideep Nair, John Paul Shen, and James E Smith. "Direct CMOS Implementation of Neuromorphic Temporal Neural Networks for Sensory Processing". In: *arXiv preprint arXiv:2009.00457* (2020) (cit. on p. 68).

[113] Yiming Chen et al. "FAST: A Fully-Concurrent Access SRAM Topology for High Row-Wise Parallelism Applications Based on Dynamic Shift Operations". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 70.4 (2022), pp. 1605–1609 (cit. on pp. 67, 69).

[114] Kyeongho Lee, Jinho Jeong, Sungsoo Cheon, Woong Choi, and Jongsun Park. "Bit parallel 6T SRAM in-memory computing with reconfigurable bit-precision". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6 (cit. on p. 67).

[115] Jingcheng Wang et al. "14.2 A compute SRAM with bit-serial integer/floating-point operations for programmable in-memory vector acceleration". In: *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE. 2019, pp. 224–226 (cit. on p. 67).

[116] Xin Si et al. "A twin-8T SRAM computation-in-memory unit-macro for multibit CNN-based AI edge processors". In: *IEEE Journal of Solid-State Circuits* 55.1 (2019), pp. 189–202 (cit. on p. 67).

[117] Avishek Biswas and Anantha P Chandrakasan. "CONV-SRAM: An energy-efficient SRAM with in-memory dot-product computation for low-power convolutional neural networks". In: *IEEE Journal of Solid-State Circuits* 54.1 (2018), pp. 217–230 (cit. on p. 67).

[118] Amogh Agrawal, Akhilesh Jaiswal, Chankyu Lee, and Kaushik Roy. "X-SRAM: Enabling in-memory Boolean computations in CMOS static random access memories". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.12 (2018), pp. 4219–4232 (cit. on p. 67).

[119] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. "A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling

logic-in-memory". In: *IEEE Journal of Solid-State Circuits* 51.4 (2016), pp. 1009–1021 (cit. on p. 67).

[120] Kyeongho Lee, Joonhyung Kim, and Jongsun Park. "Low-Cost 7T-SRAM Compute-In-Memory Design based on Bit-Line Charge-Sharing based Analog-To-Digital Conversion". In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 2022, pp. 1–8 (cit. on p. 75).

[121] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037 (cit. on pp. 76, 92).

[122] Hoang Anh Dau et al. "The UCR time series archive". In: *IEEE/CAA Journal of Automatica Sinica* 6.6 (2019), pp. 1293–1305 (cit. on p. 76).

[123] Qianli Ma, Jiawei Zheng, Sen Li, and Gary W Cottrell. "Learning representations for time series clustering". In: *Advances in Neural Information Processing Systems*. 2019, pp. 3781–3791 (cit. on p. 76).

[124] John Paul Shen and Harideep Nair. "Cortical Columns Computing Systems: Microarchitecture Model, Functional Building Blocks, and Design Tools". In: *Neuromorphic Computing*. IntechOpen, 2023. Chap. 8. DOI: 10.5772/intechopen.110252. URL: https://doi.org/10.5772/intechopen.110252 (cit. on pp. 78, 79).

[125] Shinya Takamaeda-Yamazaki. "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL". In: *Applied Reconfigurable Computing*. Vol. 9040. Lecture Notes in Computer Science. Springer International Publishing, Apr. 2015, pp. 451–460. DOI: 10.1007/978-3-319-16214-0_42. URL: http://dx.doi.org/10.1007/978-3-319-16214-0%5C_42 (cit. on pp. 92, 94).

[126] Carlos HM Oliveira, Matheus T Moreira, Ricardo A Guazzelli, and Ney LV Calazans. "ASCEnD-FreePDK45: An open source standard cell library for asynchronous design". In: *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2016, pp. 652–655 (cit. on pp. 92, 97).

[127] Chuanyou Li et al. "ANNA: Accelerating Neural Network Accelerator through software-hardware co-design for vertical applications in edge systems". In: *Future Generation Computer Systems* 140 (2023), pp. 91–103 (cit. on p. 92).

[128] Nicolas Bohm Agostini et al. "Bridging Python to Silicon: The SODA Toolchain". In: *IEEE Micro* 42.5 (2022), pp. 78–88 (cit. on p. 92).

[129] Hoang Anh Dau et al. *The UCR Time Series Classification Archive*. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/. Oct. 2018 (cit. on p. 95).

[130] Qianli Ma, Jiawei Zheng, Sen Li, and Gary W Cottrell. "Learning representations for time series clustering". In: *Advances in neural information processing systems* 32 (2019) (cit. on p. 97).

[131] Karan Grewal, Jeremy Forest, Benjamin P Cohen, and Subutai Ahmad. "Going beyond the point neuron: active dendrites and sparse representations for continual learning". In: *bioRxiv* (2021), pp. 2021–10 (cit. on p. 103).

[132] Abhiram Iyer et al. "Avoiding catastrophe: Active dendrites enable multi-task learning in dynamic environments". In: *Frontiers in neurorobotics* 16 (2022), p. 846219 (cit. on p. 103).

[133] Nvidia. *cuBLAS*. https://docs.nvidia.com/cuda/cublas/index.html (cit. on p. 117).

[134] C. Nugteren. *CLBlast*. https://github.com/CNugteren/CLBlast (cit. on p. 117).

[135] Da Yan, Wei Wang, and Xiaowen Chu. "Demystifying tensor cores to optimize half-precision matrix multiply". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 634–643 (cit. on p. 117).

[136] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017 (cit. on p. 117).

[137] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge computing: Vision and challenges". In: *IEEE internet of things journal* 3.5 (2016) (cit. on p. 117).

[138] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. "Edge AI: On-demand accelerating deep neural network inference via edge computing". In: *IEEE Transactions on Wireless Communications* 19.1 (2019), pp. 447–457 (cit. on p. 117).

[139] Michael Ditty, Ashish Karandikar, and David Reed. "Nvidia's xavier SoC". In: *Hot chips: a symposium on high performance chips*. 2018 (cit. on p. 117).

[140] Stephen Cass. "Taking AI to the edge: Google's TPU now comes in a maker-friendly package". In: *IEEE Spectrum* 56.5 (2019), pp. 16–17 (cit. on p. 117).

[141] Ratko Pilipović, Vladimir Risojević, Janko Božič, Patricio Bulić, and Uroš Lotrič. "An Approximate GEMM Unit for Energy-Efficient Object Detection". In: *Sensors* 21.12 (2021), p. 4195 (cit. on p. 117).

[142] Zhi-Gang Liu, Paul N Whatmough, and Matthew Mattina. "Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference". In: *IEEE Computer Architecture Letters* 19.1 (2020), pp. 34–37 (cit. on p. 117).

[143] Andrey Ignatov et al. "Ai benchmark: Running deep neural networks on android smartphones". In: *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*. 2018 (cit. on p. 117).

[144] Di Wu et al. "UGEMM: Unary computing architecture for GEMM applications". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 377–390 (cit. on p. 118).

[145] Daisuke Miyashita, Edward H Lee, and Boris Murmann. "Convolutional neural networks using logarithmic data representation". In: *arXiv preprint arXiv:1603.01025* (2016) (cit. on p. 118).

[146] Naveen Mellempudi, Abhisek Kundu, Dipankar Das, Dheevatsa Mudigere, and Bharat Kaul. "Mixed low-precision deep learning inference using dynamic fixed point". In: *arXiv preprint arXiv:1701.08978* (2017) (cit. on p. 118).

[147] N Wang, J Choi, and K Gopalakrishnan. *8-Bit Precision for Training Deep Learning Systems*. 2018 (cit. on p. 118).

[148] Siting Liu and Jie Han. "Energy efficient stochastic computing with Sobol sequences". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 650–653 (cit. on p. 119).

[149] Vincent T Lee, Armin Alaghi, and Luis Ceze. "Correlation manipulating circuits for stochastic computing". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 1417–1422 (cit. on p. 119).

[150] Armin Alaghi and John P Hayes. "Exploiting correlation in stochastic circuit design". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE. 2013, pp. 39–46 (cit. on p. 119).