SANDIA REPORT
SAND2018-10553
Unlimited Release
Printed September 2018

Neural Algorithms for Low Power Implementation of Partial Differential Equations

James B Aimone, Aaron J Hill, Richard B Lehoucq, Ojas Parekh, Leah Reeder, and William Severa

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods/

SAND2018-10553
September 2018
Unlimited Release

Neural Algorithms for Low Power Implementation of Partial Differential Equations

James B Aimone[1], Aaron J Hill[2], Richard B Lehoucq[3], Ojas Parekh[4], Leah Reeder[1], and William Severa[1]
[1]Data-driven and Neural Computing, Center for Computing Research
[2]Sensors & Embedded Systems, Threat Intelligence Center
[3]Computational Mathematics, Center for Computing Research
[4]Discrete Math and Optimization, Center for Computing Research
Sandia National Laboratories
P. O. Box 5800
Albuquerque, New Mexico 87185-MS1327

Abstract

The rise of low-power neuromorphic hardware has the potential to change high-performance computing; however much of the focus on brain-inspired hardware has been on machine learning applications. A low-power solution for solving partial differential equations could radically change how we approach large-scale computing in the future. The random walk is a fundamental stochastic process that underlies many numerical tasks in scientific computing applications. We consider here two neural algorithms that can be used to efficiently implement random walks on spiking neuromorphic hardware. The first method tracks the positions of individual walkers independently by using a modular code inspired by grid cells in the brain. The second method tracks the densities of random walkers at each spatial location directly. We present the scaling complexity of each of these methods and illustrate their ability to model random walkers under different probabilistic conditions. Finally, we present implementations of these algorithms on neuromorphic hardware.

TABLE OF CONTENTS

FIGURES

NOMENCLATURE

Abbreviation	Definition
AI	Artificial Intelligence
ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
CMOS	Complementary metal-oxide semiconductor
CPU	Central Processing Unit
DSMC	Direct Simulation Monte Carlo
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
HPC	High-performance computing
LIF	Leaky Integrate-and-Fire
PDE	Partial Differential Equation
ReRAM	Resistive device Random Access Memory
RW	Random Walk

1. INTRODUCTION

Progress in computing research, and especially research in high-performance computing (HPC), has been driven largely by application demands. While there are a wide range of computing applications; scientific computing tasks, such as the solution of large systems of partial differential equations (PDEs), has long been central to the development of new computing approaches. The ability to solve large scientific computing tasks has long had recognized economic, societal, and national security importance. Whether considering global climate models for environmental studies, protein folding assessments for drug-design, financial derivative models for business risk management, or aerodynamics models for a new airplane wing design, the application of PDE-dependent applications on HPC is pervasive. Accordingly, key computing foundations that persist to modern times, such as the von Neumann architecture, were originally conceived with the application of numerically-sensitive scientific computing tasks in mind.

A notable exception to this application-driven technology development is *neuromorphic computing*. Neuromorphic computing, broadly defined, refers to computing hardware architectures designed to mimic some aspect of the human brain [17, 18]. While this broad definition has led to a plethora of somewhat unrelated approaches starting from the 1980s, in recent years there has been a consolidation towards scalable architectures that can yield significant benefits in power-efficiency and potential algorithmic capabilities. The potential for a novel low-power architecture based on the brain is being seen as a critical advance at a time when Moore's Law is thought to be slowing down and the related Dennard scaling law is believed to have ended[*] [34]. There are several aspects of neuromorphic hardware that make it well-suited for low-power computation (see Section 0). A number of large computing companies, notably Intel [11] and IBM [22], have focused on a few of these features, including *spiking* (the event-based communication of low-precision information between neurons) and extreme parallelization of neurons (targeting $>10^5$ neurons per chip), while constructing architectures that should be readily scalable. Interestingly, unlike early neuromorphic efforts that focused on analog computing, the Intel and IBM solutions, as well as a growing number of smaller academic efforts, are focusing on leveraging the best of conventional computing (such as high-density CMOS transistors and digital logic) in their designs.

While neuromorphic computing remains appealing to the broader research community due to this low-power potential and its potential suitability to emerging artificial intelligence (AI) applications, the point remains that neuromorphic hardware has emerged as a potential solution without a clear application in mind. Currently, much of the research in neuromorphic computing is focused on AI applications, which is

[*] Moore's Law is the observation by Gordon Moore of Intel that the number of transistors in an integrated circuit tends to double every couple of years[23] . This relates directly to the miniaturization of transistors over time and the associated costs of fabricating smaller devices. While smaller devices are still being forecasted, the rate has been slowing since approximately 2010, and the costs of smaller transistors are increasingly seen as prohibitive. Dennard scaling is a parallel observation that smaller transistors require less power, keeping power density on hardware constant [12]. Dennard scaling ceased to be valid around 2006; at which point thermal effects began to limit the speed at which transistors could operate.

reasonable due to their structural similarities to artificial neural networks (ANNs) and their ability to emulate other forms of neural computation that may emerge as critical AI solutions [1]. Perhaps due to the association of neuromorphic computing with low-precision analog computing or learning-dependent ANN algorithms, scientific computing applications have received comparably little attention from the neuromorphic community.

While the link between scientific computing and neuromorphic computing has remained relatively unexplored, there is potentially significant value for using a new low-power approach to computing tasks typically relegated to conventional CPUs and GPUs. As seen in Figure 1.1 below, while HPC systems have grown dramatically since 2005, the rate of growth of the world's fastest supercomputer (per Top500.org) has slowed considerably in recent years, falling below a trendline set from 2005-2011. Part of this slowing can be explained by the high-power costs associated with large HPC systems, which severely limits the growth of systems. An exception to this is the recent #1 Summit machine at Oak Ridge, which achieved significant power savings from heavy use of GPUs to achieve their performance. While GPUs are generally programmable and can be highly efficient for many tasks, their benefits are not ubiquitous, as will be discussed in later sections.

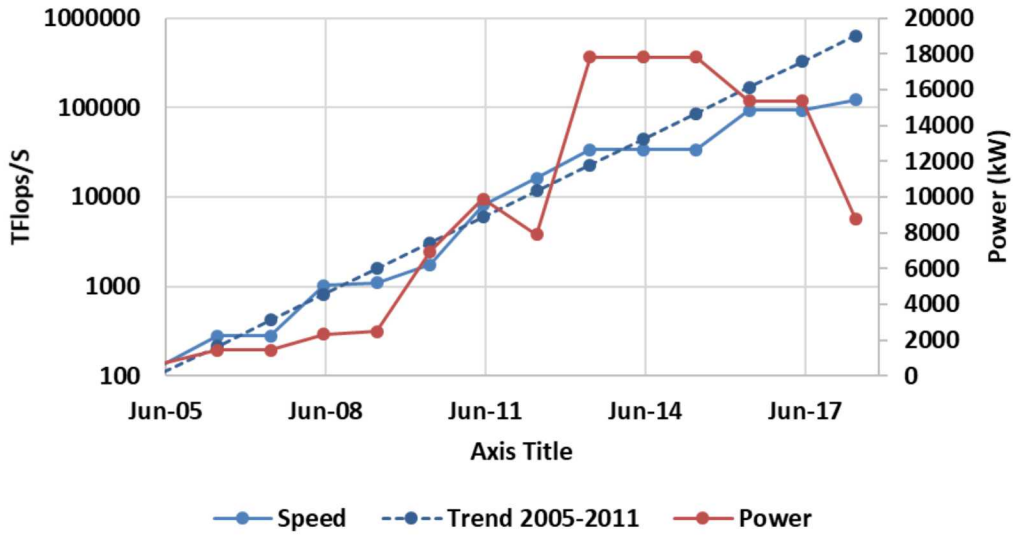

Figure 1-1: Speed and power costs of #1 Supercomputer from Top500 from 2005 until 2018. Note that aside from the 2018 leader (the GPU-heavy Summit machine at ORNL), advancing HPC systems beyond 1 petaflops has required significant increases in power.

This report considers the potential use of emerging neuromorphic computing hardware for the use in solving scientific computing tasks. Specifically, we consider the specific case where neural algorithms can provide solutions to the diffusion PDE through modeling the random walk (RW) process. Diffusion is a critical component of many scientific computing PDE problems, and the stochastic solutions leveraged in this study are currently used in a number of application areas today.

There are three important findings of this project that the subsequent chapters will expand on:

1. RW algorithms appear to be well-suited for neuromorphic architectures.
2. The neural perspective for algorithm design enables well-understood numerical techniques to be reformulated, potentially yielding novel computational benefits.
3. Current neuromorphic platforms potentially are already competitive with current HPC systems if scaled appropriately.

Combined, these findings strongly support the project's ultimate conclusion that neural algorithms, when leveraging neuromorphic hardware, merit full consideration for scientific computing applications, especially at HPC scale.

2. OVERVIEW OF NEUROMORPHIC HARDWARE

As neural-inspired algorithms have become more prevalent and more successful, there has been a renewed focus on developing neural-inspired hardware. Various neuromorphic architectures have been developed by industry as well as academia [17, 18]. This section provides a background of the neuromorphic computing field and describes the basics of the emerging model of computation that these platforms will enable.

2.1. History and overview of neuromorphic architectures

While core principals behind conventional computing have not changed dramatically since the Von Neumann architecture was introduced in the middle of the 20th century; the idea of basing future computing architectures on the brain has been discussed for nearly as long. Von Neumann's last scientific contribution was a series of lectures, written but never delivered nor fully completed, on the topic "The Computer & the Brain" [33]. This this work, written just prior to his death and in an era with limited knowledge of neurobiology, von Neumann recognized several key aspects of the brain that were distinct from the computers of his day, and indeed those that have followed. First, the brain is clearly inherently highly parallel, making it markedly different than the serial processing typically considered within computing. Second, neurons can be thought of as logic units, albeit ones with considerably more complexity than conventional Boolean logic. Third, time is non-trivial in biological neurons, though the implications of this were unclear. Fourth, the brain's memory must be of a different form than that used in conventional systems at the time. And finally, the brain seemed to benefit from a mixed digital / analog operation.

While von Neumann clearly identified many of the key features that would make neuromorphic computing a viable alternative to his namesake architecture; his observations were made several decades before there was sufficient neuroscience to constrain those concepts or capabilities in microelectronics to implement features. The first concerted effort towards neuromorphic computing, which is still felt today, was by Carver Mead in the 1980s and early 1990s [21]. Mead's work, and that of groups derived from his, has focused primarily on the relationship of *analog* computing, specifically sub-threshold transistor computing, with neuromorphic principles. Arguably, the relationship created at this time between analog computing and neuromorphic computing was so strong that the two concepts have been difficult to dissociate to this day. Early on, much of this effort focused on the ability for analog processors to emulate the analog dynamics observed in neuron electrophysiology, and work focused primarily on instantiating individual neuron or synapse dynamics efficiently. In recent years, there has been a more of an effort to scale these analog approaches up to more application-useful scales. Of particular note are the Neurogrid system out of Stanford [5] and the BrainScaleS system out of the Human Brain Project [26]. While these systems can scale to a large number of neurons and operate at high speeds, they tend to be spatially quite large due to the space required for analog components (e.g., BrainScaleS uses wafer-scale integration).

An alternative approach that has seen renewed emphasis in recent years is to forego the emphasis on analog computing and rather focus on the highly parallel implementation of neurons using digital CMOS technology. While analog presumably confers benefits in energy efficiency, analog components are challenging to fabricate reliably and tend not to allow the same transistor density that digital approaches can level. Digital neuromorphic platforms aim to achieve energy-efficiency instead by leveraging *event-driven communication* through neuron spiking (described in detail below). This approach has proven remarkably successful in seeing high-density, low-power neuromorphic capabilities. The IBM TrueNorth chip, by leveraging a hierarchical communication structure and spiking communication, has just over 1 million neurons per chip without embedded learning [22]; and the recently released Intel Loihi chip has 128K neurons per chip with embedded learning capabilities [11]. Both of these chips are extremely low-power –TrueNorth operates around 100mW for instance.

The final approach to neuromorphic computing platforms that is broadly considered in the field is the use of configurable conventional platforms, such as field-programmable gate arrays (FPGAs) and specialized ARM processors, to streamline neural computation. Often, these platforms are considered as intermediate design stage that allow an architecture or system to be prototyped in advance of investing in an application-specific integrated circuit (ASIC) that will be more efficient, such as Sandia's STPU neuromorphic FPGA architecture [16]. However, there is an increasing appreciation in the community that there is no "ideal" neuromorphic architecture, and as such a platform that preserves the flexibility of a programmable FPGA or ARM design may be preferable in many application domains. The most notable example of this is the SpiNNaker neuromorphic platform [14], from the University of Manchester under the Human Brain Project. SpiNNaker is basically a highly scalable parallel architecture of ARM-chips configured to perform lightweight neural computation. The original, and still primary, intent of SpiNNaker is to achieve realistic-scale simulations of biological neural circuits. Each SpiNNaker chip has 18 cores; 16 of which are committed to simulating ~250 neurons. SpiNNaker then uses a sophisticated communication routing system to transmit information between neurons in the systems, which can scale to thousands of chips.

It is important to note that while all three of these platforms (analog, digital, and configurable) leverage modern fabrication capabilities to enable individual operations to be performed very rapidly, the limiting factor in terms of time is most often the communication. Whether a system is modeling the brain or performing abstract neural computing applications, most often these platforms have algorithms that achieve benefits from neurons communicating at effectively large distances on the chip. The spike-based communication used in neuromorphic platforms is highly energy-efficient, but typically speaking the numerous stages of routing (however implemented) forces the processing to occur at a far slower rate than typically seen in conventional electronics. For instance, while each SpiNNaker core operates around 180MHz, the overall SpiNNaker platform operates at a significantly slower speed (~10k model updates per second). TrueNorth is similarly throttled back to run at similar speeds to avoid collisions in the spike communication.

While each of these approaches as clear benefits and challenges; it is important to note that the field is rapidly evolving. Each successive generation of neural processor adds new capabilities (such as learning on Loihi) and pushes previous limitations in neuron or synapse density and power. Further, these platforms are positioned to take advantage of parallel efforts in microelectronics; such as the development of resistive memory technology (ReRAM) [3, 13, 30]. Although memristors or other emerging memory technologies such as phase-change devices are sometimes called neuromorphic simply because of their energy-efficiency and analog nature; they are not inherently neuromorphic in isolation. However, these technologies are potentially very well-suited for achieving higher synapse density or even potentially more compact neuronal dynamics [25]. Combined with the aforementioned neuromorphic architectures, it is reasonable to expect that the capabilities of brain-inspired chips to continue to improve in the near to medium term.

2.2. Core computation in neuromorphic hardware

There are two fundamental components that all neuromorphic platforms have to have: *neurons* and *synapses*. How these two components are implemented and how these neurons and synapses interact with one another differ considerably across architectures, and not surprisingly the details of how neurons and synapses are implemented on a given architecture often matter considerably when it comes to whether a neural algorithm is an appropriate fit for a given hardware system. This section describes each of these two conceptually and provide considerations of their varying implementations, as well as a discussion about how neural algorithms can be considered generically above the neural hardware considerations.

2.2.1. The neuron

As the name implies the workhorse of neuromorphic computing is the neuron itself. Biological neurons are incredibly complex entities, exhibiting immense diversity in dynamics (most neurons have action potentials, aka *spikes,* though several populations do not), influence (some neurons have excitatory impact on targets, some are inhibitory, and some "modulate" the computation of other neurons), and size (motor neurons can extend for over a meter; certain interneurons are less than a few dozen microns in size). This diversity is not necessarily surprising for a genetically-encoded system, and it is perhaps advantageous within the organic, three-dimensional substrate of the brain. However, this diversity does present a challenge to neuromorphic hardware, as there cannot be a formal mathematical definition for what a neuron is available from the neuroscience perspective.

From a neural computing perspective, it is useful to define the dynamics of a neuron as follows:

$$C\,{}^{dV(t)}/_{dt} = \sum I_{ions}(t,V) + I_{syn}(t,V) + I_{dendrites}(t,V)$$

Where C is the capacitance of the neuron soma, $V(t)$ is the neuron voltage, at time t, $I_{ions}(t, V)$ represents the collection of ionic currents (i.e., Na+, K+, Cl- in a biological neuron), $I_{syn}(t, V)$ is the synaptic input at t, and $I_{dendrites}(t, V)$ is current that may be

arising from some other part of the neuron, typically from the dendrites. Note how in the general case, the local ionic currents, intracellular currents, and synaptic currents are all potentially a function of the local voltage.

For a Hodgkin-Huxley neuron, the ionic currents are quite complex, requiring at least several additional differential equations. When configured correctly, the structure of those dynamics is sufficient to intrinsically produce an *action potential*, or the neuron's "spike," when a voltage threshold is crossed. Further, simulating the dynamics of a morphologically complex neuron often requires a large number of spatial components, further increasing the size of the model. Likewise, a thorough biophysical accounting of synapse dynamics can become complex, especially if additional neurotransmitters, such as calcium which is involved in learning, are considered.

For neuromorphic hardware, it is prudent to reduce the dynamical description as much as is possible given the desired neural application. There are a rather large number of reduced neural models that have been proposed, each aiming to maximally account for a few aspects of neuronal computation while minimizing disruption to other behaviors. Furthermore, there is a value to representing spikes discretely (i.e., as an "all or none" event) for communication, as opposed to the extended action potential dynamics present in a Hodgkin-Huxley formulation. For these reasons, artificial spiking neurons typically have a form as follows:

$$C\,{}^{dV(t)}/_{dt} = I_{syn}(t,V) - g_{leak}V(t) + I_{dendrites}(t,V)$$

$$if\ (V(t) > V_{threshold})\ \begin{matrix} f(t) = 1 \\ V(t) \leftarrow V_{reset} \end{matrix}$$

Where $f(t)$ is the firing of a neuron at time t, $V_{threshold}$ is the voltage at which a neuron spikes, V_{reset} is the voltage that the neuron takes after a spike, and g_{leak} is the conductance of an artificial "leak" current. This model, known as a "leaky-integrate and fire" or LIF neuron, may take a similar form, but generally reduces the computational complexity by consolidating ionic currents into a single "leak" current and by eliminating all of the dynamics associated with action potential generation and recovery by a discontinuous discrete "spike" and "reset" operation. Further, many implementations – although not all – treat neurons as "point neurons," whereby there are no compartments and thus no dendritic current. Similarly, most neuromorphic implementations make simplifying assumptions on the synapses, as will be discussed in the next section.

2.2.2. The synapse

While neurons are the source of the most substantial computation in neural models; in practice the synaptic connections between neurons are typically the computationally most prohibitive. This is the case for both biological models as well as neural algorithms. Individually, even the most complex synapse model is generally far simpler than its associated neuron model; however, there are typically hundreds or thousands of synapses per neuron.

The complexity of synapse models can vary considerably, especially if the synapse is capable of learning (through a mechanism such as spike-timing dependent plasticity or long-term potentiation / depression). As the applications described here do not involve learning, and as most neuromorphic chips do not leverage learning (although this is changing) we will focus on basic non-learning synapse dynamics.

There are three general classes of synapse models that are worth considering for neuromorphic hardware. First is what is known as a conductance based synapse, whereby

$$I_{syn}(t) = g_{syn}(t)(E_{syn} - V(t))$$

$$g_{syn}(t) = g_{syn,0}e^{-\Delta t/\tau}$$

In this model, a spike opens a conductance of level $g_{syn,0}$, which progresses to decreases exponentially from the time of the spike (Δt) according to the time constant τ. The associated synaptic current is then the product of the open conductance times the difference of neuron's voltage and the reversal potential of that synapse (more precisely, the reversal potential of the neuromodulator associated with that synapse). For instance, the reversal potential for glutamate, the standard excitatory transmitter, is 0mV, so the amount of current due to a glutamatergic synapse is proportional to the neuron's voltage. In contrast, the reversal potential for chlorine is roughly -80mV, making the opening of that conductance inhibitory.

Conductance-based synapses are more biologically accurate and provide some interesting dynamics such as shunting inhibition, but in general for neuromorphic hardware they are somewhat more complex than desirable. For this reason, most neuromorphic platforms simplify synapses. The second model widely used in neural algorithms is the current-based synapse, whereby

$$dI_{syn}(t)/_{dt} = -\tau I_{syn}(t) + f_{source}(t)w_{syn}$$

Where the synaptic current exponentially decays according to τ, while new synaptic inputs are provided as a fixed delta function when the source neuron spikes (or offset by an appropriate delay), multiplied by a synaptic weight, *w*. In this case, the weight is not strictly a conductance, but rather a weighted value that increases the overall current through that synapse. Importantly, this model for synapses allows all of the similar synapses (i.e., the same τ) onto a neuron to be summed together easily and efficiently, meaning that a neuron only need to keep track of the source activations and one (or two) synaptic currents, not thousands of independent synaptic sources.

Finally, the simplest synapse model is the basic point synapse.

$$I_{syn}(t) = \sum_{sources} f_{source}(t)w_{source,syn}$$

Point synapses are not particularly biological, rather they are more similar to artificial neural net synapses. However, in neuromorphic hardware, they can be useful approximations because there are no dynamics that have to be tracked – at any given

time step the currents from active synapses are all accumulated and sent to the neuron for processing.

While the computational complexity of these synapse types does not seem markedly different, the large number of synapses in most neural algorithms makes the differences quite striking. A point synapse is highly compact and, when combined with event-driven computation, requires relatively few computing operations to process. Further, compact memory technologies, such as SRAM or envisioned ReRAM, likely will achieve maximal advantage by point-like synapses. In contrast, a conductance based model requires several state variables and parameters, six or seven operations per synapse per timestep, and the decay means that even when no spiking events occur they must still be checked. Given that there are potentially millions of synapses on a chip, these differences in complexity can become quite prohibitive. For this reason, neuromorphic hardware has leaned towards the simpler synapses.

2.2.3. *Connectivity*

The final consideration around neuromorphic hardware concerns the routing of spikes. While neurons and synapses are relatively local, consisting of a few state variables that must be updated according to some rules at each time step, the connections between neurons and synapses can be quite extended. In the brain, a typical neuron receives ~10^4 synapses, and of these a large number are often what would be considered "non-local," in that they come from neurons that are located relatively far away within the brain. The 3D configuration of the brain, coupled with the incredible efficiency and reliability of axons (the output channels of neurons) makes this feasible. However, on 2D silicon where long-distance communication is costly, the ability to reliably and efficiently communicate from two arbitrary neurons is a non-trivial challenge.

Like other design decisions, there are a few approaches to spike-based communication on neuromorphic hardware. The most common historically has been "Address Event Representation," or AER. This protocol is simply that a spiking neuron communicates its address (typically a binary neuron id), and the routing substrate then passes that address by all other potential neurons (typically through some routing process) that check if those neurons are on the receiving end of a spike from that source. While AER is effective and increasingly a commonly-used standard for communication between spiking sensors and hardware, it is not the only approach.

While AER is source-addressed, with all targets receiving it and checking for suitability, communication on TrueNorth is destination-addressed, wherein a firing neuron sends the spike off with a pre-set path to a target location. Once that spike hits the target core (which consists of 256 possible neuron outputs), a cross-bar like lookup mechanism is used to check which targets get activated. Other methods, such as SpiNNaker and the STPU, use hybrid scheme by which a source address, like AER, is used to communicate at large distances but then a subsequent routing set-up based on the local chip architecture is used to distribute the spikes locally.

Unlike the neuron and synapse models, the precise manner of communication is more of an architectural feature or cost that needs to be accounted for in forecasting

performance, but it rarely disrupts an algorithm's performance. That is unless a hardware platform imposes restrictions on connectivity along one dimension or another. In practice, all hardware platforms will have some limitation, however this is more often the case with platforms such as TrueNorth, which restrict outputs to 256 targets (along with a few other restrictions).

2.2.4. Neuromorphic algorithm design

When considering the above descriptions, the potential programming of a neuromorphic platform may seem both complex and unconstrained, particularly since most well-known neural algorithms for computing (e.g., neural networks) do not leverage many of the dynamics discussed here.

To simplify the discussion, the algorithms described in the following sections are rather simple; using a simple LIF model with no spatial complexity. Some neurons in these algorithms leverage decays (the "leaky" aspect), whereas other neurons are strictly integrate-and-fire, with total decay between time-steps. All the synapses described in the following sections are point synapses, without any intrinsic dynamics.

When considering very simple synapses and rather simple neurons, it is perhaps easier to consider these neuromorphic platforms as very large-scale parallel machines, with each "neuron" representing a very simple computational core that is capable of only a very limited number of operations (basically, integrating over a set of inputs and a thresholded output). Once this logical machine is considered, the challenge of algorithm design is to construct a neuron circuit, or a neuron graph, that produces the desired computation.

While the prospect of constructing an otherwise serial algorithm as a set of neurons with hand-crafted connections between them is not a common practice, we have begun to develop a series of algorithms with this approach. These include algorithms for cross-correlation [27], optimization [32], sorting and related functions like max / min [31], and matrix multiplication [24]. In many respects, these algorithms have considerable similarity to the extensive work from the threshold gate community in the 1980s and 1990s [28]; but in other ways the spike-based logic that is described here often depends on leveraging the temporal dynamics conferred by a system of neurons, even if their local activity has limited dynamics.

3. OVERVIEW OF RANDOM WALKS AND DIFFUSION

The classic random walk, a stochastic process, underlies many numerical computational tasks. The random walk is a direct reflection of the underlying physical process and models Brownian Motion, among other processes. Random walks have found myriad applications across a range of scientific disciplines including computer science, mathematics, physics, operations research, and economics[20]. For instance, the treatment of ionic movements as a random walk process is critical to deriving Nernst-Plank dynamics for ions in understanding the biophysics of neurons [19]. Additionally, random walks are also used in non-physics domains, such as financial option pricing [6, 35] and ecology [9].

Random walks are typically straightforward to implement, and can be computationally appealing in high dimensional domains that are ill-suited for other numerical approaches. Because they are typically used to independently sample a population, simulations of many random processes are easily distributed across a parallel machine; with each computational core responsible for a distinct process. However, the utility of multi-core systems for multi-agent models such as random walks is still limited in many applications [8]. Most simulations that utilize random walks to statistically arrive at a solution require the aggregation of a population of walkers before any conclusions can be made. Thus, while the walkers themselves are easily parallelized, the overall simulation is still constrained by the integration of information across the population.

3.1. Random Walk Model

Consider a system, S, that consists of a mesh of discrete locations. For simplicity we will consider the case where the mesh is a lattice of N grid points along each of D dimensions, although in practice a lattice is not a requirement. Within S is a population that evolves through a random walk process that is suitable to model as a population of independent particles, such as a diffusion process where each particle moves through space according to a Brownian motion evolution. We consider only the case where each particle is independent without interactions.

If a simulation models K independent particles, then the average position of the K particles approaches the expected value of the population at a rate of $O(K^{-0.5})$ as a consequence of the central limit theorem.

3.2. Rationale for Mapping Random Walks to Neuromorphic Hardware

Neuromorphic hardware presents a compelling architecture to consider the implementation of random processes. In the ideal, a neuromorphic platform can be viewed as an incredibly large parallel architecture, albeit one with very simple processors (i.e., the neuron) [27]. In particular, we hypothesize that neuromorphic platforms that leverage spiking neurons, such as the LIF neuron, and have inherent capability for probabilistic sampling, such as either stochastic synapses or probabilistic thresholds, may offer compelling advantages for modeling a random walk process.

This paper describes two spiking neural circuits for simulating random walkers. We then analyze these models in the context of emerging neuromorphic computing

architectures, such as the Intel Loihi chip [11] and the ARM-core based Manchester SpiNNaker platform [14]. We note that the approach taken here for modeling stochastic processes relies on relatively small circuits with very precise use of stochastic events, whereas an alternative approach to modeling stochastic inference consists of more dynamical population models of neurons [7].

3.3. Two perspectives of random walks

The following two chapters of this report describe spiking neural algorithms for simulating the random walk process. Notably, they perform this simulation on a diff.

- The *particle algorithm* (Section 4) simulates the random walk process by committing a population of neurons to each walker to represent location (and any other state variables), with potentially no neuron connectivity between walkers. This approach is embarrassingly parallel over walkers, and it is conceptually the same with the Monte Carlo simulations of Markov random walk processes common on conventional platforms, including HPC simulations (**Figure 3-1**).

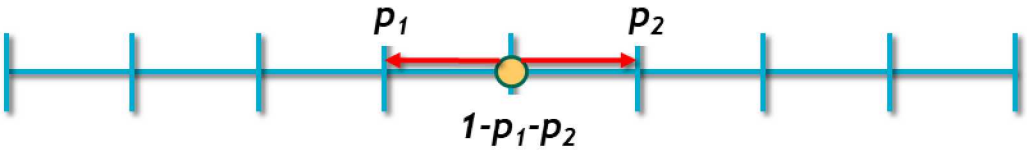

Figure 3-1: Walker-centric basis for particle algorithm

Because this approach is conceptually the same as conventional random walk simulation approaches, it also comes with comparable considerations and overhead. Most notably, while the behavior of any given walker has some value, particularly for path-dependent computations, for most PDE applications what is desired is a population average or distribution over all of the walkers. As a result, while the simulation of these walkers is entirely parallelizable, the consolidation of information from all the walkers is an additional cost that must be considered. Further, because the walkers are all independent, the relationships between walkers (such as for interactions) is an additional cost, potentially a substantial one, that must be considered if interactions part of the model.

- The *density algorithm* (Section 5) uses a distinct basis for computing the random walk. Rather than committing a population of neurons to each walker to represent location, the density algorithm rather commits a population of walkers to each spatial location, such as a vertex in a mesh. This population of neurons then can represent the probability density of particles at each location.

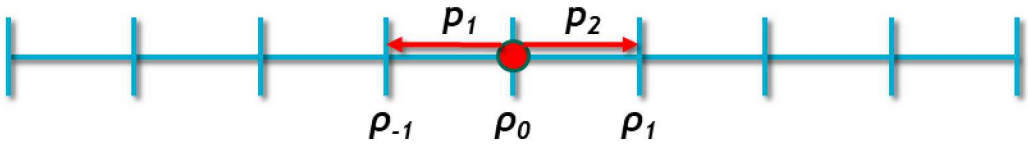

Figure 3-2: Location-centric basis for density algorithm

The crux of the density method's simulation is that at each time step, walkers are moved from one location to another. For instance, in **Figure 3-2**, each walker at location 0 has a probability of moving left (p_1) or right (p_2) at a given time step. Upon the time-step initiation, the totality of walkers at location 0 (ρ_0) are then sent to its neighbors according to those probabilities, and the neighbors likewise send walkers back according to their probabilities.

This approach is mathematically the same as the particle-based approach, but it has different advantages and disadvantages. Most notably, the algorithm is no longer embarrassingly parallel – it can only operate at the speed of the location with them most walkers. However, it confers several key benefits. First, an approximation of the local density of the walkers are what is represented explicitly at every time-step; this is convenient as this is often the relevant measure necessary for PDEs. Second, the locations being modeled do not need to be Euclidean or even spatial – rather they can comprise any graphical structure that has weighted probabilities of transitions. Finally, boundary conditions, such as absorption or reflection conditions, can often be directly instantiated in the graph itself, and identifying which particles may interact is potentially encoded implicitly in the densities as well.

To illustrate that these two methods are effectively equivalent in approximating the diffusion process, we implemented a simple random walk simulation in MATLAB, whereby 2000 random walk processes starting at the origin were independently simulated over the range [-1, 1] for 200 timesteps (**Figure 3-4**). As seen in the figure, the mean of the population does not change, while the walkers begin to diverge considerably from their stating location.

Figure 3-3 shows what the local densities of these particles look like over time for different bins in space, and is what the output of the density method simulation of this would produce directly. In cases where this histogram representation of the probability density is the desired output of a simulation, the density method directly provides an approximation of that over time.

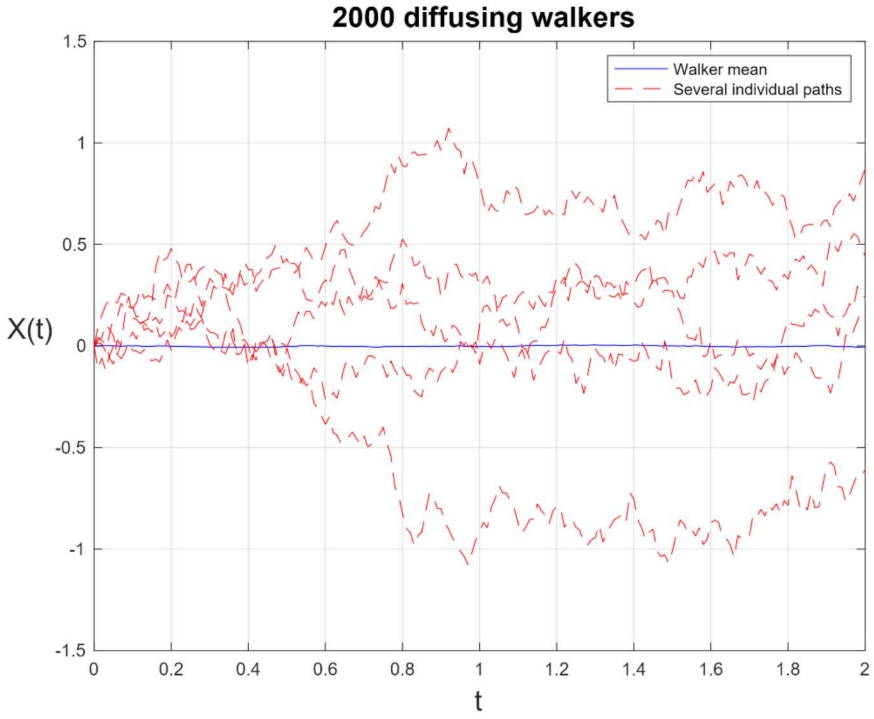

Figure 3-4: Example walkers of 2000 walker Monte Carlo simulation.

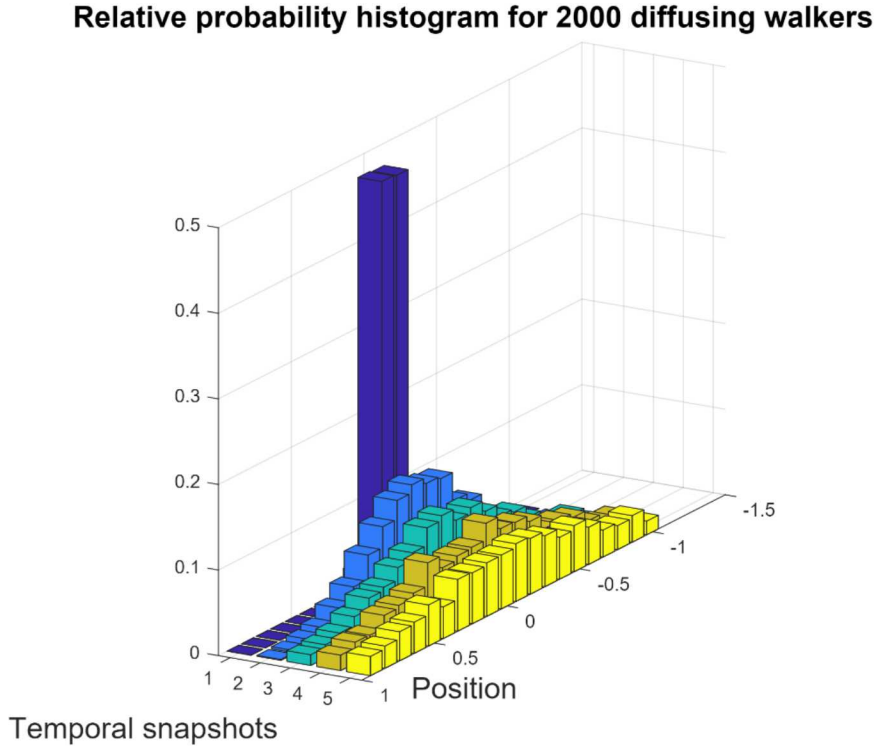

Figure 3-3: Approximated density derived from 2000 walker simulation

4. PARTICLE METHOD NEURAL ALGORITHM

The most straightforward approach to modeling a random walk is to commit a subset of neurons to modeling each particle independently. A simple neural algorithm for a particle consists of three parts: the *stochastic process*, which determines what random action is taken, a *spatial location*, which tracks the location, and an *action circuit*, which updates the location based on the output of the stochastic process and any boundary conditions, if relevant.

In most implementations, the dominating neuron cost for simulating individual walkers will be the spatial location. Even if particles are relatively restricted in their local movements, each particle circuit must be able to represent all spatial locations that are relevant for the simulation. Thus, if space (i.e., number of neurons) is the primary consideration, a compact code, such as a binary representation is well suited, as it requires only $O(D \log N)$ neurons to represent space. However, a binary code is non-trivial to update using neurons, and the average activity of the network is relatively dense. Alternatively, a unary code --- where one neuron represents each spatial location --- can be highly energy efficient (only one spike required to communicate location) and straightforward to update, albeit spatially impractical (requires $O(N^D)$ neurons to represent space).

Here, we present a neural algorithm that lies between these extremes, offering a compromise between a binary and unary representation of space.

4.1. Algorithm Description

One potential model that lies between unary and binary is a modular code, also known as a residue numeral system. Our approach to implementing a modular code is shown in **Figure 4-1**. This model is inspired by a model for grid cells in the entorhinal cortex brain, which has been shown to have very high capacity for spatial locations relative to the more unary-like place cells in the hippocampus [29].

For each dimension, the particle circuit will have M ring oscillators, each with a unique prime number of neurons, C_i for $i \leq M$, with states at time t, $c_i(t)$ for $i \leq M$ with the combined state represented by the vector $C(t)=[c_1(t), c_2(t), ..., c_M(t)]$, where each state is the integer index of which neuron is active in each ring. This provides the circuit with $C_M=\Pi(C_i)$ possible states. For example, consider a particle with $M=3$ and $C_1=3$, $C_2=5$, and $C_3=7$, then the particle's spatial code would have $C_M=105$ possible states.

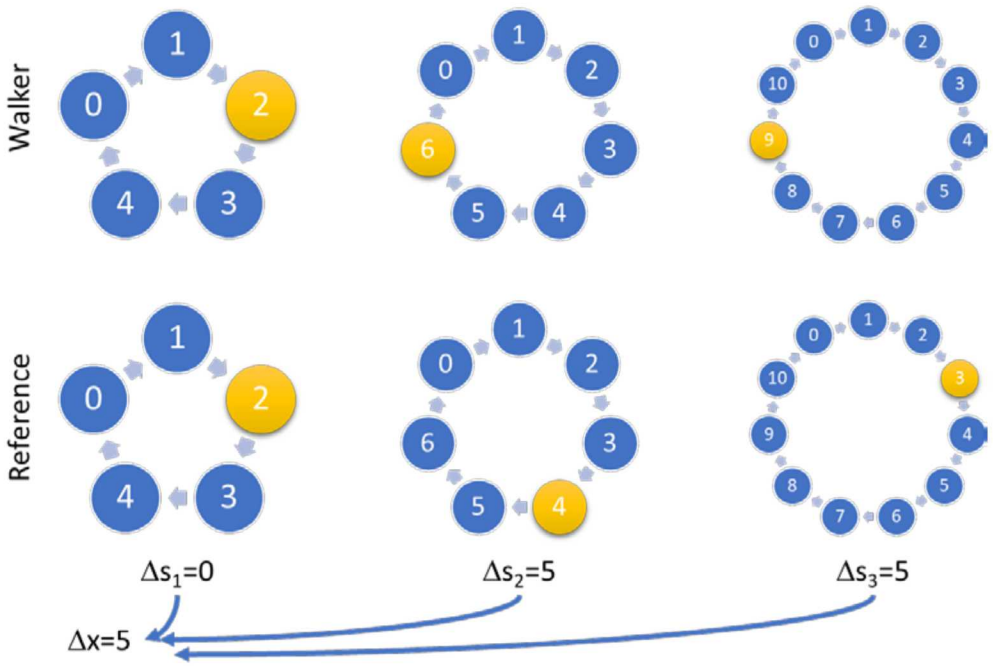

Figure 4-1: Modular position circuit. Each walker consists of M rings of different lenghts (lengths are co-prime). At each time step, each ring advances by one unless there is a particle movement. The position of the walker can be computed by computing the difference of the ring positions of the walker and a reference walker (i.e., set at the origin). The set of differences forms a modular code; in this case [0,5,5] out of a [5, 7, 11] modular basis. This code would represent a shift of +5.

To implement the random walk in neurons, we consider the case where a position x is encoded by the offset between the particle's state vector C and an equivalently sized reference population, R, which consists of rings of the same size. At each time-step, for the state of each ring oscillator in the reference and particles advances by one,

$$c_i(t+1) = \begin{cases} c_i(t)+1, & if\ c_i < C_i \\ 0, & otherwise \end{cases}$$

The position, x is then generated from C and R by subtracting the two states. For each oscillator, a difference

$$\delta_i = (c_i - r_i)$$

is computed, from which we know, by the Chinese Remainder Theorem, that the position, x, can be decoded. (One useful reference may be pages 873-876 in [10].) One extension of residual codes such as these is that addition and multiplication involving x can be performed by the equivalent modular arithmetic operation on each of the component rings. Therefore, a change of Δx in the position of a walker can be represented by adding Δx to each of the states $c_i(t)$.

Structuring a neural circuit to advance a ring oscillator continuously is straightforward, with each ring of C_i size being comprised of C_i LIF neurons (see Section 2.2.1)connected in a ring configuration, with the synaptic efficacy being sufficient to drive the downstream neuron to fire. However, a non-obvious circuit is necessary to reliably speed up or slow down the oscillators if the random walk moves the location. **Figure 4-2** shows one circuit solution that uses spike delays to `add' and `subtract' to the position of the ring by one using spike delays. The integrate-and-fire neurons in this circuit all have a spiking threshold of *1*, a reset value of *0*, and immediate decay (i.e., a time constant of *0*). In this implementation, a ring neuron at location *i* is connected to the neuron at *i+1* with a weight *1*, and to the neuron at *i+2* and to itself with weight *0.5*. Each of these ring connections has a delay of *2*. With this setup, without any other inputs the ring will advance by one state every *2* clock cycles.

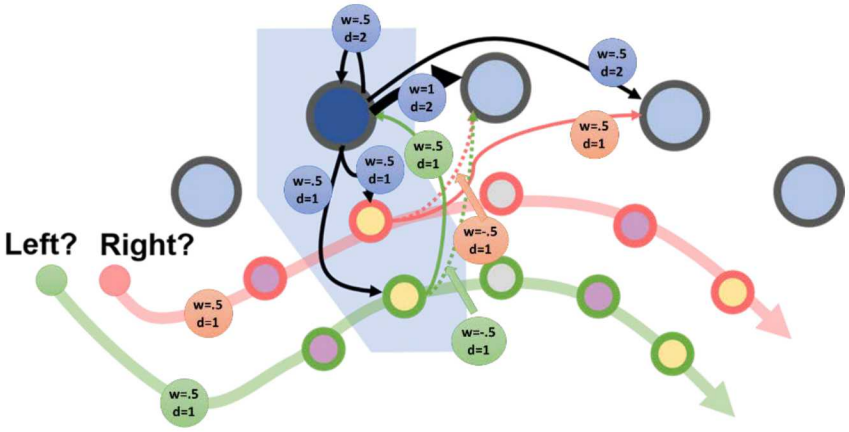

Figure 4-2: Update circuit for neural rings. At least 3 (up to 5) pairs of update neurons are required within each ring oscillator. For each ring neuron, a positive shift and a negative shift neuron are associated with it. The activation of a given update neuron (through coincident activation of the assigned ring neuron and an update 'left' or 'right' signal) then will bias the ring's activation forward or back by one neuron according to the circuit shown here.

A secondary circuit is then placed on all rings of a walker to advance or stall the circuit (thus generating an offset relative to the reference). We consider here the case where the particle has three potential movements (`left', `right', or `stay'); with a source neuron for each direction using a stochastic threshold or synapse to determine whether to move in one direction or not and communicating that action to each of the particle's rings for that dimension. The currently active ring neuron, *i*, sends an input of weight *0.5* and delay *1* to both of its respective update neurons. In the case where the circuit is advanced (labeled `right' in **Figure 4-2**), all positive update neurons get a *0.5* input as well, allowing the appropriate positive update neuron to fire. That neuron then sends a *+0.5* to the *i+2* ring neuron and a *-0.5* to the *i+1* ring neuron, effectively shifting the ring forward by *1*. The negative update is similar, except for driving the source *i* neuron rather than the *i+2* neuron.

Importantly, because the rings are only locally activated and impact up to two ring neurons away, these update neurons can be reused every three ring neurons. Ultimately, this means either four or five pairs of update neurons are required, because there are a prime number of ring neurons.

The dynamical representation of position as the offset of these oscillators confers several advantages. First, it is consistent with the transient state of neurons. Rather than a neuron having to self-activate to maintain a state, the ring simply evolves at a steady rate when position is not changing. Second, it allows updates to be more efficiently implemented. When there is a random movement of the particle, in whatever dimension is being considered, the particle's rings are in unison accelerated or decelerated by one. The use of a common reference for all particles also allows changes in the frame-of-reference to be efficiently accounted for as well – a simple shift in the reference state is the equivalent of shifting all the particles in unison. This may be of use in models where an observer of a random walk is itself in motion. Similarly, because the reference is used only in the decoding of position, it is possible to have multiple references, or to readily compute the distance between particles without using a reference at all.

4.2. Theoretical Assessment

Each walker for the above model requires $2+\Sigma(C_i+2*(3+C_i \% 3))$ neurons and $\Sigma(9*C_i+2*(3+ C_i \% 3))$ synapses. Only one spike is required per ring, for M total, when there are no updates, and $M+1$ additional spikes required for an update.

There is a global cost as well, with an additional set of rings for the reference position (although unless the reference position is also in motion, update neurons would not be required). Each dimension would consist of its own rings.

This model presents a useful trade-off between a dense code, with lots of rings, and a sparse code, which is more energy efficient but requires more neurons to cover a space. The dense code would approach $O(D \log N)$ total neurons, with systems with fewer rings approaching $O(DxN)$ total neurons and with a correspondingly lower number of spikes. Table 4-1 highlights the complexity of particle walkers in the formulation described here.

Table 4-1: Theoretical Complexity of Particle Algorithm

Measure	Cost (for k locations; 1-D case)
Position memory per walker	O(k^(1/N)), where N is # rings of length s_n, s.t. $\Pi_N(s_n)>k$
Connection memory per walker	Σ_N (9*s_n+2*(3+s_n mod 3))
Total neurons per walker	2+Σ_N (s_n+2*(3+s_n mod 3))
Time per physical timestep	2
Position energy per timestep	O(N)
Update energy per timestep	O(N)

Table 4-2 provides specific examples of the required neuron and activity costs of three small random walk configurations using the algorithm described above. For instance,

Table 4-2: Example Scaling of Particle Algorithms

Measure	k = 1000, 1D	k = 1000, 2D	k=10000, 1D
Modular set	{7, 11, 13}	{7, 11, 13} x2	{17, 23, 29}
Position memory per walker	1001	1002001	11339
Connection memory per walker	293	583	639
Total neurons per walker	59	118	101
Time per physical timestep	2	2	2
Position energy per timestep	3	6	3
Update energy per timestep	3	3	3

4.3. Simulation Results

First, we demonstrate the particle method by showing a random walk in free space. **Figure 4-3**A illustrates the appropriate random trajectories of the particles over 100 time steps. **Figure 4-3**B shows a longer time course, with particles moving for 1000 time steps.

One key limitation of the modular method described above is its behavior when a walker's position exceeds the precision of the neural circuit. Because the modular code described above has a finite capacity, eventually particles in a free space will drift beyond the provided spatial resolution, wrapping around the space as if it is a torus. An example of this is shown in **Figure 4-3**C, wherein a small modular code (rings of size 3 and 7) led to a perceived jump of the particle from position *-10* to *+10*.

Next, we illustrate how more complex walks can be examined, such as non-uniform probabilities. **Figure 4-3**D shows a case where random movement is biased heavily in the negative direction, with the movement of the particles drifting as a population towards the bottom left.

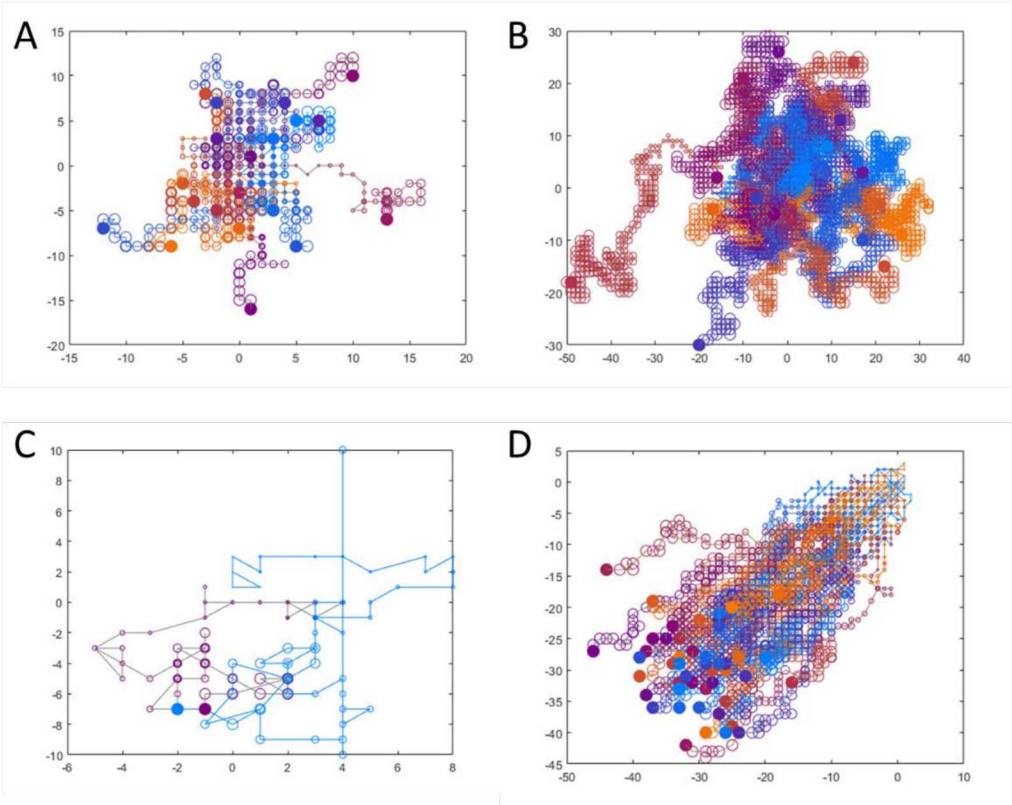

Figure 4-3: (A) Random walk of 20 particles over 100 time steps in free space originating at the origin, with a balanced probability of moving in either dimension equal to 0.25. Each particle used rings of size 5, 7, and 11 neurons. Steps are represented by progressively larger circles, with the solid dot representing the end location. (B) Random walk of 20 particles over 1000 time steps in free space originating at the origin, with a balanced probability of moving in either dimension equal to 0.25. Each particle used rings of size 5, 7, and 11 neurons. (C) Random walk of 2 particles over 50 time steps in free space originating at the origin, with a balanced probability of moving in either dimension equal to 0.25. Each particle used rings of size 3 and 7. Note the misencoding of the blue particle due to reaching the capacity of the code. (D) Random walk of 50 particles over 200 time steps in free space originating at the origin, with a weighted probability (of 20%) of moving in the negative direction in each dimension, versus 5% of moving in the positive direction. Each particle used rings of size 5, 7, and 11 neurons.

4.4. Boundary Condition Implications

The particle method described here was designed to be a proof-of-principle implementation of efficient updating and encoding of walker location in a finite number of neurons. Real-world applications require two additional features to the model. For most applications, such as radiation transport or molecular dynamics, the model will have to account not only for spatial location, but also additional state variables. For this algorithm, additional state variables are rather trivial – they are just another set of rings that are either deterministically or stochastically updated according

to whatever model is being implemented. The number and sizes of rings required would be a function of the necessary precision of that state variable, and would not necessarily need to equal what is used for spatial resolution.

In this current structure, boundary conditions were not an emphasis. Because position is encoded as the difference between the ring states and that of a reference, a decoder circuit would be required in this implementation to check for boundary conditions; and

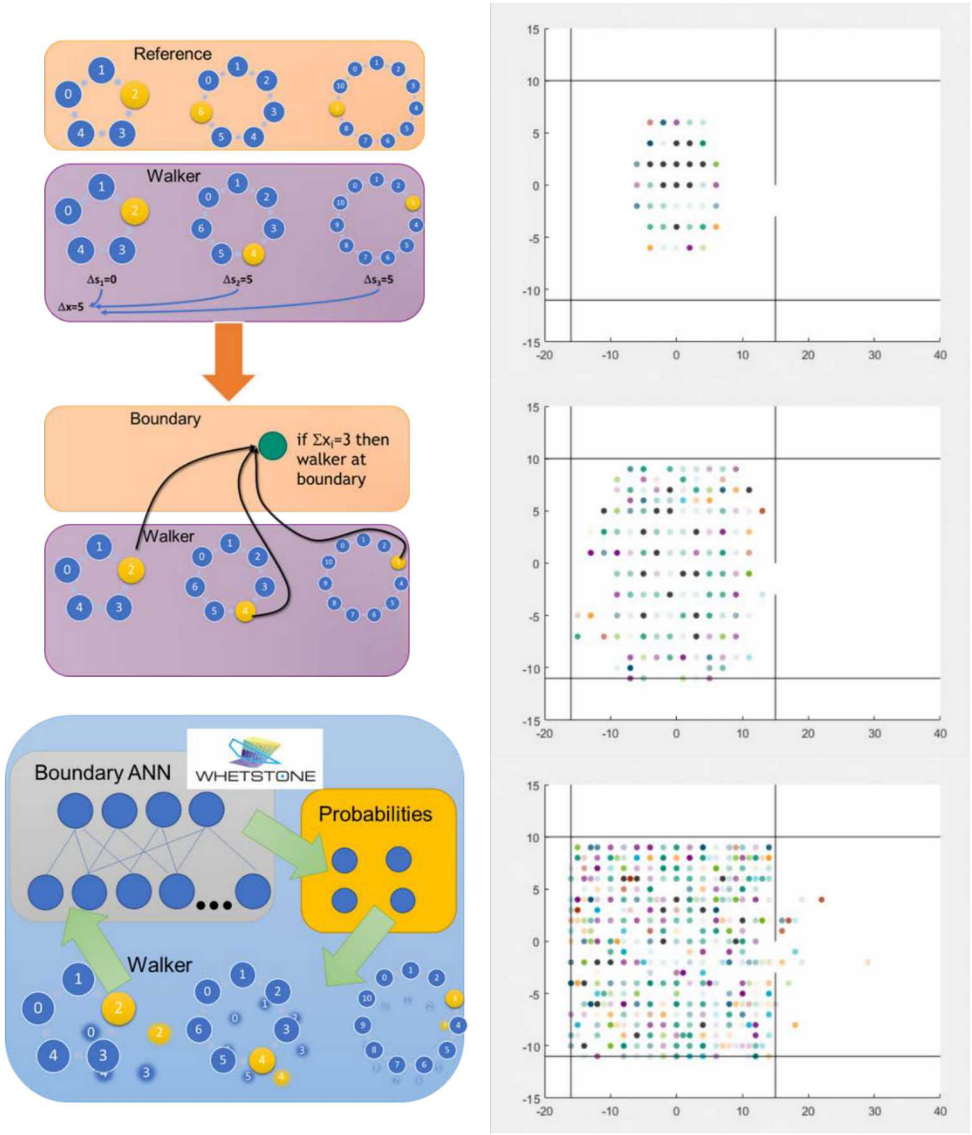

Figure 4-4: Illustration of boundary conditions in particle method. Transforming the rings to static unless moved enables a new set of neurons to "check" if a boundary has been hit; which in turn biases probabilities of moving. Right - illustration of particles released in a box with a single hole to escape. Boundaries represented by a check neuron to block movement at y=10, y=-11, x=15, x=-15, and a subset of neurons to undo the block at x=15, y=[-3:0]

that circuit would presumably have to scale with the complexity of boundaries being checked.

However, an alternative approach is readily available. An equivalent algorithm to the constantly evolving rings described here is one that steps forward or backwards along the rings only if the walker is moved (thus the active neuron does not change on a static walker). This may introduce complications related to neuron load balancing – certain neurons will be active more than others – but it does eliminate the need to check for a reference. In this case, each walker could get a set of neurons that check for whether the current state of the rings equals a pre-determined boundary check (i.e., does the x-position equal a boundary x-position), and if so, then change the behavior of the particle accordingly. As shown in **Figure 4-4**, these boundary check neurons can actually be an artificial neural network trained to represent the boundaries.

4.5. SpiNNaker Results

To examine the hardware suitability of the particle algorithm, we implemented it on the 48-chip SpiNNaker platform housed in the Neural Exploration Research Laboratory at Sandia. SpiNNaker provided an interesting challenge because although at its core it is a highly parallel neuron simulator, its programming stack is optimized for biological neural simulations using a neural simulation description tool known as PyNN. PyNN, and its associated SpiNNaker interface known as sPyNNaker, presented a few interesting research challenges that led to rethinking some aspects of the particle algorithm.

Our initial programming of SpiNNaker was limited because the default synapse types available on the platform have synaptic decays, but at the close of the project we have developed a PyNN neuron part that does allow point synapses. Point synapses are particularly critical in the timing of the particle algorithm, since anything but absolute resetting of synapses between time-points will cause the network timing to drift. We were able to avoid much of this on SpiNNaker through two changes to the algorithm. First, we set all decays between neurons to 5ms or 10ms (as opposed to 1 ms or 2 ms), which slowed down the computation considerably but allowed sufficient decay to effectively reset the synapses. Second, we added a special population of neurons that offset the problems associated with stochastic source neurons firing two time-steps in a row, which, when combined with incomplete decay, would disrupt the bias neurons. We anticipate that both of these alterations – that reduce the efficiency of SpiNNaker – to be unnecessary in the future due to our new dynamics-free synapse type.

An additional challenge was that the sPyNNaker programming process prefers defining a small subset of neuronal populations with common dynamical parameters, and then it distributes those populations over the available cores. Because of SpiNNaker's configuration, it is generally preferred to have only one neuron type per core, and no more than 250 neurons per core. Our implementation on SpiNNaker initially required 3 populations – one population for the stochastic inputs, the aforementioned population to "separate" the stochastic inputs, and one very large population of all of the ring and update neurons. This skewed distribution led to a

somewhat challenged graph embedding problem. Simply stated, the compiling of neurons across walkers eliminates the embarrassingly parallel structure of the network model, but keeping the walker neurons separated underutilizes cores due to the small number of stochastic neurons per walker.

In theory, if we assume that each walker requires less than 250 neurons; adding the *K+1* walker to the existing *K* on the board, so long as we are under capacity, should be trivial, since there is no circuitry between the walkers. However, this is not the case as shown by **Figure 4-5**. While loading around 100-200 walkers is quick, any loading beyond that rapidly becomes slower, reaching prohibitive levels (~1 hour to perform the network embedding on SpiNNaker) when reaching about 10% of the board's capacity.

In addition to these compilation time challenges, we have encountered other challenges in maximally leveraging SpiNNaker's capacity, suggesting that a different approach to structuring the models should be considered in the future.

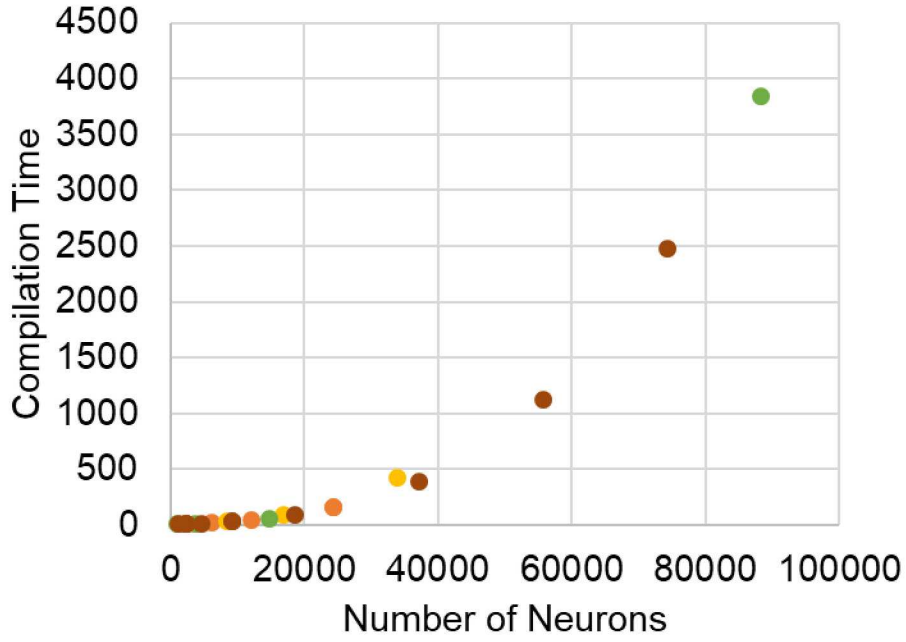

Figure 4-5: Scaling of compilation time, in seconds, on Spinnaker due to the number of neurons. The different colors relate to the number of neurons per walker, which does not appear to affect the compilation time.

These challenges aside, the SpiNNaker experiment was successful, showing that we could indeed implement the particle method on the SpiNNaker platform and simulate for extended periods of time. As shown in **Figure 4-6**, SpiNNaker's random walk simulation performs as expected, with particles diffusing through the expected random walk process according to the built-in biases on the network. Once compiled onto the board, the simulations ran in constant time regardless of how many walkers were being simulated, albeit relatively slowly due to the throttled back speed of SpiNNaker and the aforementioned extended delays to offset the synapse dynamics.

In conclusion, the use of SpiNNaker for prototyping these random walk models is a reasonable course of action, because it does impose rigorous adherence to the requirements of a neuromorphic platform, and it provides a relatively straightforward programming interface. In particular, now that some of the limitations associated with sPyNNaker's assumption of biophysically relevant dynamics have been bypassed, it is likely that SpiNNaker will be more efficient from a testing point of view.

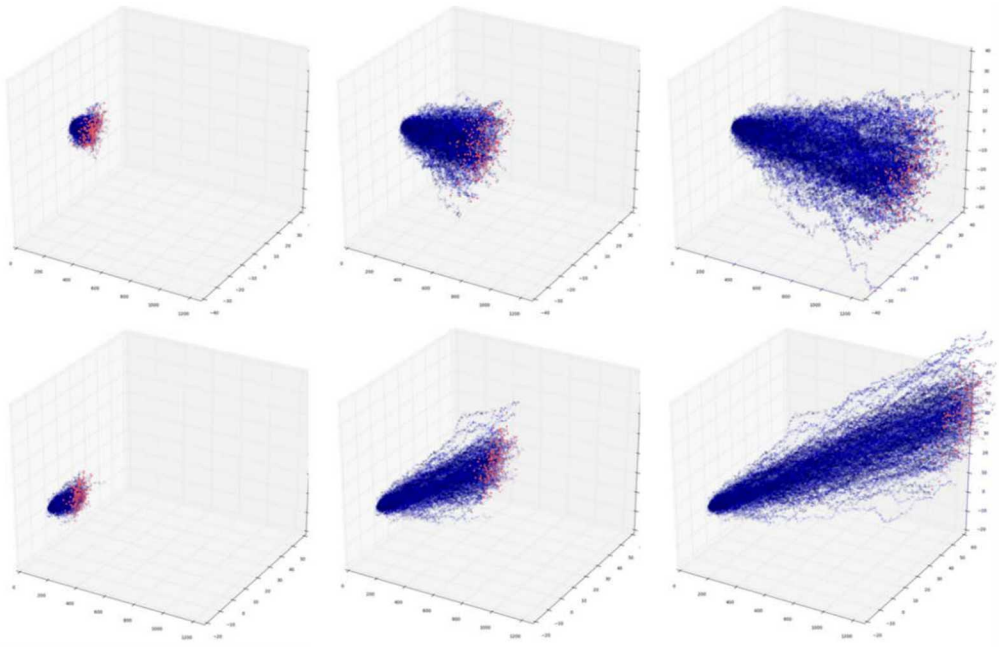

Figure 4-6: Illustration of random walks on SpiNNaker, 250 walkers, 1250 time steps, rings of size 11, 13, and 17 in both dimensions. Top is no bias, bottom is a significant bias to up and right.

That said, the relatively low neuron densities per chip will probably preclude application use of SpiNNaker as is today in real-world applications. Each chip has 16 cores committed to neuron simulations, which each handling about 250 neurons. Even if ignoring the graphical restrictions, at most this would allow about 1 core per walker for the 2-D simulations shown above. 16 walkers per chip is not a sufficient simulation paradigm, especially if the chip is running between kHz and MHz speed. Rather, the use case of SpiNNaker likely requires a significant change in either the architecture or the network mapping. For instance, if SpiNNaker 2.0, which is underdevelopment, indeed can provide an order of magnitude more neurons per core as has been suggested, it is possible that a chip may be able to run several hundred walkers at one time. Further, if those cores could be structured to take advantage of the fact that all communication would be local (as opposed to chip-to-chip), then the chip speed could potentially be increased considerably. Each chip running ~1,000 walkers at MHz speeds would then potentially be a viable substrate for real-world applications, though it remains to be seen what future SpiNNaker hardware specifications will be.

5. DENSITY METHOD NEURAL ALGORITHM

One alternative to tracking the particles independently is to keep track of the density of particles at every location and randomly move walkers. The main advantage of a particle density approach is that the complexity of the spatial graph, in terms of number of neurons, is independent of the number of walkers. While a density representation is the equivalent of the particle method in terms of producing estimated density distributions at different times, path dependent statistics are not readily available. Instead, they must be decoded from the timing of the spikes. This can impact some application, such as estimating the prices of certain path-dependent financial options. On the other hand, summary statistics, particularly the number of walkers at a location at a timestep, are inherent in the local activity of network. Hence, walker information does not need to be collected and synchronized for this data.

5.1. Algorithm Description

5.1.1. Circuit-Level Description

As in the particle model, we need to either discretize a continuous space or equivalently assume that the Markov process exists on a graph. For each node on the graph, we instantiate a spiking circuit which we call a *unit*. A schematic of a two-neighbor unit is pictured in **Figure 5-1**. Within a component there exists several key components:

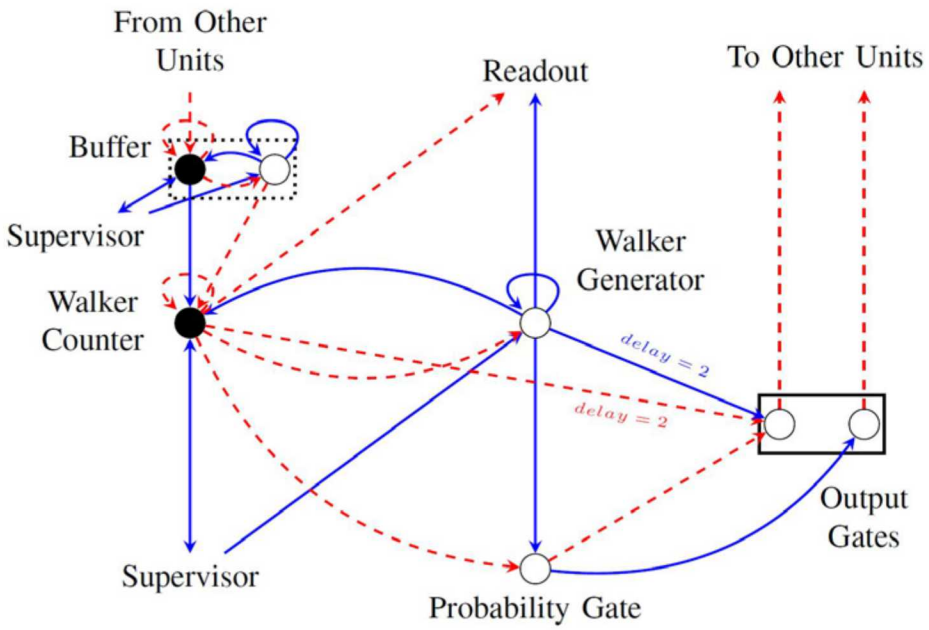

Figure 5-1: Diagram of density algorithm.

1. *Walker Counter* - The walker counter is a simple neuron with threshold *0* and contains running count of the number of walkers at a given node. Walkers are passed from unit to unit by spikes with negative weight (inhibitory signal). Hence,

a sub-threshold potential of *-5* corresponds with *5* walkers being at the corresponding node.

2. *Walker Generator* - The walker generator is a self-excitatory neuron that `counts' out the walkers stored in the walker counter. After being initiated by a separate supervisory signal, the walker counter sends positively weighted spikes to the walker counter, until the walker counter hits its threshold. At this point, all walkers have started their next transition and inhibition from the walker counter halts the walker generator.

3. *Probability Gate and Output Gate* - This group of neurons interacts with the output gates to ensure that each walker is sent to exactly one other unit, weighted by the specified probabilities. More specifically, a tree of neurons subdivides (through selective excitation and inhibition) the potential outputs according to conditional probabilities. In **Figure 5-1**, the unit only has two neighbors and so only one neuron is needed for the random draw.

4. *Buffer* - The buffer is an optional component for synchronized operation. Without the buffer, the walkers may each take a different number of steps. By incorporating a buffer, the walkers are first stored in the counter, sent to buffers of neighbor units, and then flushed from the buffers into the counters. Structurally, the buffer contains a counter and generator neuron.

5.1.2. Temporal Description

In the density method, rather than tracking each individual walker's position at any given point in time, we can instead keep track of the nodes on the graph and count how many walkers are at each node at any given point in time. When we are running the spiking simulation, we embed a spiking circuit that is located at each node in the graph. An outline of a frame of the random walk simulation utilizing the density method is described below.

1. An injected current starts initial walkers at a given node in graph by sending signals to walker generator neurons and walker counter neurons

2. Walkers are distributed throughout the circuit at that node through excitatory signals from the walker generator neurons

3. When a signal is received by output gate neurons, their potentials are modified

4. If the output gate neuron's thresholds are met, the output gate neurons determine whether or not to spike based on a certain probability (discussed in next section)

5. If the neurons spike, a signal is sent from the output gate neurons in the current circuit to the determined walker generator neurons and walker counter neurons at the neighboring node's circuit

A simulation using this density model is performed as a series of manually or automatically triggered tasks. Initially, current injection is used to place walkers at the desired initial position. Then, walkers are counted and distributed by sending an

excitatory signal to the walker counter and walker generator. This automatically sends walkers to neighboring nodes via the probability gate and output gate. We connect a `walks complete' neuron downstream of the walker counters so that we know when all the walkers have been distributed. If the units use synchronization buffers, the buffers are cleared in the same way via an excitatory signal. Likewise, when the buffer is flushed, we use the resulting excitatory signal to trigger the next simulation timestep (i.e. the walkers take their next step). Referring to the terminology of the particle model, the stochastic process occurs within the probability gates, the spatial location is stored in the potentials of the walker counters, as each unit has a location, and the action circuit is a combination of the walker generator and the output gates.

5.1.3. Stochastic Neuron Requirements

For this construction, we assume that the underlying neuron model is capable of stochastic firing. That is, after a threshold potential is exceeded, the neuron spikes according to the draw of a random number. This stochastic model is representative of currently available neuromorphic hardware. However, with more advanced neuron models, such as one that supports stochastic synapses (i.e., spikes are sent to post-synaptic neurons according to independent random draws) could allow for simplified circuits.

Additionally, we note that the stochastic spiking mechanism can be induced by a several other stochastic components. For example, on TrueNorth, we utilize stochastic positive leak to produce a reliable stochastic signal. This signal is integrated by downstream neurons whose behavior is effectively a stochastic spike. Similar methods can be used for stochastic thresholds, delays, and dendrites with implementations dependent on target hardware.

5.2. Theoretical Assessment

This density-based approach allows for the neuron requirement to be tied only to the size of the underlying space/graph and not to the number of walkers. Overall, the neuron cost for a n-node graph is $O(n)$ assuming the number of neighbors for any node is much smaller than the total number of nodes. The runtime is dependent on the number and distribution of walkers. The time taken to evaluate one simulation timestep is asymptotically linearly proportional to the largest number of walkers at a node.

5.3. Results

To examine the density model, we first explored a one-dimensional space where nodes are connected in a cycle, with transitions to adjacent nodes having a *50%* probability. Pictured in **Figure 5-2** is the distribution of walkers with units *10* and *13* being initialized with *30* walkers each. Only the Markov simulation time is shown. The checkerboarding seen is a result of the fact that each node is connected only to two neighbor nodes, and any given walker must move to one of these two options.

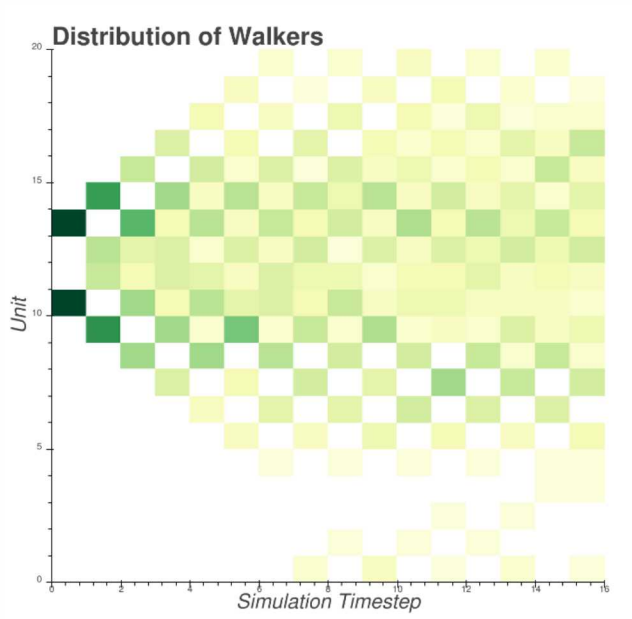

Figure 5-2: Walker distribution over time for a one-dimensional random walk, with 50% probability in both up and down directions. Walkers begin on units 10 and 13. Units are arranged spatially in a cycle so that walkers can `wrap around.

The corresponding spike raster for the 1D case is shown in **Figure 5-3**. As expected, the walkers tend towards a uniform distribution. **Figure 5-3** also illustrates how time

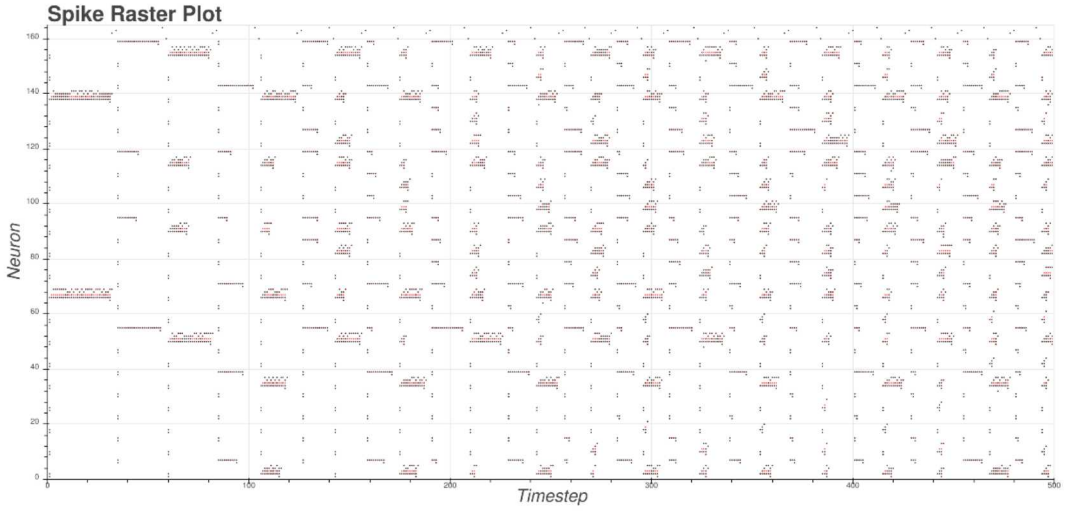

Figure 5-3: The spike raster plot for a one-dimensional random walk, with 50% probability in both up and down directions. Walkers begin on units 10 and 13. Black dots represent spike events; red dots represent failed probability calls (i.e., neuron threshold is met, but the neuron does not spike due to stochasticity). Neurons are grouped by unit, but are not sorted. Each simulation time step requires fewer computational time steps as walkers become more diffuse.

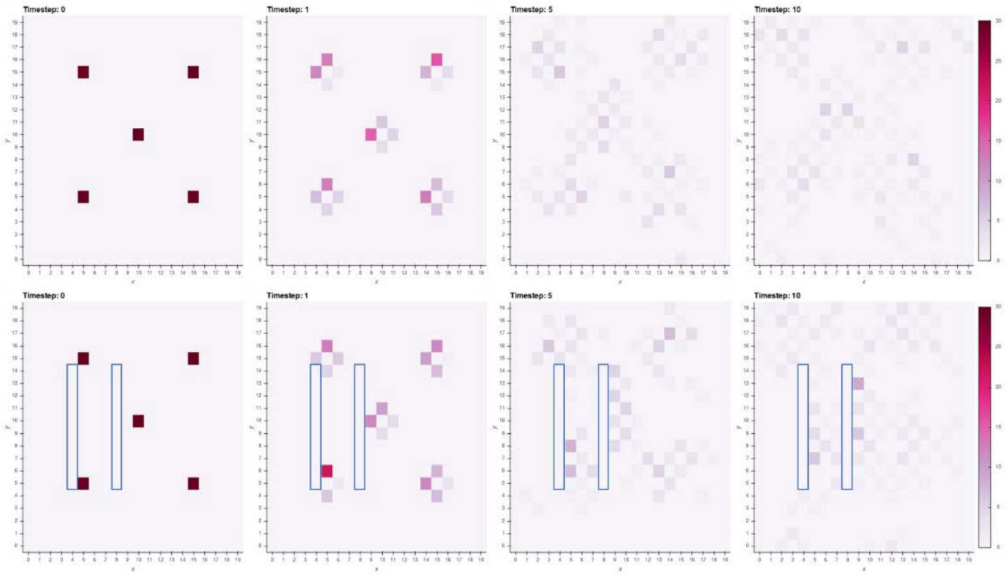

Figure 5-4: Sample walker distributions from two separate two-dimensional experiments. In the first row, 30 walkers start on units (5,5), (5,15), (10, 10), (15, 5), (15,15); directions up, down, left, right have probabilities 35%, 35%, 15%, 15% respectively. In the second row, the simulation has the same setup, except there are two obstacles (highlighted in blue). The walkers have 0 probability to enter the highlighted areas; the probability to enter a wall is distributed evenly to the perpendicular directions.

is treated differently the density method. The amount of simulation time required to model one-time step of the system evolution is non-deterministic, requiring enough time to potentially move all particles within any given location. Because the system progresses forward synchronously for spatial locations, the model time required to advance is dependent on the distribution of walkers. Because we are modeling a random walk, most simulations will drift towards more diffuse distributions, requiring progressively less time to simulate the model.

Figure 5-4 illustrates two time-courses of the density model in two dimensions on a torus. The top half shows an instance where the probabilities are uniform across the space (though weighted towards the up and left directions). The bottom half explores a case where walkers are prevented from entering two disjoint obstacles. In **Figure 5-5**, we plot the walker density at three different locations.

5.4. Boundary Condition Implications

In contrast to the particle method described above, spatial boundary conditions are readily implemented within the density method. As each neural circuit includes representations of the probabilities associated with its spatial location, the inclusion of location dependent behaviors – such as a wall or absorption condition – are straightforward. For most of these simple boundaries, configuring the graph should be sufficient.

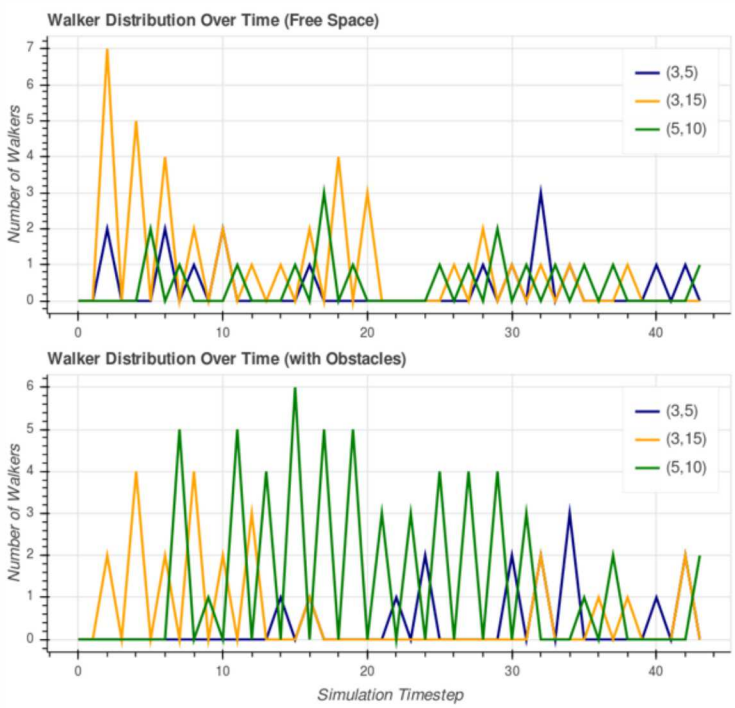

Figure 5-5: Plotted are the number of walkers over time at three different locations. The top graph corresponds to the top row of Figure 5-4, as the bottom graph does with the bottom row. The evolution of the walker distribution affected by the additional obstacles.

However, while spatial boundary conditions are straightforward, the implicit representation of walkers within the activity of neurons in each spatial location makes carrying additional state variables along with each walker non-trivial. One obvious technique would be to simply expand the mesh to account for the extra dimensionality conferred by state variables. In cases where the precision of these parameters is rather low, this may work; however, in cases where the additional states for each walk are high an alternative technique may need to be considered. The consideration of higher dimensional models will be a subject of future study.

5.5. TrueNorth Results

IBM's TrueNorth neuromorphic architecture [2] consists of 2^{20} neurons on a single chip. This 1 million neuron chip subdivides the neurons across 4096 cores with 256 neurons per core. Neuron connectivity is limited and creates a challenging effort to map unconstrained neural algorithms into this architecture. As a result, when implementing scalable algorithms there is always a tradeoff between efficient use of resources and mapping complexity. This complexity is manifested in the Corelet Programming Environment [4], a MATLAB programming tool set for defining the neural connectivity for TrueNorth implementations of any given neural algorithm.

Based on the algorithmic description covered in section 5.1, three implementations of the algorithm were created: a one-dimensional case composed of two directions, a

two-dimensional case composed of four directions, and a two-dimensional case composed of eight directions. The number of walkers placed at each node and the directional probabilities of each node are completely parametrized allowing full customization of each node in the defined topology. Each case implemented the buffer

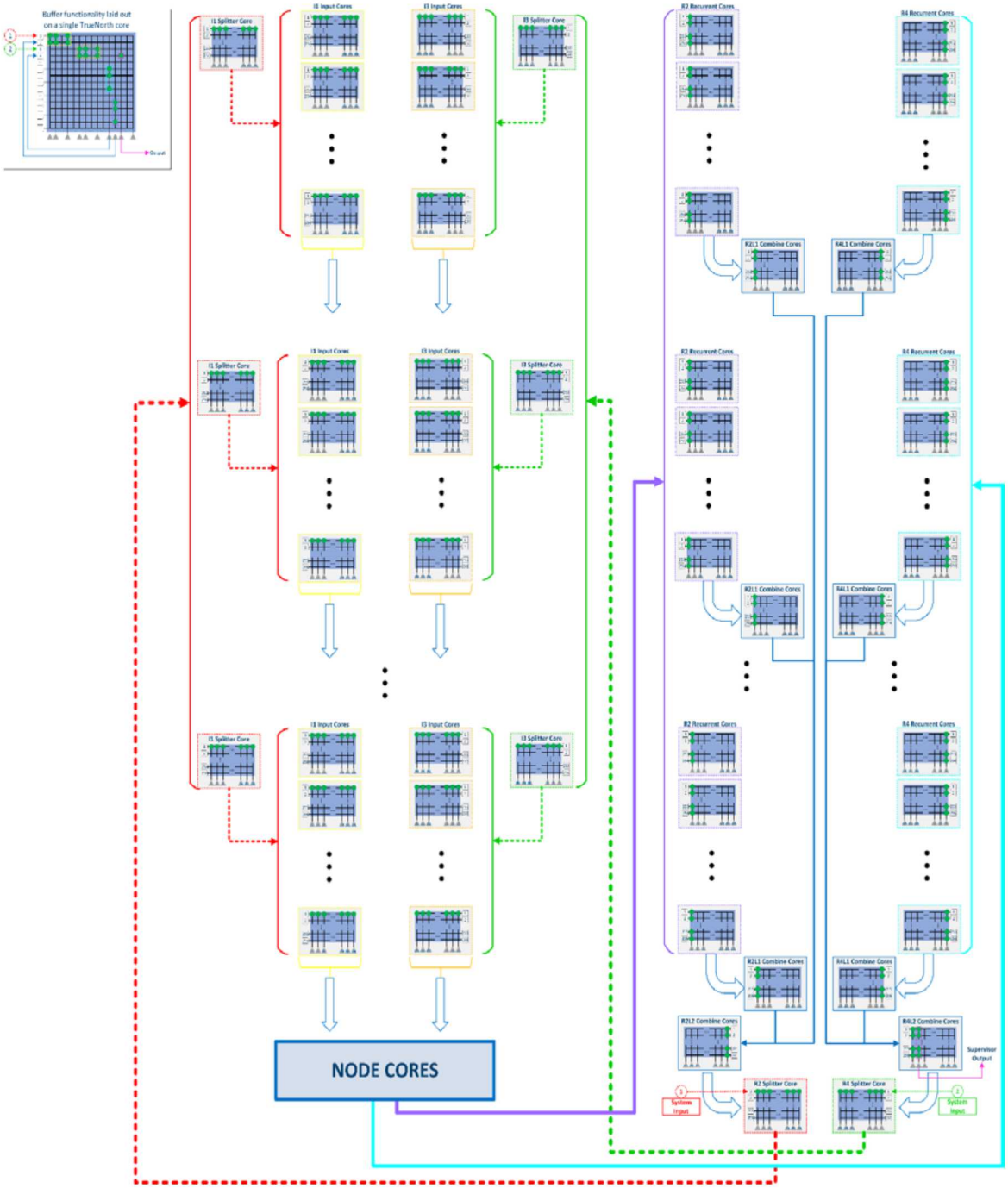

Figure 5-6: A scalable buffer functionality implementation in TrueNorth. Top left of figure indicates a base implementation of the buffer functionality mapped into a single TrueNorth core. Color coding of bounding lines indicate how each functional piece of the buffer is represented in the multi-core scaled implementation.

functionality to time-synchronize the walkers; all walker movements complete before another round of walker movements begins. Due to the limit connectivity nature of the TrueNorth architecture the buffer functionality must be split across multiple cores when scaling the algorithm to a very large number of nodes. As a result, the buffer function of the algorithm inherently has a large amount of wasted resources. See **Figure 5-6** for the structural layout of the buffer functionality on TrueNorth.

For simplicity of scalability, all nodes were confined within the limits of one core, though multiple cores are used to achieve higher node counts. For the 1D case, 13 neurons per node are used resulting in 19 nodes per core. For the 2D case with 4 directions, 21 neurons per node are used resulting in 12 nodes per core. For the 2D case with 8 directions, 37 neurons per node are used resulting in 6 nodes per core. Because the number of directions are tied to a binary tree structure, the number of directions increase by powers of 2. This scaling equation can be written as the number of neurons per node is equal to $5 + 4D$ where D is in the set of all powers of 2 that are greater than 1.

The 1D case is easy to visualize by looking at the spike raster plot output of the experiment. The location of spike outputs directly indicates the walkers' location in space. When each segment of time is laid out in series, an image is created detailing the walkers path. The image in **Figure 5-7** is somewhat of an illusion since the movement is occurring over a 1D space. Though, this image is equivalent to a 2D mesh of nodes in which walkers can move in only two directions, right-up or right-down.

The 2D case is a good representation of the tree structure expressed in a TrueNorth Corelet. **Figure 5-8** details the functional mapping of a random walker node on to a

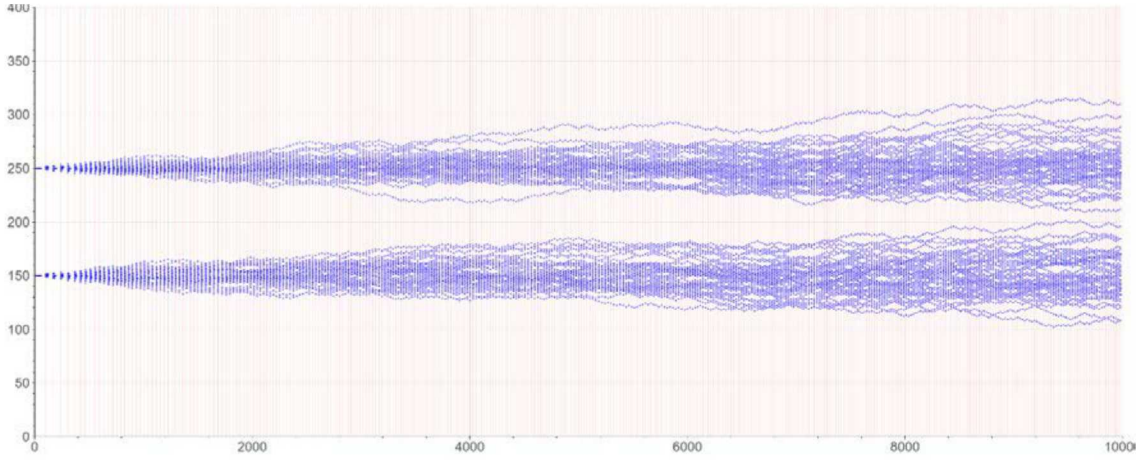

Figure 5-7: A 1D random walk example. The landscape is composed of 400 nodes with 50 walkers each starting at nodes 150 and 250. At every node, each walker has an equal probability of moving left or right. The trial is executed over 10,000 ticks or neural clock cycles where 1 tick takes 1ms to complete. Because all walker movement is synchronized by a buffer supervisor, this example consists of 378 buffer synchronization steps.

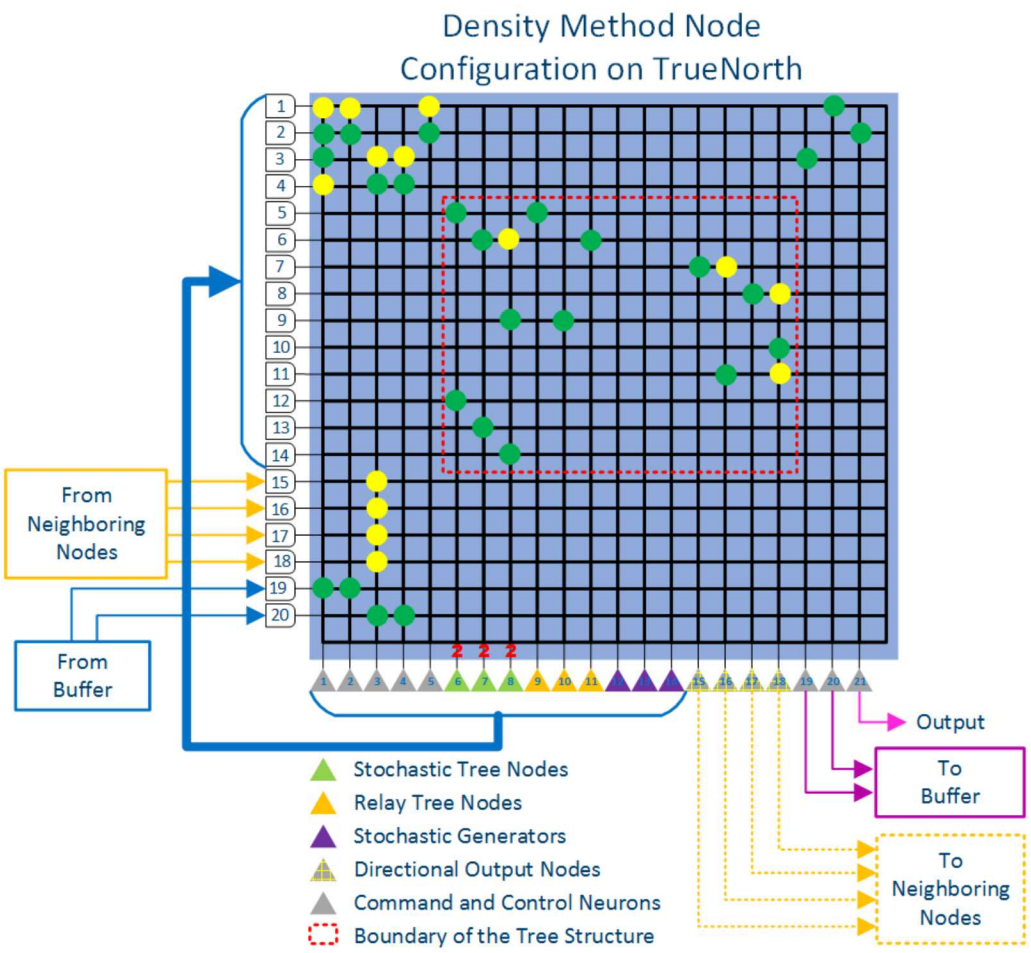

Figure 5-8: Graphical representation of a TrueNorth crossbar configuration implementing a node supporting the density method. Green circles represent excitatory connections and yellow circles represent inhibitory connections. All neuron thresholds are set at a value of 1, except in the case of the Stochastic Tree Nodes, which have a threshold value of 2. Neuron to axon connectivity for neuron/axon indexes of 1 – 14 are mapped one to one.

TrueNorth crossbar configuration. The neural connectivity of the tree structure is confined within the red dashed-line bounding box. All other connections support the interface with the buffer, neighboring nodes, and the host output.

Utilizing the node structure of **Figure 5-8** a 2D mesh was constructed consisting of 22,500 nodes in a 150 by 150 square landscape. Five initial walkers were placed at every node and unique probabilities defined based on the Sandia logo of the Thunder Bird shown in **Figure 5-9**. Probabilities were defined so that walkers had a greater likely-hood of crossing edge boundaries towards the black regions of the image of **Figure 5-9**. Each pixel of the image surrounded by the same color defined equal probability of moving in all directions. The experiment was run over 10,000 TrueNorth time steps. Eight frames of the time series were extracted to illustrate the evolution of the walker's movement in **Figure 5-10**. This example utilized 2,233

TrueNorth cores and produced a total of 22 million (22,005,161) output spikes in 10 seconds of run time. A similar example to this but utilizing an 800 x 800 image, or 640,000 nodes is expected to utilize 63,378 TrueNorth cores and be implemented on the 16 TrueNorth chip board hosted at Lawrence Livermore National Labs.

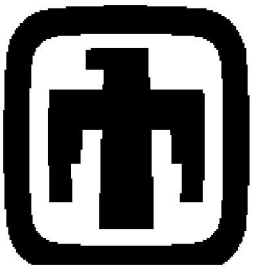

Figure 5-9: Sandia Thunderbird image used to define unique probabilities for each node.

In the algorithmic description of the density method it is defined that a tree structure is utilized to subdivide the potential outputs according to conditional probabilities. This structure creates a $log(d)$, where d is the number of directions, time delay from initial excitation spike to the node and the output spike of the walker along a single direction. Additionally, it requires $d - 1$ stochastic neurons. Through implementation of this algorithm onto TrueNorth a different structure was discovered that is based on the binary encoding of directions. This approach produces a constant-time time delay of 2 and only requires $log(d)$ stochastic neurons. However, the tradeoff is that the probability definitions for each direction become more coupled and much harder to tune through direct manipulation of the stochastic neurons.

The binary encoding of direction works by treating each stochastic neuron as a bit in a binary string. Based on the combination of neural firings of the stochastic neurons a binary value is produced. Each unique binary value represents a direction. For example, two stochastic neurons will produce a 2-bit binary value and we can encode directions as follows: Up (0 0), Down (0 1), Left (1 0), and Right (1 1). The TrueNorth

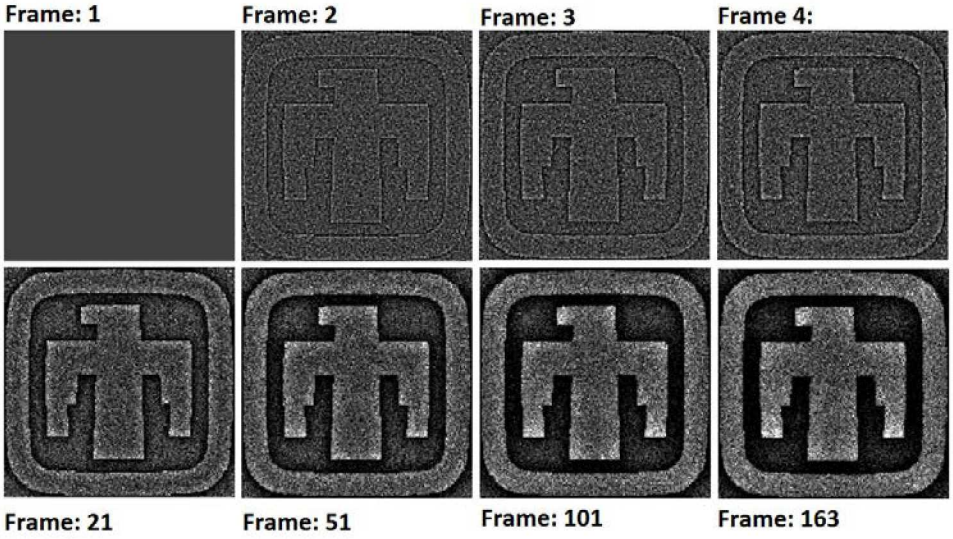

Figure 5-10: Evolution of walker movements on a 150 x 150 2D mesh with node probabilities defined to "encourage" the creation of the Sandia Thunder Bird logo. Black pixels are an absence of walkers and white pixels are a presence of 20 or more walkers.

crossbar configuration for this structure is defined in **Figure 5-11**. Notice that in contrast to the tree structure defined in **Figure 5-8**, no relay neurons are utilized and for the same four directions, 16 neurons are used instead of 21.

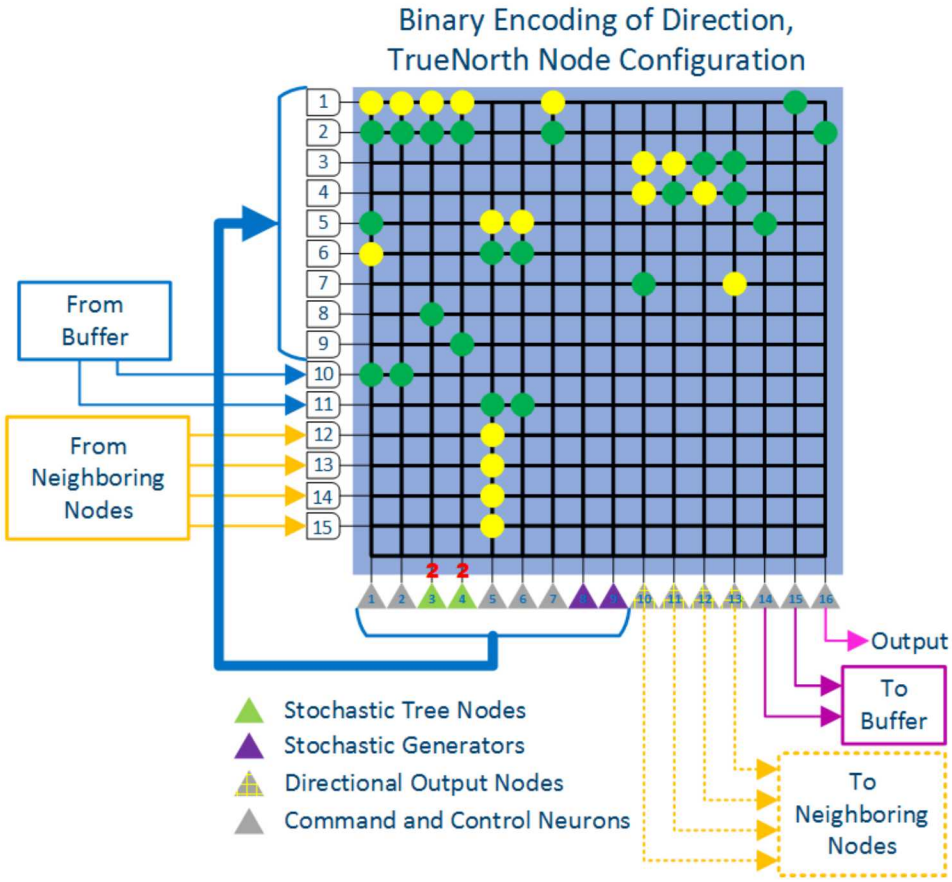

Figure 5-11: Binary encoding of directions in a TrueNorth crossbar configuration. Green circles represent excitatory connections and yellow circles represent inhibitory connections. All neuron thresholds are set at a value of 1, except in the case of the Stochastic Tree Nodes, which have a threshold value of 2. Neuron to axon connectivity for neuron/axon indexes of 1 – 9 are mapped one to one.

6. APPLICATION IMPACT

6.1. Comparison of Particle and Density Methods

Notably, the two models of random walks shown here are functionally equivalent, but each offer advantages under particular circumstances. For instance, the number of neurons required for density method scales with spatial resolution, and the number of particles being modeled is dynamically accounted for in the time required for the model to run. This configuration may thus be well-suited for neuromorphic systems whose neurons are capped at a fixed level whereas the time a simulation can be run is flexible. Thus, the number of particles can be tuned to achieve the statistical significance demanded by an application. Alternatively, the particle method models each walker independently, thus the time for a simulation to run is independent of the number of walkers so long as there are sufficient neurons to represent the requisite spatial resolution within each neuron.

There are several reasons beyond scaling that one method may be preferable to the other. While perhaps not as obvious, the paths taken by individual particles are preserved within the spike timings of the density method; however, the behavior of individual paths is directly retrievable from the particle methods. This is of use in models of certain path-dependent financial options for instance [15]. On the other hand, for many applications the density of walkers at a given spatial location and time is the critical output of stochastic process models. The density method by its nature provides an estimation of the density at all locations of the space at all times, whereas the particle method would require a subsequent integration of information from all of the independent circuits.

Finally, the two models here each offer compelling potential advantages on different neuromorphic platforms, such as the IBM TrueNorth chip [22], Intel's Loihi chip [11], Sandia's STPU architecture [16], and the Manchester SpiNNaker platform [14]. The mapping of these algorithms to spiking neuromorphic systems will be a subject of a future study. However, we anticipate that these algorithms should map well to these and other platforms, as the highly parallel nature of random walk processes makes them well suited for neuromorphic architectures. We conclude by highlighting the point that the efficient implementation of a strictly numerical process such as the random walk on neuromorphic hardware would represent a major new capability for systems generally designed to implement tasks such as neural processing and machine learning.

6.2. Consideration towards Radiation Transport and Molecular Dynamics

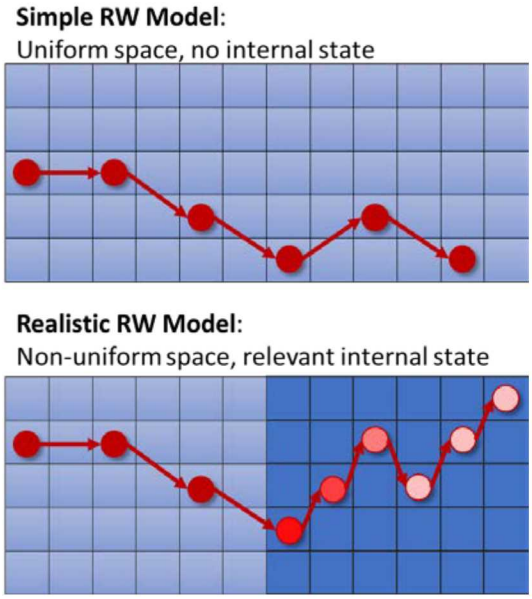

Figure 6-1: Illustration of additional model details necessary for random walk simulation of radiation transport

A significant application area for applying neuromorphic hardware simulations of random walks would be domains such as radiation transport and molecular dynamics. In radiation transport, established code bases implement Monte Carlo simulation of photons traversing through a physical space, with random perturbations based on the energy of particles and the local parameters (e.g., density) of a spatial location. The parallel simulation of many non-interacting particles is well-suited for our candidate stochastic algorithms; however, these algorithms and their associated theoretical analysis are only partially applicable. As shown in **Figure 6-1**, the dependency of the RW on local spatial structure and an internal state of the particle (that changes over time) is a complexity that requires accounting for in our stochastic models, and accordingly may impact the theoretical trade-offs associated with our candidate methods. One reason that radiation transport is of particular interest is that its Monte Carlo requirements are not particularly well-suited for GPU acceleration, which is what future HPC platforms are beginning to emphasize. Part of the mismatch between GPUs and random walk simulations lies in the divergent behavior of independently simulated particles – GPUs are ideal when the same operations can be performed on a subset of data, but in the case of divergent random walks, no matter how well balanced initially, the individual particles will begin to behave differently.

A second application domain worth considering is molecular dynamics. Molecular dynamics simulations, such as Direct Simulation Monte Carlo (DSMC) typically leverage a random walk step and then an interaction step. Because potential interactions scale worse than particle movements, the random walk stage is generally not seen as the limiting cost in these models. However, the density algorithm in

particular would also appear to have advantages in implicitly identifying which particles may interact (i.e., spatially co-located).

We have done some preliminary scoping of our algorithms against the SPARTA DSMC code-base benchmarks (http://sparta.sandia.gov/bench.html), in which they quantify the suitability of different HPC platforms on weak- and strong-scaling of the diffusion process of DSMC. In these benchmarks, they determined that state-of-the-art HPC nodes (either Mira at Argonne National Laboratory or Chama at Sandia National Laboratories) can perform roughly 100 million particle moves per second per node over a 1M grid cell mesh. Through back-of-envelope calculations, we can determine that the TrueNorth implementations described above, if scaled to 1M grid cells, would require about 40 TrueNorth chips and could match the HPC node throughput. While 40 TrueNorth chips is a lot to match a single node (and suggests that not enough TrueNorth chips exist globally to match a decent HPC system), the overall power of that theoretical 40 TrueNorth board would be *at least* an order-of-magnitude lower than the corresponding HPC node (<5 Watts vs >100 Watts for the HPC node).

Notably this benchmark assessment is preliminary, and our random walk model is not designed with the same precision and state-variables as the DSMC codes. However, this result does suggest that there is a viable path forward for such implementations, particularly since neither TrueNorth nor our algorithms are optimized for this application.

6.3. Density Method → Graphs?

One strength of the density method is that the algorithm makes no assumptions about the structure of the underlying graph. Hence, the method readily extends to random walks on arbitrary (di-)graphs. This notion provides many potential application spaces.

6.3.1. Finding the Shortest Path

The task of finding the shortest path between nodes is a classic graph problem with many real-world applications (e.g. navigation, recommendations). Spiking network algorithms to compute shortest path are easy to write (maybe a citation?) but rely on the ability to send arbitrarily many spikes. In practice, on-chip spike routers will likely get overwhelmed in such a situation, limiting the size of a potential graph search. Instead, we look to approximate the shortest path using our random walk mechanism. The process is simple: Assume we want to know the shortest path between nodes A and B. Start n walkers at node A. Run the simulation until a walker appears at node B. The time it takes for this to occur is an upper bound on the shortest path.

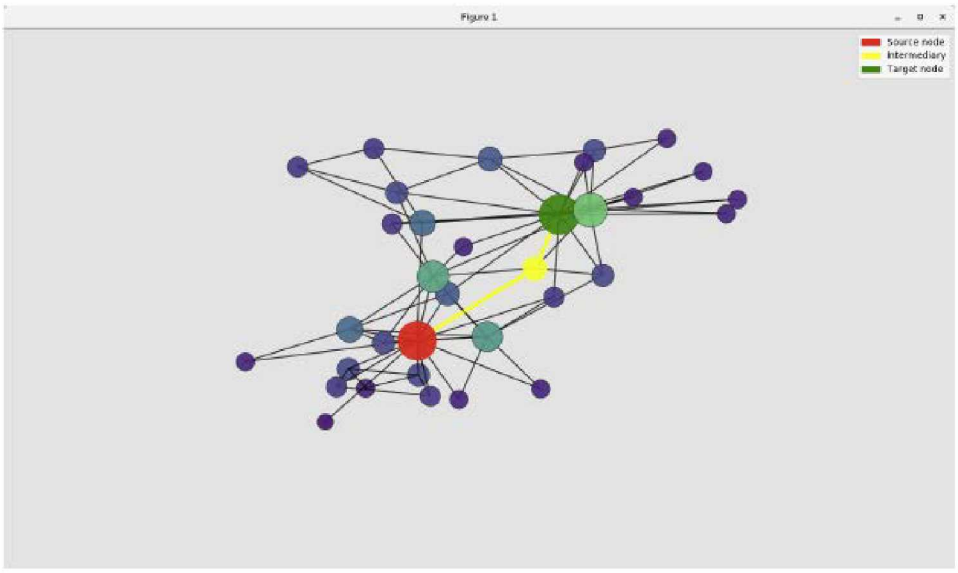

Figure 6-2: Shortest path in graph found with density algorithm

6.3.2. Triangle Inclusion

The algorithm in 6.4.1 can be easily extended to detect if a node is located within a triangle of the graph. We simply look for paths of length 3 starting and end at node *A*.

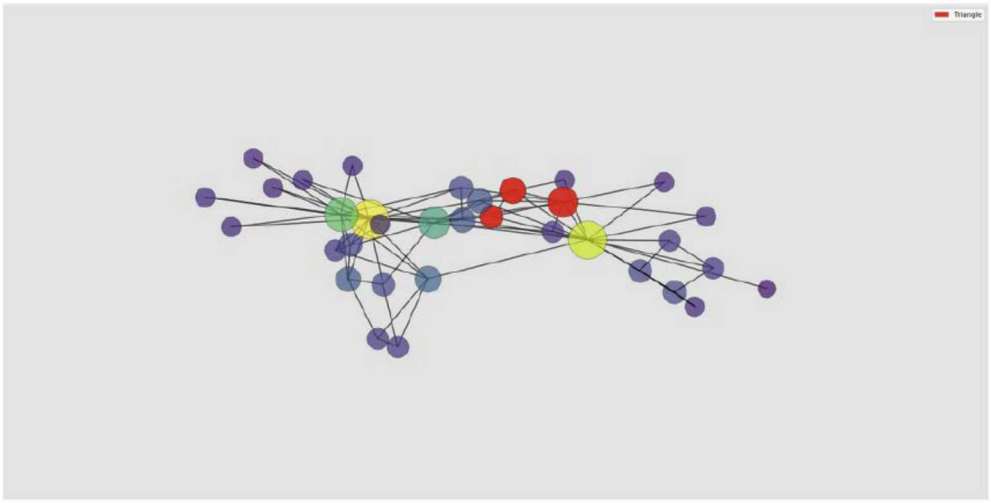

Figure 6-3: Triangle counting with density algorithm

Though, we note that this is just an approximation and that the probability of correctly determining inclusion is dependent on the shape of the graph and the number of initial walkers *n*.

6.3.3. Graph Partitioning

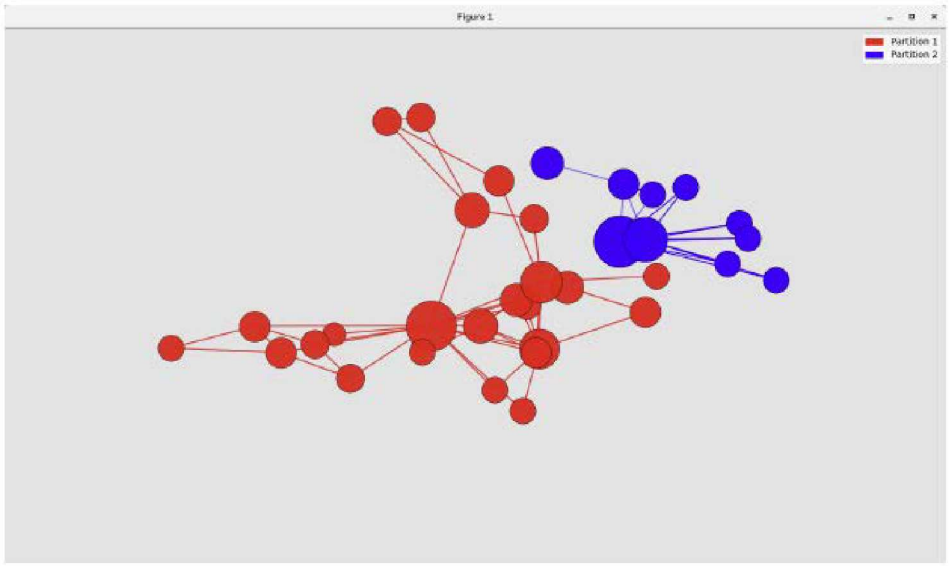

Figure 6-4: Graph partitioning with density algorithm

Additionally, we can use the algorithm 6.4.1 to as a method for graph partitioning. The goal of graph partitioning is to divide a graph into two (or more) subgraphs. However, gauging the quality of the partitioning is difficult, with many different accepted application-specific metrics. Instead of targeting a particular metric, we approach this generally and note that many different graph partitioning schemes depend on distances between nodes which we can compute readily. We have implemented a simple threshold-based partitioning scheme were nodes that are close (estimated shortest path is below a pre-set threshold) are determined to be in one partition and the remainder is in the other.

6.3.4. Image Segmentation

Random walks are also an effective method for image segmentation (separating objects of interest from the background). We devised a simple, density-method compatible, image segmentation algorithm. For this, the pixels are each represented by a node on the graph, and the probability of transition is determined by the relative pixel intensity. Initial walkers are distributed through the graph either randomly or uniformly. When the simulation is run, the walkers tend towards darker areas of the image, and we the density of walkers will provide the segmentation. Note, however, that this is different than a threshold segmentation algorithm due to the local differencing and the stochastic nature.

7. CONCLUSION

In this project, this project has demonstrated that small-scale neural circuits can efficiently and scalably implement random walk simulations. While this paper does not examine other aspects of stochastic process models that are critical for many applications, such as complex boundary conditions and interactions between particles, the models in this paper are designed to be extended towards such considerations.

The primary results of this report are best described as a "proof-of-principle." The algorithms here are designed to illustrate the scalability and feasibility of these different perspectives on simulating random walks on neuromorphic hardware. They are not optimized, and the fact they appear to have some scaling advantages over conventional approaches already is somewhat surprising. Further, as has been mentioned before, these algorithms are not yet configured to accepted other real-world considerations, such as boundary conditions or additional state variables. Nevertheless, the results we see here suggest that these additional considerations should be feasible, if not straightforward, to implement.

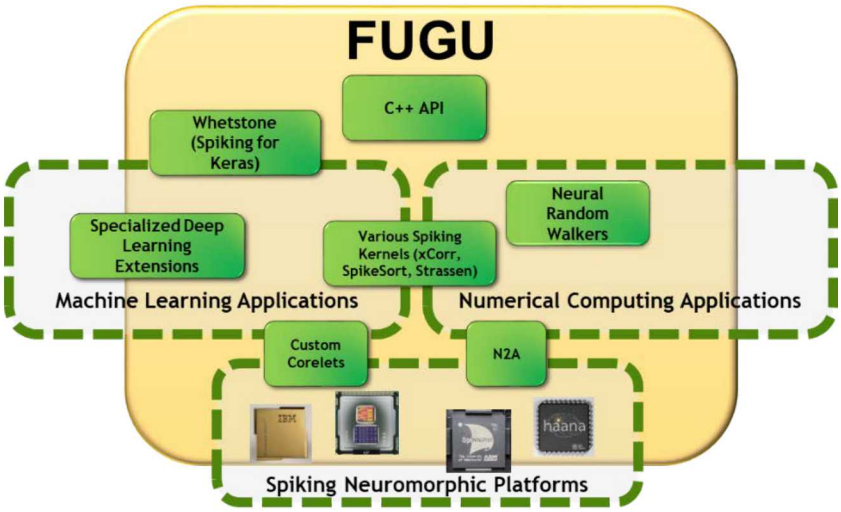

Figure 7-1: Conceptual illustration of Fugu programming stack for neuromorphic hardware. The random walk algorithms described here would fit into the intermediate levels of this approach.

Real-world leveraging of these approaches will require more than just more sophisticated and optimized algorithms. First, a more accessible programming paradigm by which general computer scientists and physicists can leverage these approaches is critical. Neural computing is "weird," and it cannot be expected that an expert in a given application area will be familiar with encoding their problem as a dynamic neural graph. For this reason, a more sophisticated software stack will be required for the community to adopt new techniques such as this. In parallel efforts at Sandia, some of the authors of this report have begun to develop a software interface for neuromorphic hardware known as *Fugu* (Fugu being the Japanese name of pufferfish, which is both a "spiky" animal as well as the source of tetrodotoxin, a

chemical useful in neuroscience studies of neuron spiking dynamics). The random walk algorithms described here, along with other spiking neural algorithms we have developed for simple numerical kernels, are an example of the middle-stage of the Fugu pipeline (**Figure 7-1**). Ideally, to use these algorithms, a user would only need to make a simple function call from C++ or Python to produce the appropriate neural circuit of random walkers, and that then gets compiled on the hardware.

REFERENCES

1. Aimone, J.B. Exponential scaling of neural algorithms-a future beyond Moore's Law? *arXiv preprint arXiv:1705.02042*.
2. Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., Imam, N., Nakamura, Y., Datta, P. and Nam, G.-J. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *34* (10). 1537-1557.
3. Ambrogio, S., Narayanan, P., Tsai, H., Shelby, R.M., Boybat, I., Nolfo, C., Sidler, S., Giordano, M., Bodini, M. and Farinha, N.C. Equivalent-accuracy accelerated neural-network training using analogue memory. *Nature*, *558* (7708). 60.
4. Amir, A., Datta, P., Risk, W.P., Cassidy, A.S., Kusnitz, J.A., Esser, S.K., Andreopoulos, A., Wong, T.M., Flickner, M. and Alvarez-Icaza, R., Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, (2013), IEEE, 1-10.
5. Benjamin, B.V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A.R., Bussat, J.-M., Alvarez-Icaza, R., Arthur, J.V., Merolla, P.A. and Boahen, K. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, *102* (5). 699-716.
6. Black, F. and Scholes, M. The pricing of options and corporate liabilities. *Journal of political economy*, *81* (3). 637-654.
7. Buesing, L., Bill, J., Nessler, B. and Maass, W. Neural dynamics as sampling: a model for stochastic computation in recurrent networks of spiking neurons. *PLoS computational biology*, *7* (11). e1002211.
8. Chuang, T. and Fukuda, M., A parallel multi-agent spatial simulation environment for cluster systems. in *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, (2013), IEEE, 143-150.
9. Codling, E.A., Plank, M.J. and Benhamou, S. Random walk models in biology. *Journal of the Royal Society Interface*, *5* (25). 813-834.
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. Introduction to algorithms second edition, The MIT Press, 2001.
11. Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S.H., Dimou, G., Joshi, P., Imam, N. and Jain, S. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro*, *38* (1). 82-99.
12. Dennard, R.H., Gaensslen, F.H., Rideout, V.L., Bassous, E. and LeBlanc, A.R. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, *9* (5). 256-268.
13. Edwards, A.H., Barnaby, H.J., Campbell, K.A., Kozicki, M.N., Liu, W. and Marinella, M.J. Reconfigurable memristive device technologies. *Proceedings of the IEEE*, *103* (7). 1004-1033.
14. Furber, S.B., Lester, D.R., Plana, L.A., Garside, J.D., Painkras, E., Temple, S. and Brown, A.D. Overview of the spinnaker system architecture. *IEEE Transactions on Computers*, *62* (12). 2454-2467.
15. Goldman, M.B., Sosin, H.B. and Gatto, M.A. Path dependent options:"Buy at the low, sell at the high". *The Journal of Finance*, *34* (5). 1111-1127.
16. Hill, A.J., Donaldson, J.W., Rothganger, F.H., Vineyard, C.M., Follett, D.R., Follett, P.L., Smith, M.R., Verzi, S.J., Severa, W. and Wang, F., A Spike-Timing Neuromorphic

Architecture. in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*, (2017), IEEE, 1-8.
17. Indiveri, G., Linares-Barranco, B., Hamilton, T.J., van Schaik, A., Etienne-Cummings, R., Delbruck, T., Liu, S.-C., Dudek, P., Häfliger, P. and Renaud, S. Neuromorphic Silicon Neuron Circuits. *Frontiers in Neuroscience*, *5*. 73.
18. James, C.D., Aimone, J.B., Miner, N.E., Vineyard, C.M., Rothganger, F.H., Carlson, K.D., Mulder, S.A., Draelos, T.J., Faust, A. and Marinella, M.J. A historical survey of algorithms and hardware architectures for neural-inspired and neuromorphic computing applications. *Biologically Inspired Cognitive Architectures*.
19. Johnston, D. and Wu, S.M.-S. *Foundations of cellular neurophysiology*. MIT press, 1994.
20. Masuda, N., Porter, M.A. and Lambiotte, R. Random walks and diffusion on networks. *Physics Reports*.
21. Mead, C. Neuromorphic electronic systems. *Proceedings of the IEEE*, *78* (10). 1629-1636.
22. Merolla, P.A., Arthur, J.V., Alvarez-Icaza, R., Cassidy, A.S., Sawada, J., Akopyan, F., Jackson, B.L., Imam, N., Guo, C. and Nakamura, Y. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345* (6197). 668-673.
23. Moore, G.E., Progress in digital integrated electronics. in *Electron Devices Meeting*, (1975), 11-13.
24. Parekh, O., Phillips, C.A., James, C.D. and Aimone, J.B., Constant-Depth and Subcubic-Size Threshold Circuits for Matrix Multiplication. in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, (2018), ACM, 67-76.
25. Pickett, M.D., Medeiros-Ribeiro, G. and Williams, R.S. A scalable neuristor built with Mott memristors. *Nature materials*, *12* (2). 114.
26. Schemmel, J., Briiderle, D., Griibl, A., Hock, M., Meier, K. and Millner, S., A wafer-scale neuromorphic hardware system for large-scale neural modeling. in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, (2010), IEEE, 1947-1950.
27. Severa, W., Parekh, O., Carlson, K.D., James, C.D. and Aimone, J.B., Spiking network algorithms for scientific computing. in *Rebooting Computing (ICRC), IEEE International Conference on*, (2016), IEEE, 1-8.
28. Siu, K.-Y., Roychowdhury, V. and Kailath, T. *Discrete neural computation: a theoretical foundation*. Prentice-Hall, Inc., 1995.
29. Sreenivasan, S. and Fiete, I. Grid cells generate an analog error-correcting code for singularly precise neural computation. *Nature neuroscience*, *14* (10). 1330.
30. van de Burgt, Y., Lubberman, E., Fuller, E.J., Keene, S.T., Faria, G.C., Agarwal, S., Marinella, M.J., Talin, A.A. and Salleo, A. A non-volatile organic electrochemical device as a low-voltage artificial synapse for neuromorphic computing. *Nature materials*, *16* (4). 414.
31. Verzi, S.J., Rothganger, F., Parekh, O.D., Quach, T.-T., Miner, N.E., Vineyard, C.M., James, C.D. and Aimone, J.B. Computing with spikes: The advantage of fine-grained timing. *Neural computation*. 1-31.
32. Verzi, S.J., Vineyard, C.M., Vugrin, E.D., Galiardi, M., James, C.D. and Aimone, J.B., Optimization-based computation with spiking neurons. in *Neural Networks (IJCNN), 2017 International Joint Conference on*, (2017), IEEE, 2015-2022.
33. Von Neumann, J. *The computer and the brain*. Yale University Press, 2012.
34. Waldrop, M.M. The chips are down for Moore's law. *Nature News*, *530* (7589). 144.

35. Wilmott, P., Howison, S. and Dewynne, J. *The mathematics of financial derivatives: a student introduction*. Cambridge university press, 1995.

DISTRIBUTION

1	MS0899	Technical Library	9536 (electronic copy)
1	MS0359	D. Chavez, LDRD Office	1911
1	MS0161	Legal Technology Transfer Center	11500


Sandia National Laboratories