SANDIA REPORT

SAND2021-14142
Printed October, 2021

Neuromorphic Graph Algorithms

Ojas Parekh, Yipu Wang, Yang Ho, Cynthia A. Phillips, Ali Pinar, James B. Aimone, William Severa

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

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

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

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 Road
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

Neuromorphic Graph Algorithms

Ojas Parekh[1], Yipu Wang[1], Yang Ho[1], Cynthia A. Phillips[1], Ali Pinar[1], James B. Aimone[1], William Severa[1]

ABSTRACT

Graph algorithms enable myriad large-scale applications including cybersecurity, social network analysis, resource allocation, and routing. The scalability of current graph algorithm implementations on conventional computing architectures are hampered by the demise of Moore's law. We present a theoretical framework for designing and assessing the performance of graph algorithms executing in networks of spiking artificial neurons. Although spiking neural networks (SNNs) are capable of general-purpose computation, few algorithmic results with rigorous asymptotic performance analysis are known. SNNs are exceptionally well-motivated practically, as neuromorphic computing systems with 100 million spiking neurons are available, and systems with a billion neurons are anticipated in the next few years. Beyond massive parallelism and scalability, neuromorphic computing systems offer energy consumption orders of magnitude lower than conventional high-performance computing systems.

We employ our framework to design and analyze new spiking algorithms for shortest path and dynamic programming problems. Our neuromorphic algorithms are message-passing algorithms relying critically on data movement for computation. For fair and rigorous comparison with conventional algorithms and architectures, which is challenging but paramount, we develop new models of data-movement in conventional computing architectures. This allows us to prove polynomial-factor advantages, even when we assume a SNN consisting of a simple grid-like network of neurons. To the best of our knowledge, this is one of the first examples of a rigorous asymptotic computational advantage for neuromorphic computing.

[1]Sandia National Laboratories

ACKNOWLEDGMENT

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission 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. We are grateful to Joseph Mitchell (Stony Brook University), Michael Bender (Stony Brook University) and David Tench (Stony Brook University and Rutgers University) for helpful discussions.

CONTENTS

1. Introduction **10**

2. Neuromorphic Computing Fundamentals **13**

2.1. Neuromorphic platforms . . . 13
2.2. Leaky-integrate and fire neurons . . . 14
2.3. Spiking neural network model . . . 15
2.4. Considerations for neuromorphic advantages . . . 18

3. Neuromorphic Graph Algorithms **21**

3.1. Spiking graph algorithm for shortest paths . . . 21
3.2. Our results . . . 22
3.3. Dynamic programming . . . 25

4. Embeddings **27**

4.1. Hardness results for embedding in a Loihi-like architecture . . . 29

5. Experiments **31**

5.1. Hardware limitations . . . 31
5.2. Results . . . 32
5.2.1. Probed vs unprobed . . . 32
5.2.2. Scaling . . . 33

6. Conclusions **35**

References **36**

LIST OF FIGURES

Figure 2-1. An Intel Loihi Nahuku form factor, which can support up to 32 chips (16 on the underside) attached to a host FPGA board (left). Each chip offers 128K neurons. 13
Figure 2-2. Current neuromorphic architectures aggregate many-core chips into boards. 14
Figure 2-3. Circuit (A) uses neurons to simulate an $O(d)$ synaptic delay on neuromorphic architectures that do not natively support such delays. When the first neuron activates, its feedback loop causes it to repeatedly fire until the second neuron receives $d-1$ spikes. When the second neuron fires, it stops the first neuron. Circuit (B) shows how to use neurons as memory. The self-loop on neuron M allows it to act as a latch, firing indefinitely once it has fired. The *recall* input at neuron C propagates the value of M to the *output*. Neuron M can be *reset* by an inhibitory (negative weighted) link from C to M (not shown). 17

Figure 3-1. An illustration of the neuromorphic algorithm computing a shortest path from source S to sink T. Blue nodes fire a spike. Red nodes have received a spike and will not fire again. 22

Figure 4-1. The stacked grid (or crossbar) H_3. The general H_n is a topology we may reasonably expect as a subset of every neuromorphic architecture. 28

Figure 5-1. Chart that shows run time (s) as a function of the Width of the input grid graph. It is cleared that using spike probes incurs an order-of-magnitude run-time overhead. 32
Figure 5-2. Experiment looking at how our shortest path algorithm scales on layer graphs. Although asymptoically the run time grows linearly, we note that there are jumps in run time that occur at chip and board core limits, i.e. when the number of cores needed to embed the layer graphs excceed a single chip/board. 33
Figure 5-3. Experiment looking at how our shortest path algorithm scales on layer graphs with saturated edges between layers, thereby producing a neural network where the synapse fan-in limit for each layer is met. We note that the chip and board boundary jumps are not present. 34

LIST OF TABLES

Table 2-1. Selection of Current Scalable Neuromorphic Platforms 14

Table 3-1. Comparison of complexities of neuromorphic and conventional methods for Single-Source Shortest Path (SSSP) problems. For this table, we assume that there is a single source node and single destination node. The number of nodes and edges are denoted by n and m, respectively. L is the length of the shortest path with at most k edges (where $k = n-1$ for SSSP); U is an upper bound on the edge lengths; α is the number of edges in the shortest path; and c is the number of words in the smallest, fastest memory level (typically a constant with respect to n and m). The m in the lower bounds can be replaced with the total number of bits in the input, taking edge lengths into account. Lower bounds are computed using the DISTANCE model described in Section 2.3.
The neuromorphic algorithms can offer an asymptotic advantage in all cases except polynomial-time SSSP when ignoring data movement costs. As an example, for the polynomial-time k-hop SSSP neuromorphic algorithms, we obtain a factor $\Omega(k/\log n)$ advantage when ignoring data-movement costs, and a factor $\Omega(m^{1/2}/\log n)$ advantage with data-movement costs, under the reasonable assumptions that U is polynomial in n and $c = O(1)$. The latter can better, depending on the relationship of n, m, and k. 24

Parts of this report are drawn or adapted from the publications produced under our project [4–6].

1. INTRODUCTION

The brain has been proposed as a potential inspiration for parallel computing since the earliest days of computer science [37]. Efforts to emulate the brain's architecture, known as neuromorphic computing, began in the 1980s. Recently, there are increasingly large-scale efforts, including industrial efforts from IBM (TrueNorth, [28]) and Intel (Loihi, [13]), and academic efforts including SpiNNaker [24], NeuroGrid [10], and BrainScales [33]. Most neuromorphic systems use a hierarchical graph network architecture, with local cores containing up to 1,000 highly interconnected neurons and many cores networked together on each chip, as described in more detail in Table 2-1 in Section 2.1. Neuromorphic systems with 100 million total neurons are currently available [23, 29], and systems with 1 billion neurons are anticipated with a few years.

While neuromorphic hardware seems naturally suited to emerging cognitive and artificial intelligence applications [3, 17, 34], there has been growing interest in more general computational applications. Yet it remains unclear what, if any, theoretical advantage neuromorphic computation provides. Conventional neural networks used for deep learning employ threshold gates, yet threshold-gate algorithms are not entirely satisfying as examples of neuromorphic algorithms since they do not leverage some of the features of current spiking neuromorphic architectures (SNAs) such as neuron dynamics or recurrent computation. Thus it has remained an open question if the power of SNAs can be harnessed to demonstrate rigorous neuromorphic resource advantages over conventional computing systems.

Spiking neural networks Theoretical models capturing the behavior of spiking neural networks (SNNs) were first proposed by Maass [26], and a specialization focusing on leaky-integrate and fire (LIF) neurons, featured in current and emerging SNAs, was recently proposed by Kwisthout and Donselaar [25]. Few theoretical results exploring the power of SNNs are known, with some recent examples by Hitron & Parter [19], Hamilton, Mintz, & Schuman [16], Ali & Kwisthout [7], Hitron, Parter, & Perri [20], and Hitron, Musco, & Parter [18].

Spiking neural networks generalize feed-forward circuit families of threshold gates, which is the natural computational model associated with the well-studied complexity class TC. Threshold gates are more powerful than (unbounded fan-in versions of) conventional Boolean logic gates in the sense that constant-depth threshold circuits, corresponding to the complexity class TC^0, can compute functions that constant-depth conventional Boolean circuits with unbounded fan-in cannot [14, 38, 39]. This suggests that threshold circuits, and more generally SNNs, may offer resource advantages over conventional models of parallel computing. Indeed Parekh et al. [32] recently gave a TC^0 circuit family, constituting a constant-time neuromorphic algorithm, for matrix multiplication using a sub-cubic number of neurons (related to the complexity of fast matrix multiplication algorithms). In contrast more conventional parallel algorithms for matrix multiplication require logarithmic time.

We seek to demonstrate resource advantages offered by *recurrent* networks of spiking LIF neurons and provide a means for fair comparison with conventional algorithms. The latter is particularly challenging since, for example, in SNNs memory and computation are conflated and implicitly represented by neurons. We note that SNNs where spike times are discretized may be simulated, with polynomial overhead, in TC by using layers of a threshold gate circuit to simulate discrete time steps. Some care needs to be taken to ensure that LIF dynamics (described in the next section) are properly simulated. This builds upon a standard type of construction which is used to show that Boolean circuit families can simulate Turing machines. However, such constructions impose too much overhead for our goal of demonstrating reasonably practical neuromorphic algorithms offering polynomial advantages over conventional counterparts. For a more detailed comparison of TC with both discrete and continuous theoretical models of recurrent SNNs, see the survey by Šíma and Orponen [38].

Our contributions We initiate a formal study of neuromorphic graph algorithms, and consider as an example the k-hop shortest path problem, where (single-source) paths can have at most k edges. We propose straightforward neurmorphic implementations of conventional algorithms. Our approach generalizes to computing $A^k x$, given a matrix A and vector x (see Section 2.3). We intend for the simplicity of these problems to allow the reader to learn perhaps unfamiliar algorithmic and architectural concepts while working through comfortable examples. Our neuromorphic algorithms pass messages, relying on data movement for computation. We develop a more conventional data-movement model for fair comparison with conventional algorithms and demonstrate a polynomial-factor advantage for certain ranges of problem parameters. We observe that SNAs are a natural computing model for graph algorithms and observe connections to distributed computing.

Neuromorphic computing is in its infancy, hence obtaining our results entails a vertical mutli-disciplinary approach considering gate-level neuromorphic circuits, topological restrictions in SNAs, and a basis for fair and rigorous comparisons with conventional architectures and algorithms. The latter is paramount and challenging. In order to do so we develop data-movement models for conventional computing to enable fair comparisons with SNAs. We justify our models and comparisons by surveying existing SNAs. We intend for our neuromorphic models, primitives, algorithms, and overall considerations to foster further algorithmic and programming-model research and provide cues to hardware developers.

Our precise results for shortest-path problems are summarized in Table 3-1 in Section 3.2 and hold even when we assume an SNA with a simple grid-like network of neurons. The table lists our results for a single source node and a single destination node, but our algorithms can easily be generalized to multiple destinations. Our algorithms use a number of neurons polynomial in the size of the input graph. To the best of our knowledge, this is one of the first examples of context for fair comparison of conventional and neuromorphic algorithms, and an asymptotic demonstration of a neuromorphic advantage. Some of our results may be extended to apply for dynamic programming as detailed in [6].

Organization Chapter 2 introduces neuromorphic-computing fundamentals, including considerations for fair comparison to conventional computing. Chapter 3 illustrates a simple,

natural neuromorphic algorithm for single-source shortest paths and presents our results for shortest-path problems. Chapter 4 describes the embedding problem that must be solved in order to execute neuromorphic graph algorithms on neuromorphic architectures offering only grid-like connectivity among neurons. Finally, Chapter 5 describes experiments conducted to benchmark simple neuromorphic graph algorithms.

2. NEUROMORPHIC COMPUTING FUNDAMENTALS

2.1. Neuromorphic platforms

Today's neuromorphic systems, even those from industry, generally represent research-grade platforms still in development. Available systems range from targeting low-Size, Weight and Power (SWaP) and embedded applications [8] to large-scale, data-center-type systems [27]. Additionally, given the nascent state of these architectures, we see a wide variety of hardware-imposed trade-offs. For example, one system may prioritize neuron density, whereas another may prioritize network configurability. In Table 2-1, we outline some of the key statistics and metrics on several of today's prominent large-scale platforms in addition to an Intel Core i7 CPU for reference. We remark that several of the entries are estimates due to a memory tradespace; for example, on some platforms using a highly connected neuron, which requires a large amount of synaptic memory, may lessen the total number of neurons available.

For digital neuromorphic systems, node process is essentially on par with traditional processors (14-28nm). Power consumption is considerably less for the neuromorphic platforms (e.g. 1W for SpiNNaker, 70-150mW for TrueNorth compared to 35W), though we recognize that CPUs can concurrently run other tasks, such as graphics. Clock rates are much higher for CPUs, as evidenced by the i7's 4.3 GHz clock rate. This is partially abated by the massive parallelization on neuromorphic systems (e.g. 8 cores versus 100K-1M neurons), and we expect clock rates of neuromorphic systems to increase. Figure 2-1 shows an Intel Loihi Nahuku board with 8 chips populated (4 visible, 4 on the underside) which supports 128K neurons per chip or approximately 1 million neurons per board. Fully populated, each board supports approximately 4 million neurons, and a 100-million neuron system is currently available [23, 29].

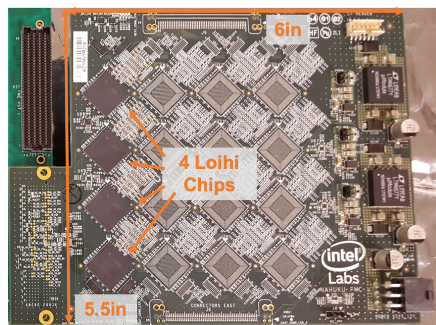

Figure 2-1. An Intel Loihi Nahuku form factor, which can support up to 32 chips (16 on the underside) attached to a host FPGA board (left). Each chip offers 128K neurons.

Table 2-1. Selection of Current Scalable Neuromorphic Platforms

Platform	TrueNorth [28]	Loihi [13]	SpiNNaker 1 [31, 35]	SpiNNaker 2 [21]	Core i7-9700T [22]
Organization	IBM	Intel	U. Manchester	U. Manchester	Intel
Design	ASIC	ASIC	ARM	ARM	CPU
Process	28nm	14nm	130nm	22nm	14nm
Clock	1KHz	Asynchronous†	-	100–600MHz (.45–.60V)	4.30GHz (Max Turbo)
Neurons/Core	256	1,024	$\approx 1,000$	$\approx$ 800k per **chip**	N/A
Cores/Chip	4096	128	16	-	N/A
pJ/Spike Event	26	23.6	$6–8 \times 10^3$	-	N/A
Running Power (Approx.)	70–150mW per Chip	$\approx$ .45 W	1W per chip (peak)	0.72 Watts	35W TDP

†Within-tile spike latency 2.1ns; barrier sync time under 465ns

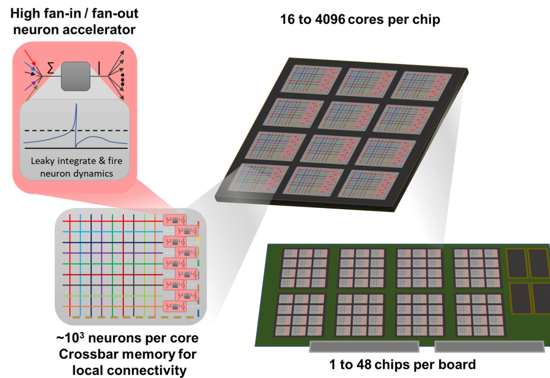

Figure 2-2. Current neuromorphic architectures aggregate many-core chips into boards.

2.2. Leaky-integrate and fire neurons

The basic processing units in a neuromorphic system are (artificial) neurons. We focus on discrete *leaky-integrate and fire* or *LIF* neurons, common in current and near-term SNAs (see Table 2-1). For a set $S \subseteq \mathbb{R}$, S_+ refers to its nonnegative elements.

LIF is a common abstraction of biological neuron dynamics, and a common target for current and proposed neuromorphic hardware. While mathematically compact, it is similar to even simpler neural computing models such as threshold gates, which have more extensive theoretical studies (e.g., the complexity class TC). Networks of LIF neurons allow energy-efficient communication, since outgoing communication from a neuron only occurs at spikes. Indeed, the promise of spiking neuromorphic hardware in large part can be attributed to this event-driven communication.

Definition 1: Leaky-integrate and fire system model Time, parameterized by $t \in \mathbb{N}_+$, proceeds in discrete steps over a collection of *neurons* $N := \{1,\ldots,n\}$. Each neuron $i \in N$ has corresponding time-dependent Boolean *spike* function $f_i(t)\colon \mathbb{N}_+ \to \{0,1\}$ and *voltage* $v_i(t)\colon \mathbb{N}_+ \to \mathbb{R}$. We say that neuron i *spikes* or *fires* at time t if $f_i(t) = 1$. For convenience we define $f_i(t) := 0$ for $t \leq 0$. In addition a *reset voltage* $v_{i,\text{reset}} \in \mathbb{R}$, a *threshold voltage* $v_{i,\text{threshold}} \in \mathbb{R}$, and a *decay rate* $\tau_i \in [0,1]$ are programmable parameters for each neuron i. Neurons have directed connections called *synapses*. Each synapse between a pair of neurons i and j has programmable weight $w_{ij} \in \mathbb{R}$ and delay $d_{ij} \in \mathbb{N}_+$.

Definition 2: Leaky-integrate and fire neuron model We now describe the dynamics of a fixed neuron j, where subscripts are dropped for readability when context permits. The neuron starts with a voltage of $v(0) = v_{\text{reset}}$. At each time step $t \geq 1$, the voltage $v(t)$ of the neuron is updated with a decay from the previous time step and a voltage change from synaptic inputs (Eqs. (2.1) and (2.4)). The synaptic input for the neuron (v_{syn} below) is the sum of the weights w_{ij} of the active incoming neurons (those where neuron i spiked d_{ij} time ago). In continuous time the decay would be exponential, but in discrete time it is a τ fraction of the amount the voltage $v(t)$ is above v_{reset}. (If $\tau = 1$, this corresponds to the threshold gates used in deep-learning neural networks). The updated voltage, $\hat{v}(t+1)$ is compared to the neuron's threshold voltage, $v_{\text{threshold}}$ to determine if the neuron spikes ($f(t+1) = 1$ in Eq. (2.2)), and if so the voltage resets according to Eq. (2.2). The dynamics for a LIF neuron are:

$$\hat{v}(t+1) := v(t) - (v(t) - v_{\text{reset}}) \cdot \tau + v_{\text{syn}}(t) \tag{2.1}$$

$$f(t+1) := \begin{cases} 1, & \text{if } \hat{v}(t+1) > v_{\text{threshold}} \\ 0, & \text{if } \hat{v}(t+1) \leq v_{\text{threshold}} \end{cases} \tag{2.2}$$

$$\text{if } f(t+1) = 1 \text{ then } v(t+1) := v_{\text{reset}} \quad \text{else } v(t+1) := \hat{v}(t+1) \tag{2.3}$$

$$v_{\text{syn}}(t) := \sum_{i \in N} (f_i(t - d_{ij}) \cdot w_{ij}). \tag{2.4}$$

2.3. Spiking neural network model

Although our basic SNN model is in the spirit of the models of Maass [26] and Kwisthout & Donselaar [25], our focus is on optimization problems in graphs, and we employ a bit more streamlined model for this purpose.

Definition 3: Spiking neural network A *spiking neural network* (SNN) is a directed graph $G = (V,E)$, that may contain cycles and self-loops, where vertices represent a set $N := \{1,\ldots,n\}$ of leaky-integrate and fire (LIF) neurons, and directed edges represent synaptic connections between them. Neuron subsets $I \subseteq N$ and $O \subseteq N$ are designated as input and output neurons, respectively.

Each vertex $u \in V$ is parameterized by a 3-tuple (r_u, t_u, τ_u), specifying the $v_{\text{reset}} \in \mathbb{R}$, $v_{\text{threshold}} \in \mathbb{R}$, and decay parameter $\tau \in [0,1]$, respectively, as described in Section 2.2. Each directed edge $ij \in E$ has an associated weight $w_{ij} \in \mathbb{R}$ and delay $d_{ij} \in \mathbb{N}_+$. Computation is initiated by inducing spikes in a subset of the input neurons $S \subseteq I$ (at time $t = 0$); this may be viewed as an $|I|$-bit binary input with $|S|$ ones. Each vertex processes input spikes as they are received and potentially generates an output spike, according to the LIF dynamics defined in Section 2.2. Computation terminates when a designated terminal neuron u_t first spikes (at time $t = T$). At this point, the state of the output neurons O (i.e., whether they fired at time $t = T$) may be read out. The *execution time* of a computation is defined to be T.

The above model captures general asynchronous computation in spiking networks of LIF neurons. In addition we introduce a model for synchronous neuromorphic graph algorithms that resembles distributed computing models.

Definition 4: Neuromorphic graph algorithm A *neuromorphic graph algorithm* (NGA) executes on a directed graph $G = (V,E)$ in rounds consisting of message-broadcasting from each node and local computation at edges and nodes. At the beginning of round r, each node $i \in V$ broadcasts a λ-bit message, $m_{i,r-1}$, across all outgoing edges ij. Each message $m_{i,r-1}$ is processed as it traverses the edge ij, resulting in the message $m_{ij,r-1}$. For the final step in round r, each node j collects all incoming messages $m_{ij,r-1}$ and computes the message $m_{j,r}$ as a function of the incoming messages. A set of input messages $m_{i,0}$ is supplied at the start of the algorithm in round $r = 1$.

We assume that each edge ij has an SNN with λ input and output neurons that computes $m_{ij,r-1}$ using $m_{i,r-1}$ in T_{edge} time steps. Each node i has an SNN with $|N^-(i)|\lambda$ input neurons and λ output neurons, where $N^-(i)$ is the set of nodes with outgoing edges to node i; the SNN at each node computes $m_{i,r}$ in T_{node} time steps. We note that for an SNN, sending the all zeros message equates to none of the output neurons firing. The total execution time of an R-round NGA is $R(T_{\text{edge}} + T_{\text{node}})$.

Example of an NGA For the neuromorphic graph algorithms we consider, all the nodes will compute the same function, and all the edges will compute the same function. For example, suppose each message represents an integer and we are given an $n \times n$ matrix A. Let the vector $m_r = (m_{1,r}, \ldots, m_{n,r})$ represent the messages of an NGA on an n-node graph. We let each edge ij compute $m_{ij,r} = A_{ij} m_{i,r}$, and each node j compute $m_{j,r+1} = \sum_{i \in N^-(j)} m_{ij,r} = \sum_{i \in N^-(j)} A_{ij} m_{i,r}$. Such an NGA computes $m_{r+1} = A m_r$, and hence in r rounds computes $A^r m_0$.

By summing entries of A with message values on the edges and taking the minimum of message values at the nodes, we obtain a well-known approach for computing k-hop shortest paths. Although we use shortest paths as an exemplar in this paper, our techniques carry over to the more general matrix-vector multiplication problem above.

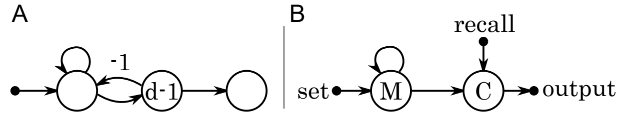

Figure 2-3. Circuit (A) uses neurons to simulate an $O(d)$ synaptic delay on neuromorphic architectures that do not natively support such delays. When the first neuron activates, its feedback loop causes it to repeatedly fire until the second neuron receives $d-1$ spikes. When the second neuron fires, it stops the first neuron. Circuit (B) shows how to use neurons as memory. The self-loop on neuron M allows it to act as a latch, firing indefinitely once it has fired. The *recall* input at neuron C propagates the value of M to the *output*. Neuron M can be *reset* by an inhibitory (negative weighted) link from C to M (not shown).

Comparison with distributed computing The NGA model is reminiscent of the LOCAL and CONGEST models used in distributed computing. In both models, we have a communication network represented by a graph; the nodes of this graph represent computational entities and the edges represent communication links on which the nodes send messages to each other. Although we assumed in the definition of the NGA model that computation occurs on edges, we could simply replace each edge with a path of length two and restrict computation only to nodes. Thus for NGA, the computational entities are spiking neural networks, the links are collections of λ synapses, and the messages are collections of λ spikes. NGAs may be readily simulated in LOCAL/CONGEST with a constant-factor overhead, as these models are only concerned with the number of rounds of communication necessary and are endowed with unbounded computation at nodes. More generally, for discrete-time SNNs, we may associate a CONGEST graph node with each neuron and a round with each time step. Each message is simply a single bit, indicating whether the neuron fired at time t, and the value of the message computed at each node may be obtained by simulating LIF dynamics from Section 2.2. Efficiently simulating delays on synapses becomes a challenge, as does handling continuous-time SNNs, since in the CONGEST model each message takes exactly one clock tick to traverse a link. This suggests a CONGEST-like model with a notion of programmable delays as a neuromorphic-inspired model for future study.

Conversely, algorithms in the CONGEST model cannot necessarily be efficiently implemented using SNNs, since CONGEST allows a node to perform any computation instantly. However, CONGEST algorithms that perform a polynomially-bounded amount of work at each node may be simulated using SNNs. Nanongkai [30] gives an $(1+o(1))$-approximation algorithm for single-source k-hop shortest paths in the CONGEST model. We have adapted his algorithm to an SNN algorithm [5]. For the sake of completeness, we mention that Ghaffari and Li [15] have described a $(1+\varepsilon)$-approximation algorithm for single-source shortest paths running in $\tau_{mix}2^{O(\sqrt{\log n})}$ time in the CONGEST model, where the input graph has n vertices and mixing time τ_{mix}.

Below we outline basic assumptions, motivated by neuromorphic hardware considerations, for implementing abstract neuromorphic algorithms in our SNN model.

Delays and synchronization We assume that there is minimum programmable delay δ, a hardware-specific constant. Each synapse has delay $d_{ij} = l\delta$, where $l \geq 1$ is an integer. Delays of 0

are prohibited, as inherent latency when a spike traverses a synapse is a reasonable physical assumption. This explicitly accounts for data-movement time and helps ensure we do not underestimate the execution time of our neuromorphic algorithms.

Our assumption of precise programmable delays is reasonable because current and near-term neuromorphic hardware is digital, where spikes are discrete events. Future neuromorphic systems may be analog or hybrid analog-digital; however, our work demonstrates the power of precise delays as a mechanism for synchronization and may guide neuromorphic-system developers seeking to understand features critical to the success of neuromorphic algorithms.

Using delays and dummy neurons, we assume that feed-forward circuits of threshold gates can run in time proportional to depth. Although many neuromorphic platforms support delays natively, some do not. We can simulate delays by replacing a synaptic link with two neurons with feedback between them (see Figure 2-3).

Neuromorphic memory Our algorithms must store information at graph nodes. We may use neurons with no leakage or self-loops to preserve state (see Figure 2-3).

Neuromorphic computational primitives Our neuromorphic graph algorithms assume basic computational primitives on λ-bit messages representing integers, such as summing values or taking the minimum or maximum over several messages. We have developed threshold-gate implementations of such circuits, which may be viewed as non-recurrent spiking neural networks [5].

2.4. Considerations for neuromorphic advantages

We seek to understand and quantify potential asymptotic computational advantages offered by neuromorphic computation over conventional computation, which is hampered by the apparent demise of Moore's Law and Dennard Scaling. To obtain as fair a comparison as possible between conventional and neuromorphic computing, we compare neuromorphic algorithms with conventional serial algorithms, rather than conventional parallel algorithms. One could argue that a shared-memory parallel system might also serve as a fair point of comparison, especially given the connections between other circuit models such as *NC* and shared-memory models such as PRAM. However, asymptotically, neuromorphic systems are expected to be more scalable than shared-memory systems and are considered a viable beyond-Moore model of computing. As suggested by Table 2-1 in Section 2.1, the neuron density of even the current infant-generation neuromorphic hardware is promising and suggests that neuron scalability is more akin to logic gates than to general-purpose CPUs. In particular, current generation neuromorphic systems have between 128K and 1M neurons per chip, while comparable CPUs have 8-32 cores per chip. Moreover, as a physical existence proof, adult human brains contain analog neuromorphic circuits with about 100 billion neurons and maximum degree of approximately 10k in the cortex [9]. Moreover, the lightweight communication of neuromorphic systems is expected to allow them to offer a greater degree of parallelizability than conventional distributed-memory parallel systems.

Thus we focus on comparing algorithms executed on a single neuromorphic chip with those executed on a single conventional chip, where both may be aggregated in a similar fashion to form larger parallel systems (see Figure 2-2 in Section 2.1).

Fair comparison of neuromorphic and conventional computing SNNs are qualitatively different enough from conventional computing models that fairly establishing polynomial-factor advantages is challenging. In particular, SNNs do not enjoy a distinction between processing and memory. Our SNN algorithms critically employ both computation and communication to provide asymptotic speedups over the best-known conventional serial counterparts. Our algorithms neuromorphically simulate propagation of information in the input graph. The more closely the SNA's native neural network resembles the input graph, the more efficient our algorithms, as dense neural networks allow for more efficient data movement.

In the design and analysis of algorithms, local data-movement within a CPU is typically treated as $O(1)$ execution-time cost under the usual assumption that any address of a random-access memory (RAM) may be accessed in $O(1)$ time. Although it is arguable whether this assumption is realistic, our goal is simply a fair comparison between neuromorphic and conventional computing. We offer two sets of comparisons. In the first, we compare conventional $O(1)$ RAM algorithms with neuromorphic algorithms that also enjoy $O(1)$ data movement. In this case we assume that we have an SNN over an arbitrary underlying graph G, allowing communication between any two vertices with minimum delay $\delta \in O(1)$. Since, as explained above, we are comparing a single neuromorphic chip to a single CPU (with perhaps $O(1)$ cores), in both cases we are making the assumption that $O(1)$ intra-chip movement is possible for any piece of datum. Our results for this regime are presented in Table 3-1 in Section 3.2.

The more interesting case is when $O(1)$ intra-chip data movement is not deemed plausible, also detailed in Table 3-1. To be fair in this case, we assume a grid-like model of data storage and movement for *both* neuromorphic and conventional algorithms. For the former, we only assume access to SNNs on a grid-like crossbar graph (see Figure 4-1 in Chapter 4). In [5], we give a linear-time embedding algorithm allowing us to simulate SNNs on arbitrary graphs using SNNs on crossbar graphs. Crossbar graphs are commonly supported in neuromorphic hardware, and our embedding scheme has been adapted from similar schemes used in other contexts [11, 36]. The embedding cost is nontrivial, and in the worst case it adds a linear multiplicative factor to the running time of our neuromorphic algorithms. This is because two adjacent nodes in the input graph may have to communicate using a long path in the neuromorphic circuit. Since links in an SNN have delay at least δ, this incurs communication cost proportional to the path length. The embedding cost is conservative since we assume the worst case of embedding a complete SNN directed graph G into a crossbar. It is likely that better embeddings exist for special graph classes of interest.

The grid-like data storage and movement assumption takes a different form for conventional algorithms. We introduce DISTANCE, a data-movement model for conventional algorithms taking into account the distance traveled by data. DISTANCE is motivated by recent observations that data-movement within a CPU's RAM can be energy intensive for conventional systems, while neuromorphic hardware is extremely energy efficient in moving data. Indeed, the standard $O(n^2)$

algorithm for computing a matrix-vector product with an $n \times n$ matrix becomes $O(n^3)$ if data-movement is taken into account in a fashion similar to DISTANCE, while a neuromorphic implementation remains an $O(n^2)$ algorithm [1]. We believe DISTANCE offers a fair comparison to SNNs on crossbars because the data-movement cost in both models is proportional to the l_1 distance traveled in a grid-like setting.

Definition 5: The DISTANCE data-movement model We describe a model that more explicitly accounts for data movement in conventional algorithms, for a fair comparison with neuromorphic algorithms. Suppose we have a memory hierarchy with at least two levels. The smallest, fastest level is made up of *c registers*. Each register holds a single word of $O(\log m)$ bits, where m is the size of the input. Furthermore, any data value needs to be moved to a register for any kind of operation involving that value. We do not distinguish between the lower levels of the hierarchy, which are collectively called *disk*. We generally assume that the data stored in the registers and disk reside on a common 2D plane, and we think of the memory as comprising lattice points in the plane. Each lattice point can hold one word, and some lattice points are registers. In addition, we can decide which lattice points are registers, but the locations of the registers are fixed for the duration of the computation. Even if we assume the data reside on $O(1)$ planes, rather than a single plane, we get lower bounds that are within a constant factor of the ones we derive in this section. In addition, we get non-trivial lower bounds even if we only assume that the data reside in three dimensions.

Lower bounds in the DISTANCE model We detaill the DISTANCE model in [5], and we provide lower bounds for implementations of the best-known shortest-path algorithms within it. These bounds arise from the assumption that memory is laid out in 2D (or perhaps a constant number of 2D layers), which we believe is reasonable for contemporary memory systems. We show that in the DISTANCE model, a conventional algorithm takes $\Omega(m^{3/2})$ time just to read an $O(m)$-sized input, which illustrates the severity of data-movement bottlenecks, and why data-movement-efficient computation, such as neuromorphic computing, is important. Futhermore, future neuromorphic systems may employ analog technologies (such as memristors [1]) or be designed to mitigate data-movement costs (as conventional systems are with memory hierarchies) in order to scale in ways that conventional systems cannot.

3. NEUROMORPHIC GRAPH ALGORITHMS

A fundamental problem in graphs is finding a shortest path between two nodes s and t (an s-t path) in a graph that has non-negative weights associated with its edges. Problems of this type are solved routinely by Google Maps in finding short navigation routes between two places on a map. In this case, the nodes of the graph represent locations of interest on a map, while edges indicate viable direct routes, with each weights indicating traversal times.

Dijkstra's algorithm is an elegant approach to finding shortest paths that is typically one of the first graph algorithms taught in algorithms courses [12, Ch. 24]. Some algorithms that are theoretically efficient are not so in practice, due to large constant factors in execution time. Yet, Dijsktra's algorithm affords efficient practical implementations. Such implementations of it can find an s-t path in a graph with n vertices and m edges in $O(m \log n)$ steps, with each step a primitive computational operation [12]. Implementations with an asymptotic running time of $O(m + n \log n)$ are known, but these do not tend to perform as well in practice, due to large constant factors hidden by the $O()$ notation. In fact, Dijkstra's algorithm solves a more general graph problem: that of finding shortest paths between a specific vertex s and every other vertex in the graph (i.e., a shortest s-v path for every node v). This is known as the single-source shortest paths (SSSP) problem.

A reason that an algorithm seeking to find a shortest path between s and t might indeed need to discover shortest paths between s and every other vertex, including t, is as follows. A shortest path, P between s and t must contain a shortest path between s and every vertex, v_i that is part of the path, P. If there were any shorter path, Q between s and some such v_i, then one would have a shorter path between s and t by going from s to v_i using Q and then continuing from v_i to t as P does. This is known as the optimal substructure property, and is a key ingredient in dynamic programming, as will be discussed in the next section. Thus it is difficult to imagine a shortest s-t path algorithm that does not also solve the SSSP problem in the process.

3.1. Spiking graph algorithm for shortest paths

Dijkstra's algorithm inspires a simple and natural spiking graph algorithm (SGA) to solve this problem, first published in 1991 [2]. As discussed in Section 2.3, our model of spiking neural architectures (SNA) includes programmable spike delay times on links. By this we mean that one may, independently for each link, specify how long a spike should take to traverse it. As communication in spiking architectures is typically asynchronous, delays are a powerful resource spiking algorithm design.

The SGA we describe assumes that the nodes and edges of the input graph can be mapped on the neurons and links of the SNA. One way to relax this assumption is by embedding an input graph into the neuron connectivity graph of an SNA; a detailed discussion of such techniques is beyond

our scope. We program delay times so that the time required for a spike to traverse a link in the SNA is proportional to the weight of the corresponding graph edge. The SGA commences by sending spikes from a source neuron, corresponding to the vertex s, to all of its outgoing neighbors. Every other SNA neuron is programmed to simply propagate the first incoming spike it receives to all its outgoing neighbors. The SGA terminates when every neuron has received a spike, or it can be programmed to terminate early as soon as the neuron corresponding to a particular node of interest, t has received a spike. This is a complete high-level description of an SGA for the SSSP problem.

Figure 3-1 gives a full example of the pseudopolynomial-time neuromorphic version of Dijkstra's algorithm. The shortest path from S to T has length 8, following edges of length $3,2,1,2$ in that order. The shortest S-to-T path with at most three hops has length 9, following edges of length $3,2,4$ in that order.

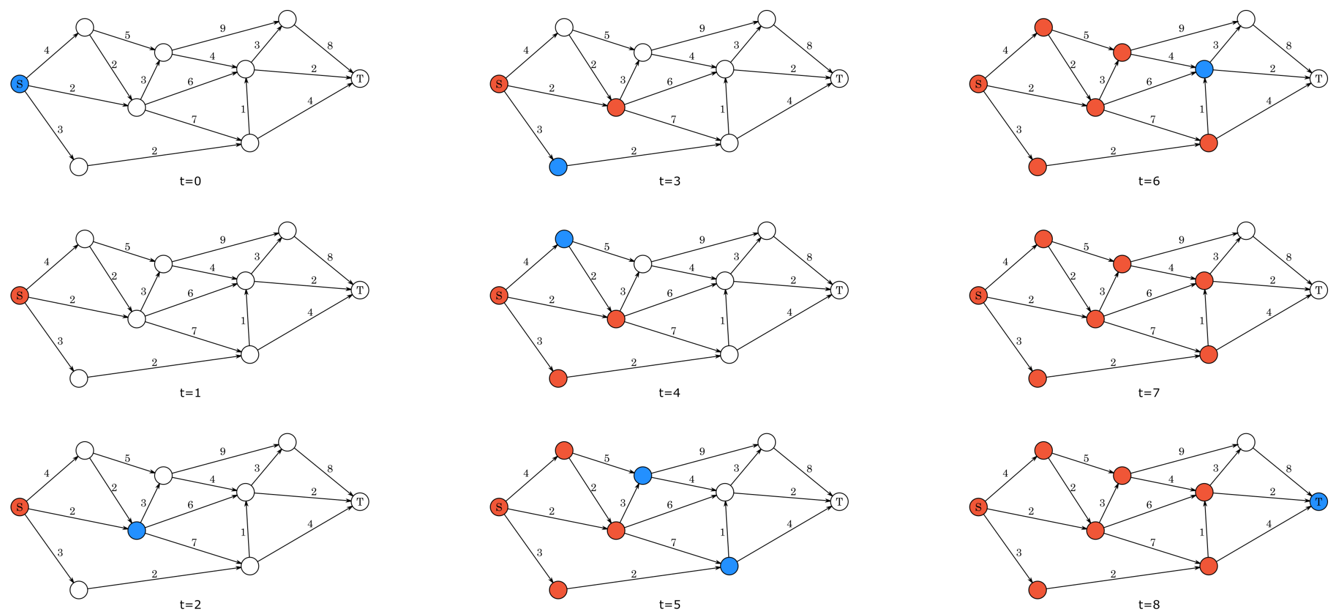

Figure 3-1. An illustration of the neuromorphic algorithm computing a shortest path from source S to sink T. Blue nodes fire a spike. Red nodes have received a spike and will not fire again.

3.2. Our results

We designed two neuromorphic algorithms for the single-source single-sink shortest path problem in graphs with positive integer edge lengths, one of polynomial time complexity and the other of pseudopolynomial time complexity. Our algorithms run in $O((n\alpha + m)\log(nU))$ and $O(nL+m)$ time, where n is the number of vertices in the input graph, m is the number of edges, L is the length of the shortest path from the source to the sink, and α is the number of edges in the shortest path. We also consider the variant where the shortest path must have at most k edges (i.e., k-hop shortest path). In this case, the polynomial-time algorithm run in $O((nk+m)\log(nU))$ time and the pseudopolynomial-time algorithm runs in $O((nL+m)\log k)$ time. Finally, we designed an approximation algorithm for the k-hop variant that runs in $O(kn\log n + m)\log(kU\log n))$ time; this

algorithm uses fewer neurons than the exact algorithms. All of our algorithms in fact work when there are multiple sinks; the single sink just makes the running times simpler to state. In addition, all of our algorithms are based on well-known distributed algorithms; we expect that through straightforward generalizations of our techniques, one can turn any simple distributed algorithm into a neuromorphic algorithm with small overhead.

When comparing the running times of neuromorphic algorithms to those of conventional algorithms, two things must be taken into account: movement cost and embedding cost. The movement cost of a conventional algorithm is the distance that data must travel over the course of the algorithm. Conventional running time analysis usually ignores movement cost or, equivalently, assumes that it is $O(1)$ per operation; one can argue that this is unrealistic, since data probably takes more time to travel longer distances. In any case, we believe that any reasonable analysis of neuromorphic algorithms must take movement cost into account, since spikes by design take a certain amount of time to traverse synapses; thus, it only seems fair to take movement cost into account for conventional algorithms, at least when comparing conventional and neuromorphic algorithms with each other. When we do so, we find that any conventional shortest path algorithm must take $\Omega(m^{3/2}/\sqrt{c})$ time, and the best-known conventional k-hop shortest path algorithm can take $\Omega(km^{3/2}/\sqrt{c})$ time. (Here c is the number of words in the smallest, fastest memory level, and is typically a constant.) Comparing these lower bounds with the running times of our neuromorphic algorithms, we see that for certain ranges of parameters, the neuromorphic algorithms are faster than the best-known conventional algorithms. For example, if L and c are constants, then our pseudopolynomial algorithms run in linear or near-linear time, while the best-known conventional algorithms do not. Finally, we note that even when data movement is ignored for both conventional and neuromorphic algorithms, we obtain similar conditional advantages for neuromorphic algorithms.

The embedding cost of a neuromorphic graph algorithm is incurred because the algorithm needs some way of mapping nodes and edges of the input graph to neurons and synapses in the hardware graph such that distances between nodes are preserved. Such a mapping might take two nodes that are close to each other and send them to two neurons that are far away from each other. Thus when the neuromorphic algorithm runs, the amount of time it takes to send a spike from one neuron to the other could be much more than the time suggested by the distance between the two original nodes. The multiplicative factor increase in running time is the embedding cost (also called dilation). One small issue in defining embedding cost is that the hardware graph is not fixed but varies depending on the exact neuromorphic computing device used. For our purposes, we assumed the hardware graph is a simple grid-like graph that we call the crossbar. We then showed that for general graphs, there is a simple embedding into the crossbar that assigns each vertex of the graph to a unique row and column of the crossbar. However, this embedding has embedding cost $O(n)$. For example, it causes the running time of our pseudopolynomial-time shortest path algorithm to increase from $O(L+m)$ to $O(nL+m)$. More detailed derivation of our results for shortest-path problems appear in [5]. Some of these neuromorphic graph algorithms may be extended to address the dynamic programming problem, which we discuss in the next section. We describe the embedding problem in Chapter 4.

Table 3-1. Comparison of complexities of neuromorphic and conventional methods for Single-Source Shortest Path (SSSP) problems. For this table, we assume that there is a single source node and single destination node. The number of nodes and edges are denoted by n and m, respectively. L is the length of the shortest path with at most k edges (where $k = n-1$ for SSSP); U is an upper bound on the edge lengths; α is the number of edges in the shortest path; and c is the number of words in the smallest, fastest memory level (typically a constant with respect to n and m). The m in the lower bounds can be replaced with the total number of bits in the input, taking edge lengths into account. Lower bounds are computed using the DISTANCE model described in Section 2.3.
The neuromorphic algorithms can offer an asymptotic advantage in all cases except polynomial-time SSSP when ignoring data movement costs. As an example, for the polynomial-time k-hop SSSP neuromorphic algorithms, we obtain a factor $\Omega(k/\log n)$ advantage when ignoring data-movement costs, and a factor $\Omega(m^{1/2}/\log n)$ advantage with data-movement costs, under the reasonable assumptions that U is polynomial in n and $c = O(1)$. The latter can better, depending on the relationship of n, m, and k.

Complexities when taking data-movement costs into account				
Problem	Conservative data-movement lower bound	Data-movement lower bound on best conventional algorithm	Neuromorphic	Neuromorphic is better when:
Polynomial Complexity				
SSSP	$\Omega(m^{3/2}/\sqrt{c})$	$\Omega(m^{3/2}/\sqrt{c})$	$O((n\alpha+m)\log(nU))$	$\log U = O(\log n)$ $c = o(m/\log^2 n)$ $\alpha = o(m^{3/2}/(n\log n\sqrt{c}))$
k-hop SSSP	$\Omega(m^{3/2}/\sqrt{c})$	$\Omega(km^{3/2}/\sqrt{c})$	$O((nk+m)\log(nU))$	$\log U = O(\log n)$ $c = o(m^3/(n^2\log^2 n))$ $c = o(k^2 m/\log^2 n)$
Pseudopolynomial Complexity				
SSSP	$\Omega(m^{3/2}/\sqrt{c})$	$\Omega(m^{3/2}/\sqrt{c})$	$O(nL+m)$	$L = o(m^{3/2}/(n\sqrt{c}))$
k-hop SSSP	$\Omega(m^{3/2}/\sqrt{c})$	$\Omega(km^{3/2}/\sqrt{c})$	$O((nL+m)\log k)$	$L = o(km^{3/2}/(n\sqrt{c}\log k))$

Complexities when ignoring data-movement costs			
Problem	Best-known conventional	Neuromorphic	Neuromorphic is better when:
Polynomial Complexity			
SSSP	$O(m+n\log n)$	$O(m\log(nU))$	never
k-hop SSSP	$O(km)$	$O(m\log(nU))$	$\log(nU) = o(k)$
Pseudopolynomial Complexity			
SSSP	$O(m+n\log n)$	$O(L+m)$	$m, L = o(n\log n)$ and $L = o(m)$
k-hop SSSP	$O(km)$	$O((m+L)\log k)$	$L = o(km/\log k)$ and $k = \omega(1)$

3.3. Dynamic programming

Dynamic programming is a fundamental algorithmic paradigm for solving combinatorial algorithms, with myriad applications across science and engineering; see [6] for details. Dynamic programs are algorithms that run in polynomial time and tend to produce efficient practical implementations. Generally, dynamic programming is a useful approach when a problem can be solved optimally by breaking it into sub-problems and then combining optimal solutions to the sub-problems. It has been adopted in many different application domains. A Google scholar search on "dynamic programming" has 3.2 million hits, including general papers on the topic and specific applications.

Two fundamental concepts underlie dynamic programming: the *optimal substructure property* and *overlapping subproblems*. The optimal substructure property applies when an optimal solution to a problem can be expressed in terms of optimal solutions for smaller subproblems. For instance, the shortest path problem illustrates this property. Consider the problem of finding a shortest path from a source vertex s to a terminal vertex t in a graph. Assume we know that a third vertex v is part of an optimal solution. Then we can reduce the problem to two independent problems: finding a shortest path from s to v and finding a shortest path problem from v to t. We can prove that merging optimal solutions to these two problems gives an optimal solution to original problem of finding a shortest path from s to t. Observe that the optimal substructure property does not apply in many other problems. Consider the path version of the traveling salesperson problem, where we try to find a shortest path from s to t that visits each vertex once. This time, we cannot break the problem into two independent subproblems finding s to v and v to t, since it is not clear whether the remaining vertices are part of the first subproblem or the second subproblem.

The optimal substructure property allows us to compute the optimal solution recursively by breaking the problem down into smaller problems. Recursively investigating many solutions leads to many subproblems being investigated multiple times. The trick in dynamic programming is to define this space of overlapping subproblems and avoid solving them more than once. Subsequently, the efficiency of dynamic programming depends on the number of subproblems that need to be solved. In practice, we avoid recursion in implementations of dynamic programming algorithms by maintaining a table that stores values of optimal solutions to subproblems, and filling in the entries of this table in a bottom-up fashion. That is, we compute the solution to a subproblem when all the subproblems it depends on have been solved.

In [6] we propose a generalized approach to solving dynamic programming problems using a generalization of our neuromorphic graph algorithms for shortest-path problems. In our approach, each subproblem is represented by a neuron or a group of neurons and neurons are connected if the solution of one is affected by the other. Neurons communicate the optimal value of their respective subproblems by their firing times. In some cases neurons fire multiple times for maximization problems, as better solutions become available.

With this neural-network structure, each neuron has enough information from optimal subproblems to compute its own solution value. The challenge is in how to compute this value given values of optimal solutions to the subproblems. We describe generic solutions to a broad class of problems, based on the structure of the recursive formulation in the dynamic programming solution.

Specifically, we describe generic solutions for constrained and unconstrained versions of maximization and minimization problems as long as the optimal solution value depends only on the minimum/maximum of subproblems. At a high level, we define dynamic programming problems in terms of graph constructions and explain how to solve them neuromorphically. General constructions for dynamic programming as well as a novel specialized neuromorphic implementation of a well-known dynamic program for the longest increasing subsequence problem are presented in [6].

4. EMBEDDINGS

To offer a fair an assessment of neuromorphic graph algorithms, we do not generally assume that the underlying neuromorphic architecture has the same connectivity among its neurons as the input graphs for our problems. If this were the case, then our neuromorphic algorithms would be able to offer a greater advantage, corresponding to the non-data-movement complexities of Table 3-1 rather than the data-movement complexities. Thus we do not assume that the neuromorphic platform at hand will offer a favorable neuron topology, yet we must assume that some concrete topology is offered.

The *crossbar* is a natural candidate for a standard neuromorphic topology that we may assume any platform offers, as it arises naturally and has served a similar role in other contexts, including quantum annealing [11,36]. In [5] we adapt crossbar embedding ideas previously used in these contexts to demonstrate how an arbitrary neuron topology can be mapped to a crossbar, at the expense of requiring additional neurons and links. We include a precise description and small example of a crossbar below.

Let $[n] = \{1, \ldots, n\}$. We use H_n to denote the *crossbar* of order n. The graph H_n is directed and is defined as follows. The vertex set of H_n is

$$\{v^-_{ij} \mid i, j \in [n]\} \cup \{v^+_{ij} \mid i, j \in [n]\}.$$

The edge set of H_n is the union of the following sets

1. $\{v^-_{ii} v^+_{ii} \mid i \in [n]\}$
2. $\{v^+_{ij} v^-_{ij} \mid i \neq j; i, j \in [n]\}$
3. $\{v^+_{ij} v^+_{i(j+1)} \mid i \leq j; i, j \in [n-1]\}$
4. $\{v^+_{i(j+1)} v^+_{ij} \mid i > j; i, j \in [n]\}$
5. $\{v^-_{ij} v^-_{(i+1)j} \mid i < j; i, j \in [n]\}$
6. $\{v^-_{(i+1)j} v^-_{ij} \mid i \geq j; i, j \in [n-1]\}$

See Figure 4-1 for an example where $n = 3$.

In [5], we give a linear-time algorithm to embed any n-vertex input graph G into the crossbar H_n. We show how to assign delays to the edges in H_n such that finding single-source shortest paths (SSSP) in G is equivalent to finding SSSP in H_n. For simplicity, we just show how to embed the complete graph K_n, which can simulate an arbitrary graph by setting some edge delays to infinity or otherwise disabling edges.

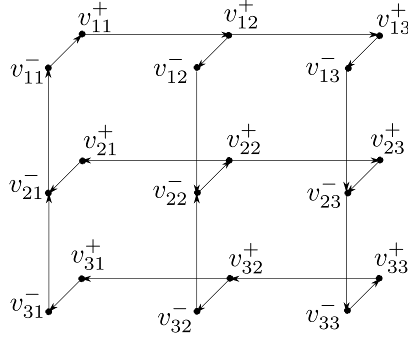

Figure 4-1. The stacked grid (or crossbar) H_3. The general H_n is a topology we may reasonably expect as a subset of every neuromorphic architecture.

It can be shown that if in the embedding we require each vertex to be assigned a unique row and column, then $O(n)$ dilation is in fact optimal, even for stars. However, if we relax this requirement, then results from the VLSI literature imply that we can embed graphs with small separators while only incurring sublinear dilation, though the embeddings are slightly more complicated. For example, planar graphs can be embedded with $O(\sqrt{n}\log n)$ dilation, and trees can be embedded with $O(\sqrt{n})$ dilation. On the negative side, there exist bounded-degree graphs (in fact, degree-three graphs) called superconcentrators that require at least linear dilation, even if we do not require that each vertex be assigned a unique row and column.

The embedding results in the previous paragraph assume that we are embedding a graph into the crossbar. In fact the same results apply when we are trying to embed into a two-dimensional grid; we can show that any embedding into a crossbar can be converted into an embedding into a grid with constant-factor multiplicative overhead in each of the two dimensions, using planarizing gadgets.

The embedding results so far have dealt with mapping the input graph into the hardware graph, which we have assumed to be a crossbar or grid. This is often a useful step in solving graph problems neuromorphically. For more general problems, though, one might instead want to view an arbitrary spiking neural network as a graph and embed it into the hardware graph. In this case, the previously mentioned embedding results still apply, up to polylogarithmic factors, if all synaptic weights are integers and all synaptic delays are bounded by a constant. The basic idea is to first convert such a spiking neural network into an equivalent Boolean circuit of bounded fan-in and fan-out with only polylogarithmic overhead; one can then directly apply the previous embedding results to the Boolean circuit (by again viewing the Boolean circuit as a graph).

Although a crossbar is a reasonable and useful abstraction of a generic neuromorphic architecture, platform-specific constraints and features can impose additional challenges as well as opportunities for further optimization. In the next section we consider hardness results for embedding problems in a model of a more realistic architecture.

4.1. Hardness results for embedding in a Loihi-like architecture

In this section, we formalize a problem motivated by mapping graph algorithms to an abstraction of the Intel Loihi chip architecture (see Section 2.1) and sketch a proof that the problem is at least weakly NP-complete. NP-completeness is a notion of formal intractability. If a problem is weakly NP-complete, the intractability depends upon numerical parameters for the problem. If a problem is strongly NP-complete the intractability does not depend upon numerical value (only structural size). In practice, if a problem is strongly NP-complete, we should not try to find a provably efficient algorithm for this problem, one where the running time is a polynomial function of the input size for all possible inputs. Instead, we should seek algorithms that provably approximate the best embedding quickly, or heuristics (unproven methods) that work well in practice. If a problem is weakly NP-complete, it is still possible to find efficient algorithms if the numerical parameters for an instance are small.

The Loihi Model We first formalize a model of the Loihi architecture following the description in Davies et al. [13].

As we have throughout this report, we represent a specific computation we wish to run using a graph where the nodes represent neurons and (directed) edges represent connections with spiking communications. To use the Loihi architecture for an algorithm represented by such a graph, we need to map (assign) the neurons to elements of the architecture obeying constraints for communication.

The Intel Pohoiki Springs architecture consists of many Loihi chips. Each chip has a fixed capacity for neurons. Each chip has a fixed number of inputs and outputs, 4096 each in the Davies et al. description [13]. We define a *net* to be a set of (directed) edges with the same source and the same weight in the graph representing application neuron connectivity. That is, it is a set of (outbound) neighbors U for a node v that all weight a spike from v the same way. If any subnet $U' \subseteq U$ is assigned to a particular chip, they can all share the same chip input. The (weighted) spike is routed to all nodes in U'. There is no direct on-chip routing, however. If node v sends a spike to node u, even if both node v and u are on the same chip, the spike must exit the chip, consuming an output, and re-enter the chip, consuming an input. The number of outputs consumed by a node v depends upon where its neighbors are mapped. To formalize, partition the neighbors of node v, into nets $U_1, U_2, \ldots, U_{n(v)}$, where $n(v)$ is the number of nets for node v. Let $H(U_i)$ be the set of chips that contain at least one node in U_i. Then the number of outputs required for node v is $\sum_{i=1}^{n_v} |H(U_i)|$.

The Loihi Embedding Problem We are given a weighted graph $G = (V, E)$ representing the graph associated with an instance of a graph algorithm we wish to run on a Loihi-like neuromorphic system. We are given C "containers," and bounds B_{in} on the number of inputs to a container and B_{out} on the number of outputs from a container. We are also given a bound on container capacity k.

The problem is to place each node $v \in V$ into exactly one of the containers (partition the nodes) such that:

- The total number of nodes assigned to any given container is at most k.
- For any given container, c_k, the total number of outgoing nets for all nodes assigned to c_k is at most B_{out}. That is, $\sum_{v \in c_k} \sum_{i=1}^{n_v} |H(U_i)| \leq B_{\text{out}}$.
- For any given container, c_k, the total number of incoming nets (over all nodes assigned to c_k) is at most B_{in}. Nets are defined locally, based on the incoming edges for each node. If any subset of incoming edges for nodes in container c_k have the same source and the same weight, they count as one net.

For now, this is just a feasibility problem.

We now sketch a proof that our embedding problem is weakly NP-complete.

We will do a reduction from the 3-partition problem. The input to the 3-partition problem is a multiset S of $n = 3m$ positive integers $a_1, a_2, \ldots, a_n$. The sum of all n integers is mT. The output is whether or not there exists a partition of S into m triplets $S_1, S_2, \ldots, S_m$ such that the sum of the numbers in each triplet is equal to T. The triplets partition the multiset S. That is, every element of S is in exactly one of the triplets.

Let the capacity k of each chip be T in a system with $m+1$ chips. The number of outputs for a chip is $n+1$. Our graph is a root node with $(m+1)T-1$ children. For each a_i, for $i = 1, 2, \ldots, n$, there are a_i children of the root that are each connected to the root with an edge of weight i. These sets account for mT of the children of the root. The remaining $T-1$ "filler" children are all connected to the root with an edge of weight $n+1$. If there is a 3-partion of the set S, there is only one feasible placement of the nodes into containers. We must place the root node and the $T-1$ filler nodes together in one container. Then we must partition the other nodes into m groups of T nodes. To be feasible, all the nodes in the group associated with input a_i must be in the same container. Otherwise, there will be too many required outputs for the container holding the root node.

This reduction only give a weakly-NP result because we can represent each integer a_i with $\log a_i$ bits. But we add a_i nodes to the graph, thus "blowing up" the representation size exponentially.

With our theoretical models and algorithms in place, we now turn to experimental studies to validate the former.

5. EXPERIMENTS

We describe experiments conducted on Intel's Pohoiki Springs platform, which consists of Nahuku boards with Loihi chips (e.g., see Section 2.1). Our goal in conducting such experiments is to identify and attempt to understand hardware considerations that may deviate from the theoretical assumptions we enacted and models we developed. In this spirit, we select our simplest neuromorphic graph algorithm, that for shortest paths in an unweighted graph (i.e., breadth-first search), and bypass embedding complications by considering input graph instances that readily map into the native Pohiki Springs topology. Theoretically, we expect to see linear scaling in such a scenario and we seek to verify this experimentally.

5.1. Hardware limitations

Although Intel's Pohoiki Springs platform currently offers around hundrend million neurons, in practice, it is difficult properly utilize the full capabilities of the hardware. As an example, consider the spiking algorithm for shortest paths. In order to extract meaningful information from the neurons (e.g. the shortest path lengths) we need to be able to know at what time step the target neuron fires. Most neuromorphic platforms, including Intel's, offer some kind of *probing* facilities. The obvious way to track neuron activity would be to use a spike probe that keeps track of when a particular neuron fires at a given time step. However, using such a probe may not be practical depending on the architecture. For example, Loihi's spike probes introduce significant overhead on the runtime of the network. Additionally, this kind of precise spike probe may not be a universal capability for all neuromorphic architectures.

Despite this, there may be possible workarounds depending on the features of the architecture. In the case of the spike-probe issue described, if a neuromorphic platform has a potential probe or some other way of measuring a neuron's final potential with minimal overhead, then it would be possible to determine the spike count of a neuron while avoiding the overhead associated with using a spike probe. To illustrate this, suppose we want to determine how many times neuron x fires while avoid using a costly spike probe. We can accomplish this by using an additional neuron p_x that has no decay, initial potential of 0 and a threshold of 1 (see Section 2.2 for the neuronal model). We then attach a synapse going from x to p_x where the delay is 0 and the weight is -1. Whenever x fires, p_x accumulates more negative potential. Therefore, if we then measure p_x's potential, then we can exactly calculate the number of times p_x has fired. We note that this construction could be modified so that we use p_x's decay function to determine at what time step x fires (assuming x spikes only once). Although our proposed workaround allows us to avoid the overhead associated with using a spike probe, it does limit the number of available neurons. More sophisticated schemes for counting spikes are studied in [19].

The spike-probe scenario described above shows that there are current hardware limitations that make benchmarking rather difficult to perform and assess. Furthermore, although it may be possible to work around the spike-probe (and other) hardware limitations/issues, these workarounds may introduce key tradeoffs (e.g. run time vs number of usable neurons).

5.2. Results

5.2.1. Probed vs unprobed

The first experiment we present focuses on illustrating the performance overhead of using spike probes. We ran our neuromoprhic shortest path algorithm on a Pohoiki Springs system. For this experiment we used unweighted 3-dimensional directed lattice graphs where the source is the bottom, left-most node, the target is the upper, right-most node, and the edges are oriented away from the source. Each node is represented as a neuron (each neuron has a threshold of 1, initial potential of 0, and no decay) on the system, and each edge corresponds with a synapse (with weight 1 and delay 0). In this experiment we can clearly see that using spike probes incurs an order-of-magnitude run-time increase. See Figure 5-1 for details. Due to these results, later experiemnts run unprobed after they have been verified on smaller instances.

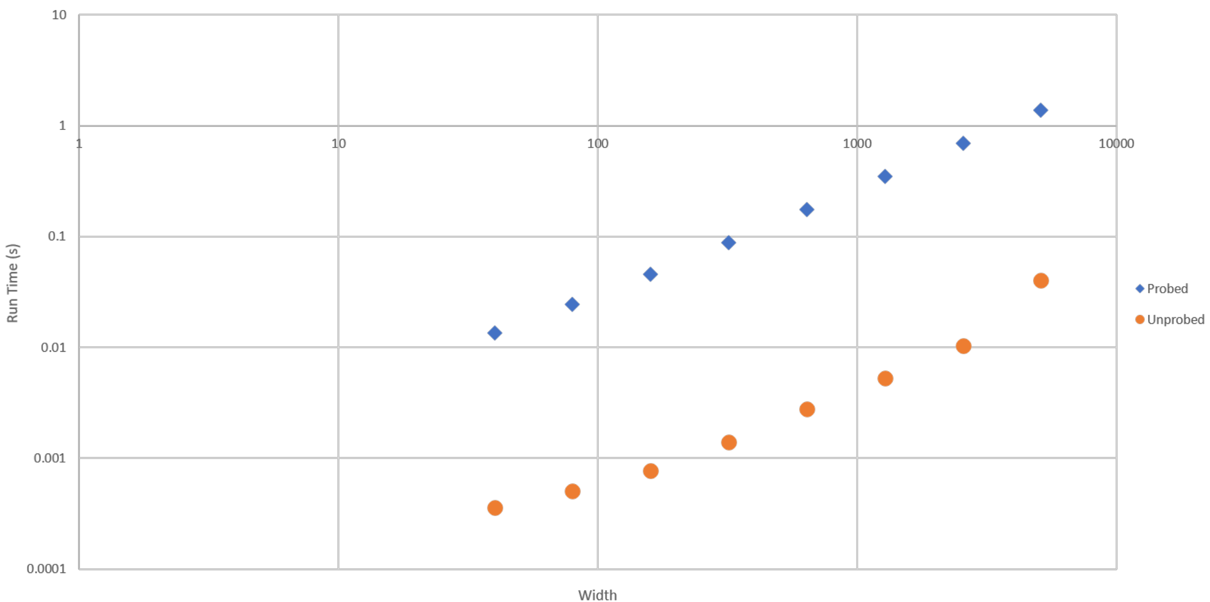

Figure 5-1. Chart that shows run time (s) as a function of the Width of the input grid graph. It is cleared that using spike probes incurs an order-of-magnitude run-time overhead.

5.2.2. *Scaling*

The next set of experiments aim at exposing the scaling behavior of our neuromorphic shortest path algorithm. These experiments were conducted on directed layer graphs where each layer is fully connected with the subsequent layer. The source node(s) reside in the first layer while the target nodes reside in the last layer. The neuron and synapse configuration is identical to the previous experiment. These experiments were conducted using Intel's Pohoiki Springs platform as well.

The first experiment looked at layer graphs with 5 nodes in each layer (results summarized in Figure 5-2). The second experiment looked at layer graphs with 64 nodes in each layer, thereby saturating the synaptic fan-in of subsequent layers (see Figure 5-3 for results).

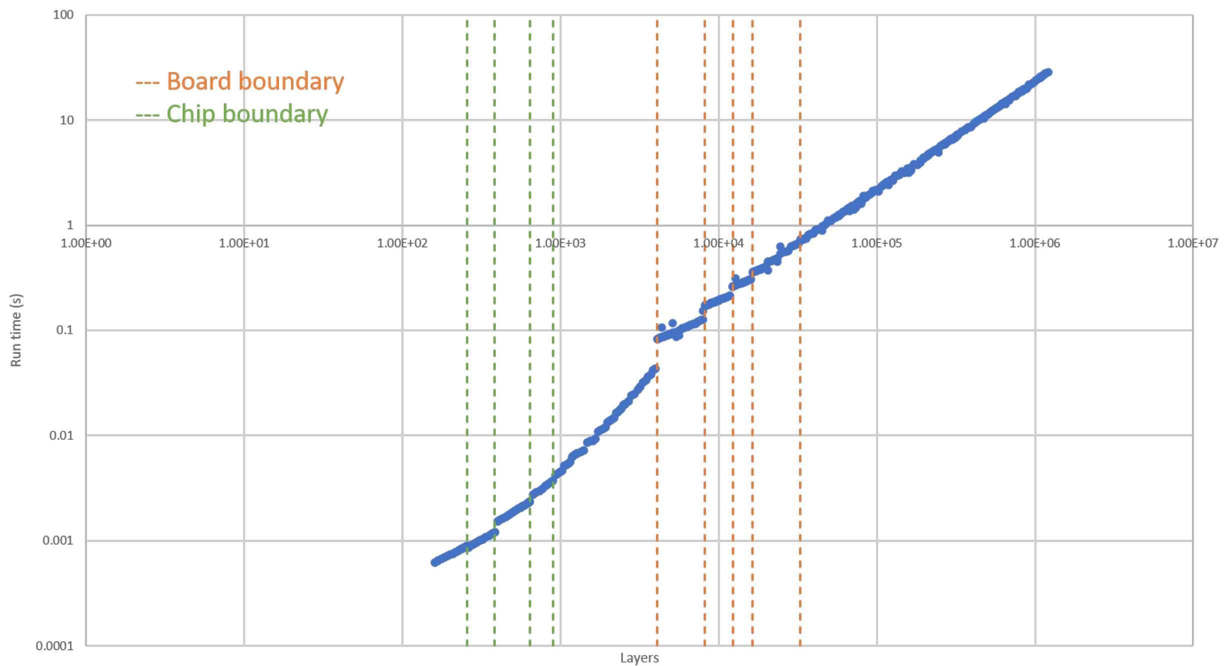

Figure 5-2. Experiment looking at how our shortest path algorithm scales on layer graphs. Although asymptoically the run time grows linearly, we note that there are jumps in run time that occur at chip and board core limits, i.e. when the number of cores needed to embed the layer graphs excceed a single chip/board.

In Figures 5-2 and 5-3 we can see the scaling behavior is linear with respect to the number of layers. However, we observed in Figure 5-2 that there were "jumps" or apparent discontinuities in run time as the number of layers increased. Looking further we noticed that these jumps occurred right at the boundary of the chip and board core assignment limits. More specifically, we used a simple embedding heuristic: each layer is assigned its own core and when the number of unique cores is reached, then core assignments loop back around to the first core. Therefore, the number of layers serves as an approximation to the number of cores utilized by a given input layer graph. Taking this into account, one hypothesis on why we encounter these jumps in run time is that chip-to-chip and board-to-board communication incurs more overhead than intra-core and intra-board

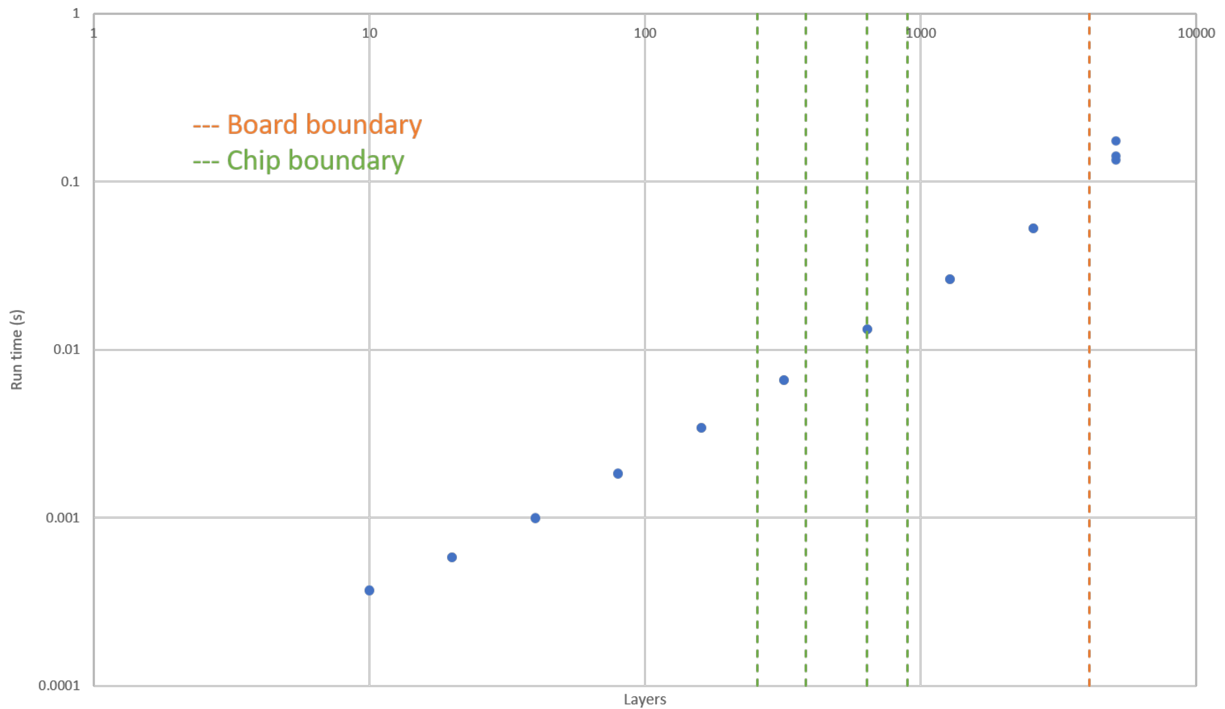

Figure 5-3. Experiment looking at how our shortest path algorithm scales on layer graphs with saturated edges between layers, thereby producing a neural network where the synapse fan-in limit for each layer is met. We note that the chip and board boundary jumps are not present.

communications. However, when we saturated the synapse fan-in (see Figure 5-3), we note that such jumps are not evident. We leave a fuller understanding of this phenomenon as a topic for future research.

6. CONCLUSIONS

We described recent work in developing the first neuromorphic graph algorithms with provable asymptotic advantages over conventional counterparts. To make such comparisons fair, we reconsidered data movement in conventional computing systems in light of our data-movement-critical neuromorphic algorithms. We described experiments to validate our theoretical findings and discovered that, at least for basic and simple cases, we can realize the anticipated scaling in certain regimes; however, several architectural considerations need to be taken into account for broader scalability.

Our work leaves several avenues for future research. Developing more sophisticated neuromorphic algorithms for other graph problems remains the biggest challenge for future work. One may have to be judicious in looking for conventional algorithms to inspire efficient neuromorphic algorithms. Effectively embedding input instances to match hardware topology becomes a major bottleneck in realizing neuromorphic wins. We have adapted from previous work a simple scheme that enables embedding any input graph into a crossbar topology. A crossbar is a "lowest-common-denominator" in the sense that we expect all current and future platforms to support efficient crossbar connectivity. However, our embedding scheme incurs a resource overhead that is likely far from optimal for real-world instances (as opposed to arbitrary instances). New embedding heuristics as well as algorithms with provable bounds on resource overheads are critical for reaping neuromorphic benefits in practical settings.

REFERENCES

[1] Sapan Agarwal, Tu-Thach Quach, Ojas Parekh, Alexander H Hsia, Erik P DeBenedictis, Conrad D James, Matthew J Marinella, and James B Aimone. Energy scaling advantages of resistive memory crossbar based computation and its application to sparse coding. *Frontiers in neuroscience*, 9, 2015.

[2] R. Aibara, Y. Mitsui, and T. Ae. A CMOS chip design of binary neural network with delayed synapses. In *1991., IEEE International Sympoisum on Circuits and Systems*, pages 1307–1310 vol.3, June 1991.

[3] James B Aimone. Neural algorithms and computing beyond moore's law. *Communications of the ACM*, 62(4):110–110, 2019.

[4] James B. Aimone, Yang Ho, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Yipu Wang. Provable neuromorphic advantages for computing shortest paths. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, pages 497–499, New York, NY, USA, 2020. Association for Computing Machinery.

[5] James B. Aimone, Yang Ho, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Yipu Wang. Provable advantages for graph algorithms in spiking neural networks. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, pages 35–47, New York, NY, USA, 2021. Association for Computing Machinery.

[6] James B. Aimone, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Helen Xu. Dynamic programming with spiking neural computing. In *Proceedings of the International Conference on Neuromorphic Systems*, ICONS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Abdullahi Ali and Johan Kwisthout. A spiking neural algorithm for the network flow problem, 2019.

[8] Arnon Amir, Brian Taba, David J Berg, Timothy Melano, Jeffrey L McKinstry, Carmelo Di Nolfo, Tapan K Nayak, Alexander Andreopoulos, Guillaume Garreau, Marcela Mendoza, et al. A low power, fully event-based gesture recognition system. In *CVPR*, pages 7388–7397, 2017.

[9] Frederico A. C. Azevedo, Ludmila R. B. Carvalho, Lea T. Grinberg, José Marcelo Farfel, Renata E. L. Ferretti, Renata E. P. Leite, Wilson Jacob Filho, Roberto Lent, and Suzana Herculano-Houzel. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *The Journal of Comparative Neurology*, 513(5):532–541, April 2009.

[10] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla, and Kwabena Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[11] Vicky Choi. Minor-embedding in adiabatic quantum computation: I. The parameter setting problem. *Quantum Information Processing*, 7(5):193–209, October 2008.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[13] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[14] Merrick Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, December 1984.

[15] Mohsen Ghaffari and Jason Li. New distributed algorithms in almost mixing time via transformations from parallel algorithms, 2018.

[16] Kathleen E. Hamilton, Tiffany M. Mintz, and Catherine D. Schuman. Spike-based primitives for graph algorithms, 2019.

[17] Demis Hassabis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick. Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258, 2017.

[18] Yael Hitron, Cameron Musco, and Merav Parter. Spiking neural networks through the lens of streaming algorithms. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[19] Yael Hitron and Merav Parter. Counting to Ten with Two Fingers: Compressed Counting with Spiking Neurons. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 57:1–57:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[20] Yael Hitron, Merav Parter, and Gur Perri. The Computational Cost of Asynchronous Neural Communication. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, volume 151 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:47, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[21] Sebastian Höppner, Johannes Partzsch, Christian Mayr, and Steve Furber. Spinnaker2 - an energy efficient realtime neuromorphic compute system in 22fdx technology. `https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-37-98/Sebastian-Hoppner-_2D00_-Neural-Networks-.pdf`.

[22] Intel core i7-9700t specifications. `https://www.intel.com/content/www/us/en/products/processors/core/i7-processors/i7-9700t.html`.

[23] Intel scales neuromorphic research system to 100 million neurons. https://newsroom.intel.com/news/intel-scales-neuromorphic-research-system-100-million-neurons.

[24] Muhammad Mukaram Khan, David R Lester, Luis A Plana, A Rast, Xin Jin, Eustace Painkras, and Stephen B Furber. Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2849–2856. Ieee, 2008.

[25] Johan Kwisthout and Nils Donselaar. On the computational power and complexity of Spiking Neural Networks. *arXiv:2001.08439 [cs]*, January 2020. arXiv: 2001.08439.

[26] W. Maass. Lower bounds for the computational power of networks of spiking neurons. *Neural Computation*, 8(1):1–40, 1996.

[27] Christian Mayr, Sebastian Hoeppner, and Steve Furber. Spinnaker 2: A 10 million core processor system for brain simulation and machine learning, 2019.

[28] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[29] Samuel Moore. Intel's Neuromorphic System Hits 8 Million Neurons, 100 Million Coming by 2020 - IEEE Spectrum. `https://spectrum.ieee.org/tech-talk/artificial-intelligence/embedded-ai/intels-neuromorphic-system-hits-8-million-neurons-100-million-coming-by`

[30] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 565–573, 2014.

[31] Eustace Painkras, Luis A Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R Lester, Andrew D Brown, and Steve B Furber. Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, 48(8):1943–1953, 2013.

[32] Ojas Parekh, Cynthia A Phillips, Conrad D James, and James B Aimone. Constant-depth and subcubic-size threshold circuits for matrix multiplication. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 67–76. ACM, 2018.

[33] Johannes Schemmel, Daniel Briiderle, Andreas Griibl, Matthias Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1947–1950. IEEE, 2010.

[34] Catherine D Schuman, Thomas E Potok, Robert M Patton, J Douglas Birdwell, Mark E Dean, Garrett S Rose, and James S Plank. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*, 2017.

[35] Evangelos Stromatias, Francesco Galluppi, Cameron Patterson, and Steve Furber. Power analysis of large-scale, real-time neural networks on spinnaker. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2013.

[36] C. D. Thompson. Area-time complexity for VLSI. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, STOC '79, pages 81–88, Atlanta, Georgia, USA, April 1979. Association for Computing Machinery.

[37] John Von Neumann. *The computer and the brain*. Yale University Press, 2012.

[38] Jiří Šíma and Pekka Orponen. General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Comput.*, 15(12):2727–2778, December 2003.

[39] Andrew C. C. Yao. Separating the polynomial-time hierarchy by oracles. In *, 26th Annual Symposium on Foundations of Computer Science, 1985*, pages 1–10, October 1985.

DISTRIBUTION

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

Sandia
National
Laboratories