SAND2021-6379C

Provable Advantages for Graph Algorithms in Spiking Neural Networks

James B. Aimone, Yang Ho, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, Yipu Wang
Sandia National Laboratories, USA
{jbaimon,yho,odparek,caphill,apinar,wmsever,yipwang}@sandia.gov

ABSTRACT

We present a theoretical framework for designing and assessing the performance of algorithms executing in networks consisting 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 neuromorphic graph algorithms, focusing on shortest path problems. Our neuromorphic algorithms are message-passing algorithms relying critically on data movement for computation, and we develop data-movement lower bounds for conventional algorithms. A fair and rigorous comparison with conventional algorithms and architectures is challenging but paramount. We prove a polynomial-factor advantage even when we assume an 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 provable asymptotic computational advantage for neuromorphic computing.

CCS CONCEPTS

• **Theory of computation** → **Shortest paths**; • **Hardware** → **Neural systems**; • **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

Neuromorphic computing, neuromorphic graph algorithms, neuromorphic complexity.

ACM Reference Format:
James B. Aimone, Yang Ho, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, Yipu Wang . 2021. Provable Advantages for Graph Algorithms in Spiking Neural Networks. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3409964.3461784

1 INTRODUCTION

The brain has been proposed as a potential inspiration for parallel computing since the earliest days of computer science [40]. 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, [10]), and academic efforts including SpiNNaker [24], NeuroGrid [8], and BrainScales [34]. 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 3 in Appendix A. Neuromorphic systems with 100 million total neurons are currently available [22, 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, 16, 35], 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 [18], Hamilton, Mintz, & Schuman [15], Ali & Kwisthout [5], Hitron, Parter, & Perri [19], and Hitron, Musco, & Parter [17].

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 [12, 41, 42]. 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 [41].

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 2.2). 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 results are summarized in Table 1 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.

Organization. Section 2 introduces neuromorphic-computing fundamentals, including considerations for fair comparison to conventional computing. Section 3 illustrates a simple, natural neuromorphic algorithm for single-source shortest paths. Section 4 presents our neuromorphic algorithms for k-hop single-source shortest paths. Section 5 gives detailed implementations of neuromorphic circuits necessary for our neuromorphic algorithms. Section 6 gives data-movement-based lower bounds for conventional algorithms. Section 7 presents a neuromorphic adaptation of a distributed CONGEST algorithm for approximate k-hop single-source shortest paths. Appendix A overviews current neuromorphic-computing systems and compares them to a conventional CPU.

2 NEUROMORPHIC COMPUTING FUNDAMENTALS

2.1 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 3). 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 $t \le 0$. 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 $t \ge 1$, 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. (1) and (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)), and if so the voltage

Table 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 that we introduce in Section 2 and consider in more detail in Section 6.
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	$\Omega(m^{3/2}/\sqrt{c})$	$\Omega(m^{3/2}/\sqrt{c})$	$O((n\alpha + m)\log(nU))$	$\log U = O(\log n)$, $c = o(m/\log^2 n)$, and $\alpha = o(m^{3/2}/(n\log n\sqrt{c}))$
k-hop SSSP	$\Omega(m^{3/2}/\sqrt{c})$	$\Omega(km^{3/2}/\sqrt{c})$	$O((nk + m)\log(nU))$	$\log U = O(\log n)$, $c = o(m^3/(n^2\log^2 n))$, and $c = o(k^2m/\log^2 n)$
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	$O(km)$	$O((m + L)\log k)$	$L = o(km/\log k)$ & $k = \omega(1)$

resets according to Eq. (2.1). The dynamics for a LIF neuron are:

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

$$f(t+1) := \begin{cases} 1, & \text{if } \hat{v}(t+1) > v_{\text{threshold}} \\ 0, & \text{if } \hat{v}(t+1) \leq v_{\text{threshold}} \end{cases} \tag{2}$$

$$\text{if } f(t+1) = 1 \text{ then } v(t+1) := v_{\text{reset}} \tag{3}$$
$$\text{else } v(t+1) := \hat{v}(t+1)$$

$$v_{\text{syn}}(t) := \sum_{i \in N} (f_i(t - d_{ij}) \cdot w_{ij}). \tag{4}$$

2.2 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 2.1. 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 2.1. 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.

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

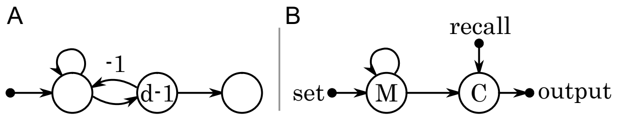

Figure 1: Circuit (A) uses neurons to simulate an $O(d)$ synaptic delay on neuromorphic architectures that do not natively support such delays. When the first neuron activates, its feedback loop causes it to repeatedly fire until the second neuron receives $d - 1$ spikes. When the second neuron fires, 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).

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 2.1. 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 adapt his algorithm to an SNN algorithm in Section 7. For the sake of completeness, we mention that Ghaffari and Li [13] have described a $(1 + \epsilon)$-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 1).

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 1).

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 discuss threshold-gate implementations of such circuits, which may be viewed as non-recurrent spiking neural networks, in Section 5.

2.3 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 3 in Appendix A, 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 [7]. 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 7 in Appendix A).

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 1 in Section 1.

The more interesting case is when $O(1)$ intra-chip data movement is not deemed plausible, also detailed in Table 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 2). In Section 4.4, 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 [9, 39]. 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 consider the DISTANCE model in more detail in Section 6, 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. Theorem 6.1 shows 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 PSEUDOPOLYNOMIAL-TIME SPIKING SSSP ALGORITHM

We now summarize a previous natural neuromorphic graph algorithm (NGA) for single-source shortest paths (SSSP) in graphs with positive edge weights. It is based on Dijkstra's classic algorithm which has serial complexity $O(m+n\log n)$ for a graph with n nodes and m edges. Aibara et al. first published the spiking algorithm in 1991 [2]. Aimone et al. gave a more recent version for our model of spiking neural architectures (SNAs) [4]. This illustrates some of the ideas we use in the k-hop-constrained shortest-paths circuits in a simpler setting.

The NGA gives each synapse (u, v) a delay proportional to the length of the corresponding graph edge. At the start, the neuron corresponding to the source vertex v_s sends a spike to each neighbor. Every other neuron propagates only the first incoming spike it receives (ties are fine) to all its outgoing neighbors. The NGA terminates when the neuron corresponding to the destination node receives a spike (or when every neuron has received a spike, if we want to find distances from v_s to all other nodes).

Because synapse delays are proportional to corresponding graph-edge lengths, a spike that arrives at a node v at time t corresponds to a path from source v_s to node v of length t. Thus the spike timing mimics the priority queue in Dijkstra's algorithm. Aimone et al. [4] give the NGA's complexity as $O(L+m+n)$, where L is the length of the shortest v_s-to-v_t path. The $O(m+n)$ term is for set up and read out. The NGA is better than the classical algorithm when paths are short compared to the graph size. Aimone et al. describe how to infer shortest paths rather than just the length of the shortest path. Each node needs to remember a neighbor that sends the first spike. Each node has a unique ID from 0 to $n-1$. When node v receives its first spike from node u, it sends a binary encoding of its ID to its neighbors, and "latches" (remembers) the ID u.

4 NEUROMORPHIC ALGORITHMS FOR k-HOP SHORTEST PATH

In this section we assume the input graph is mapped to the SNA's native hardware network. In Section 4.4 we will discuss a technique to construct such a mapping and assess its effect on performance.

4.1 Pseudopolynomial-time algorithm

We wish to find the shortest path from the source node v_s to the destination node v_t (or the shortest paths from v_s to every other node) but only consider paths that traverse at most k edges (k hops). The NGA starts with the same structure as the regular SSSP NGA. Each synapse has unit weight and delay proportional to the length of the edge it represents; each neuron has initial voltage 0, unit threshold voltage, and zero decay. However, instead of a node u sending a single spike to a neighbor v, it sends $\lceil\log k\rceil$ spikes that encode a *time to live (TTL)*. At the start, the node corresponding to v_s sends $\lceil\log k\rceil$ spikes to each neighbor encoding the value $k-1$. If a node v receives a spike message encoding the value k' at time t, then there is a path from source v_s to node v of length t that traverses $k-k'$ edges.

For each non-source node v, the length of the shortest k-hop-constrained path from v_s (if it exists) is still the time the first spike arrives. Spikes that arrive later, representing longer paths, could have traversed fewer edges and have a longer TTL. These could propagate further in the graph than previous paths with shorter TTL. So node v can propagate (spike) multiple times. If multiple spikes arrive at the same time, the one with the largest TTL dominates the others as a building block for other k-hop-constrained shortest paths. Thus the circuit computes the largest TTL k' from any of the incoming spikes, and sends a spike encoding $k'-1$ to all its neighbors if $k' \geq 1$.

One circuit to subtract 1 from a $\lceil\log k\rceil$-bit number x is to use the (fixed) 2's complement of the number 1 (which is $\lceil\log k\rceil$ ones) and add it to x. We can chain constant-depth parity circuits for two or three bits and threshold gates for the carry bit to do the addition in $O(\log k)$ depth with $O(\log k)$ neurons. Alternatively, there are depth-3 circuits with a polynomial number of neurons [14]. See Section 5 for the maximum circuits. To use $O(\log k)$-depth circuits, we must scale all graph edges so that the minimum edge length is at least $\lceil\log k\rceil$. This increases the running time by an $O(\log k)$ factor.

Performance. Aside from loading the graph, which should cost time $O(m)$, the algorithm's running time depends upon the circuits for finding the max and subtracting one. If time is most important, than the algorithm can run in $O(L)$ time using $O(m(\Delta^2 + poly(n)))$ neurons, where Δ is the maximum degree of any node in the graph and the polynomial represents the number of neurons for the best constant-depth adder circuit. If saving neurons is more important, then the running time is $O(L \log k)$ and the number of neurons is $O(m \log k)$. For the rest of this paper, we assume that we are using circuits of the second, neuron-saving type.

4.2 Polynomial-time algorithm

Let U be the length of the longest edge in the input graph. In this algorithm, each synapse has unit weight and the same delay x (specified below). Each neuron has initial voltage 0, threshold voltage 1, and decay 0. Each message is a $\lceil \log(nU) \rceil$-spike message encoding a value d representing the length of a path from source v_s. The node corresponding to v_s sends initial spikes with value 0. When a node v receives a message with value d from a node u, it propagates spike messages with value $d + w(uv)$ to its neighbors, where $\ell(uv)$ is the length of uv. If the node v transmits a spike message of value d' at time tx, then there is a t-edge path of length d' from v_s to v. The NGA terminates after kx time steps or when the node corresponding to v_t receives a spike, whichever occurs first. (If we want to find distances from v_s to all other nodes, then the NGA terminates after kx time steps or when all nodes have received a spike, whichever occurs first.)

We can add an edge length to a value d with a depth-$O(\log(nU))$ circuit. We can compute the minimum over spike messages that arrive simultaneously using a circuit of depth $O(\log(nU))$ as described in Section 5. Using circuits of this depth requires delay $\Omega(\log(nU))$ on all synapses. We thus set $x = c \log(nU)$ for some positive constant c.

Performance. The running time of the algorithm is $O(kx + m) = O(k \log(nU) + m)$. Again we include the $O(m)$ term to account for the time required to load G into the SNA.

Algorithm for SSSP. If we want to compute single-source shortest paths without the k-hop restriction, then we just set k to α, where α is the number of edges on the shortest path from v_s to v_t. The running time of the algorithm is $O(\alpha \log(nU) + m)$.

4.3 Constructing paths

The algorithms of this section only compute the *length* of the optimal shortest single-source (k-hop) paths. Constructing the path requires the algorithms to store additional information at each graph node. See Section 3 for an example. For the k-hop algorithms, the extra storage requires a multiplicative factor of $O(k)$ additional neurons.

4.4 Embedding general graphs into the crossbar

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 2 for an example where $n = 3$.

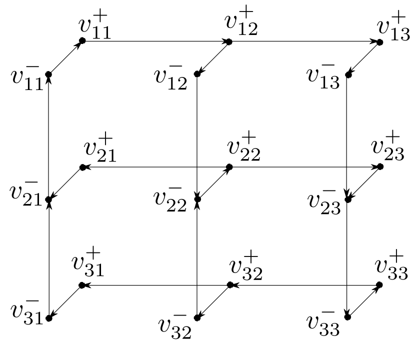

Figure 2: The stacked grid (or crossbar) H_3. The general H_n is a topology we may reasonably expect as a subset of every neuromorphic architecture.

This section shows how to *embed* any n-vertex input graph G into the stacked grid 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.

Let δ be the minimum delay we are allowed to assign to a single edge in the crossbar. This is a hardware-dependent constant, so we assume that $\delta = 1$ for the purposes of this discussion. We then scale all edge lengths in G so that the smallest length is $2n$. Now number the vertices in G from 1 to n, and let $\ell(ij)$ be the length of edge ij in G. The edges of H_n come in six types, from above. Set all edges of Types 1, 3, 4, 5, 6 to have unit delay, and set all edges of Type 2 to have delay $\ell(ij) - 2|i - j| - 1$.

Intuitively, the vertex i in G is represented in H_n by the induced graph on the vertices

$$v^-_{1i}, \ldots, v^-_{ni}, v^+_{i1}, \ldots, v^+_{in}$$

in H_n. The edge ij in G then roughly corresponds to the edge $v^+_{ij} v^-_{ij}$. Edges between vertices in H_n that correspond to the same vertex in G have unit delay.

We claim that finding a shortest path from vertex i to vertex j in G is now equivalent to finding a shortest path from v^-_{ii} to v^-_{jj} in H_n. To see this, let us verify that if ij is an edge in G, then the length of the path from v^-_{ii} to v^+_{jj} in H_n is still $\ell(ij)$; the claim then follows by induction. Indeed, the path from v^-_{ii} to v^-_{jj} goes through vertices

$$v^-_{ii}, v^+_{ii}, \ldots v^+_{ij}, v^-_{ij}, \ldots v^-_{jj}.$$

and thus has length

$$1 + |j - i| + [\ell(ij) - 2|i - j| - 1] + |j - i| = \ell(ij).$$

Recall that we scaled all edge lengths in G up until the smallest length was $2n$. This was necessary so that Type-2 edges in H_n have delay at least 1. If data movement is not assumed to take $O(1)$ time, this increases the running time of the neuromorphic Dijkstra's algorithm by a factor $O(n)$. Specifically, the time required to load the graph is still $O(m)$, but all other steps now require more time by a factor $O(n)$. For the rest of this paper, we refer to this blowup as the *embedding cost*.

Running time. We embed a graph G with n vertices and m edges into a graph H_n with $\Theta(n^2)$ vertices and $\Theta(n^2)$ edges, so naively the embedding takes $\Theta(n^2)$ time. However, in the embedding all edges of Types 1, 3, 4, 5, and 6 have delay δ, and a Type-2 edge $v_{ij}^{+}v_{ij}^{-}$ has finite delay only if ij is an edge in G. Thus if we assume that the crossbar initially has delay δ on all edges of Types 1, 3, 4, 5, 6 and infinite delay on all edges of Type 2, then when embedding G, we only need to change the delays of m edges of H_n. So embedding G takes $O(m)$ time. Furthermore, we can easily maintain this assumption even while embedding multiple graphs one after the other (i.e., not simultaneously) while only incurring a constant-factor slowdown, as follows. Suppose we wish to embed p graphs $G_1, \ldots, G_p$, in that order. For all i, let m_i be the number of edges of G_i. If G_{i-1} is currently embedded into H_n and we want to embed G_i instead, then we first "unembed" G_{i-1} by setting all m_{i-1} Type-2 edges with finite delay to have infinite delay, and then embed G_i by setting the appropriate m_i Type-2 edges to have finite delay. The resulting embedding of G_i is correct. It takes $O(m_i)$ time to both embed and unembed a graph G_i, so we only incur a constant-factor slowdown.

Remark. The above shows how to implement our pseudpolynomial time algorithms on a crossbar. For our polynomial time algorithms, extra care must be taken since each message is now λ bits. In addition we must embed the circuits used to perform arithmetic on the λ-bit messages. For the NGAs we describe in this paper, this can be done with logarithmic overhead.

4.5 Summary of running times

After taking into account movement cost and embedding cost, we get the following results for finding distances (not paths). These results are also summarized in Table 1. The difference between the running times in the two parts of the table is that we do not need to take into account embedding costs when ignoring data-movement costs, since we are assuming that data movement between vertices can take $O(1)$ time regardless of how the input graph G is embedded.

For SSSP, the pseudopolynomial-time neuromorphic algorithm runs in $O(L)$ time ignoring embedding cost and loading time. The loading time is $O(m)$. The embedding cost increases the running time of the spiking portion of the algorithm by a factor of $O(n)$, from $O(L)$ to $O(nL)$. We have thus proved the following theorem:

Theorem 4.1. *The pseudopolynomial-time neuromorphic SSSP algorithm runs in $O(L+m)$ time with $O(1)$-time data movement, and otherwise runs in $O(nL+m)$ time.*

For k-hop SSSP, the pseudopolynomial-time neuromorphic algorithm runs in $O(L\log k)$ time when not taking embedding cost or loading time into account. The embedding cost is an $O(n)$-factor increase in the running time of the spiking portion of the algorithm, from $O(L\log k)$ time to $O(nL\log k)$ time.

For each node v, with in-degree (number of incoming edges in G) indeg(v), we require a circuit of $O(\text{indeg}(v)\log k)$ neurons to compute the maximum of the indeg(v) TTLs that v receives (see Section 5). We require a circuit of $O(\log k)$ neurons to subtract 1 from this maximum TTL. Summing over all nodes v, the algorithm uses a total of $O(m\log k)$ neurons to compute TTLs. It thus takes $O(m\log k)$ time to load the circuits computing the TTLs into the SNA. This proves the following theorem:

Theorem 4.2. *The pseudopolynomial-time neuromorphic k-hop SSSP algorithm runs in $O((L+m)\log k)$ time with $O(1)$-time data movement, and otherwise runs in $O((nL+m)\log k)$ time.*

For k-hop SSSP, the polynomial-time neuromorphic algorithm runs in $O(k\log(nU))$ time when not taking embedding cost or loading time into account. The embedding cost is an $O(n)$-factor increase in the running time of the spiking portion of the algorithm, from $O(k\log(nU))$ to $O(kn\log(nU))$.

For each edge uv, we require a circuit of $O(\log(nU))$ neurons to add $\ell(uv)$ to the value of the spike message that u sends to v. For each node v, we require a circuit of $O(\text{indeg}(v)\log(nU))$ neurons to compute the minimum of the values of the indeg(v) spike messages that v is receiving. Summing over all nodes and edges, the algorithm uses $O(m\log(nU))$ neurons to correctly handle spike messages. It thus takes $O(m\log(nU))$ time to load the circuits handling the messages into the SNA. This proves the following theorem:

Theorem 4.3. *The polynomial-time neuromophic k-hop SSSP algorithm runs in $O(m\log(nU))$ time when data movement takes $O(1)$ time, and otherwise runs in $O((nk+m)\log(nU))$ time.*

For the polynomial-time neuromorphic SSSP algorithm, we use the same analysis as for k-hop SSSP but replace k with α, where α is the number of edges in the shortest path from source v_s to sink v_t.

Theorem 4.4. *The polynomial-time neuromophic SSSP algorithm runs in $O(m\log(nU))$ time when data movement takes $O(1)$ time, and otherwise runs in $O((n\alpha+m)\log(nU))$ time.*

5 CIRCUITS DESCRIPTIONS

Max Circuit Description. We describe two circuits to compute the maximum of d λ-bit non-negative binary numbers. For the pseudopolynomial circuit for k-hop shortest path, we have $\lambda = O(\log k)$. The first design is inspired by the wired-or (or global-or) method for computing the maximum of many values in the Connection Machine 2 [23, p.303]. The algorithm begins with all candidate numbers *active*. It then computes the OR of the most-significant bit (msb) from each number. If no number has a 1 in its msb (the OR is 0), all numbers remain active. Otherwise, any number with a 0 in its msb cannot be the maximum and it becomes *inactive*. This test repeats with the remaining active numbers for each bit in order. At the end all active numbers have the (same) maximum value. The second circuit is more brute force, comparing all pairs of numbers, and selecting the value that wins all comparisons. Both our max circuits offer trade-offs with the best known linear-threshold-gate circuits for computing max from [36]. Siu and Bruck [36] show how

Table 2: Overview of our neuromphic circuits to compute the maximum of d λ-bit numbers. The complexity bounds do not include embedding cost.

Name	Circuit Size (# of neurons)	Runtime (depth)
brute force	$O(d^2)$	3
Wired-or	$O(d\lambda)$	$O(\lambda)$

to turn any linear-threshold circuit with large weights into a linear-threshold circuit of small weight, constant depth, and polynomial size. Our bit-by-bit circuit sacrifices constant depth for reduced neuron counts. Our brute-force circuit uses larger synapse weights and fan-in. Table 2 summarizes the performance of these two circuits.

Notation. Let b_i denote the i^{th} input number and $b_{i,j}$ denote the j^{th} bit of b_i, where bit λ is the most significant. Unless otherwise noted, neurons have threshold 1, initial potential 0, and no decay, and synapses have weight 1 and delay δ (hardware minimum).

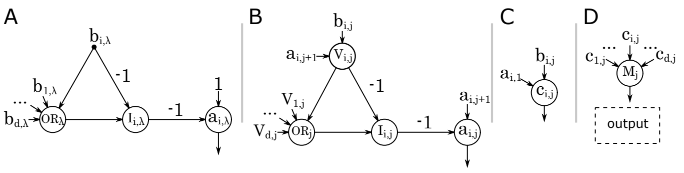

Figure 3: Circuits to compute the max of d λ-bit numbers bit by bit. (A) and (B) show circuits to deactivite non-max numbers processing from most-significant bit to least. (C) filters out numbers that do not have a maximum value (neuron c_{ij}'s threshold is 2). (D) combines all filtered inputs into a single maximum-value output. In (A) neurons $b_{i\lambda}$ for all i are input to OR$_\lambda$ (that is, $b_{i\lambda}$ is not included in the set entering OR$_\lambda$ from the left). Similarly each $V_{i,j}$ enters OR$_j$ exactly once. The threshold for $V_{i,j}$ is 2. Unspecified thresholds are 1, all initial potentials are 0, edge weights are 1, delay=δ (hardware minimum), and there is no decay. b_{ij} is the jth bit of the ith input number. Neuron a_{ij} fires iff number b_i is still active after processing through bit j.

Max Circuit details. The *time* or *depth* of a circuit is the maximum-length path from any input to any output.

Theorem 5.1. *There is a circuit to compute the maximum of d λ-bit numbers that, ignoring embedding costs, has $O(d\lambda)$ neurons (size) and $O(\lambda)$ depth.*

Proof. The bit-by-bit circuit has $\lambda + 2$ layers. The first λ layers eliminate inputs using the method described above. The last two layers select the inputs with maximum value and merge them into a final output respectively.

Figure 3 shows the circuits that gradually disqualify non-max numbers, where Figure 3A, processing the most significant bit ($i = \lambda$), is a special case of Figure 3B. Neuron a_{ij} fires if and only if (iff) number b_i is still active after processing bit j. If neuron $I_{i,j}$ fires then b_i is inactive from the processing of bit j onward; otherwise an active number stays active. Signals going into a_{ij} are delayed to ensure that $I_{i,j}$ are in sync. Neuron $V_{i,j}$ has a threshold of 2, acting like an AND gate. It fires iff b_i is still active at the start of processing bit j (i.e. $a_{i,j+1} = 1$) and bit $b_{ij} = 1$. In this case ($V_{ij} = 1$), number b_i always stays active to the next round. We say b_i is *guaranteed active* for bit j. If neuron V_{ij} fires, it sends a -1 weight into neuron I_{ij} preventing it from firing. Neuron OR$_j$ is an OR gate which fires iff any input is guaranteed active for bit j. If $V_{ij} = 0$, then disabling neuron I_{ij} fires iff OR$_j = 1$ (there is at least one number guaranteed active). Figure 3A is the same as Figure 3B, but hardwiring $a_{i,\lambda+1} = 1$ to indicate all numbers start active. In this case, there is no need for $V_{i\lambda}$, so the most-significant bits feed into the OR$_\lambda$ and $I_{i\lambda}$ neurons directly. Any number b_i where a_{i1} fires has the maximum value.

Figure 3C shows part of layer λ+1, which filters all input numbers. There is one such circuit for every bit j of every input number b_i. Neuron $c_{i,j}$ has a threshold of 2. Neuron a_{i1} fires if the ith input is still active (i.e is a maximum). If $a_{i1} = a_{k1} = 1$, then $b_i = b_k$ (tied for max). Input b_i is copied/passed to the next level iff it has maximum value. Figure 3D shows part of a circuit to merge the values from the $\lambda + 1$st layer onto the output. Each output-layer bit fires iff the max has a value of 1 for that bit.

This circuit runs in $O(\lambda)$ time and has $O(d\lambda)$ neurons, ignoring embedding costs. □

These max circuits run for each of the k hops in the shortest-paths algorithm. We must reset all the initial neuron values to 0. We assume a version of neurons that requires all inputs to arrive simultaneously, and resets to default value afterward whether it fires or not. With appropriate delays of the input bits (delay q to run on level q), any reasonable neuromorphic hardware should be able to implement this kind of neuron.

We can use the same circuit to compute the minimum of d λ-bit numbers. The only change is that we must negate each input bit using the NOT circuit in Figure 5A and use the negated bits in the circuits for the first λ layers to compute the a_{i1}.

Theorem 5.2. *There is a circuit to compute the maximum of d λ-bit numbers that, ignoring embedding costs, has $O(d^2)$ neurons (size) and depth* 3.

Proof. This circuit uses brute force, based on circuits in [36]. This circuit has 3 layers. The first computes all pair-wise comparisons of inputs b_x vs b_y for $x < y$. The left side of Figure 5A shows the circuit. The weights on the edges correspond to the numerical value of each bit's place. That is, bit r of a binary number represents the value 2^{r-1}. The sum in the threshold circuit computes $\sum_{i=1}^{\lambda}(2^{i-1}b_{xi} - 2^{i-1}b_{yi}) = b_x - b_y$. Neuron C_{xy} has threshold 1 (since the input values are integer) so it fires iff $b_x \geq b_y$. In the second layer, we compute C_{yx} for $x < y$. The circuit on the right side of Figure 5A is a NOT circuit that fires iff $C_{xy} = 0$, that is, iff $b_x < b_y$. The third and final layer determines which input contains the maximum value, breaking ties with the smallest index. The input b_x that wins all of its comparisons, as shown in Figure 5B, causes M_x to fire. If there are ties, the b_x with the smallest index x is the winner. This max-computing circuit has $O(d^2)$ neurons and depth 3. □

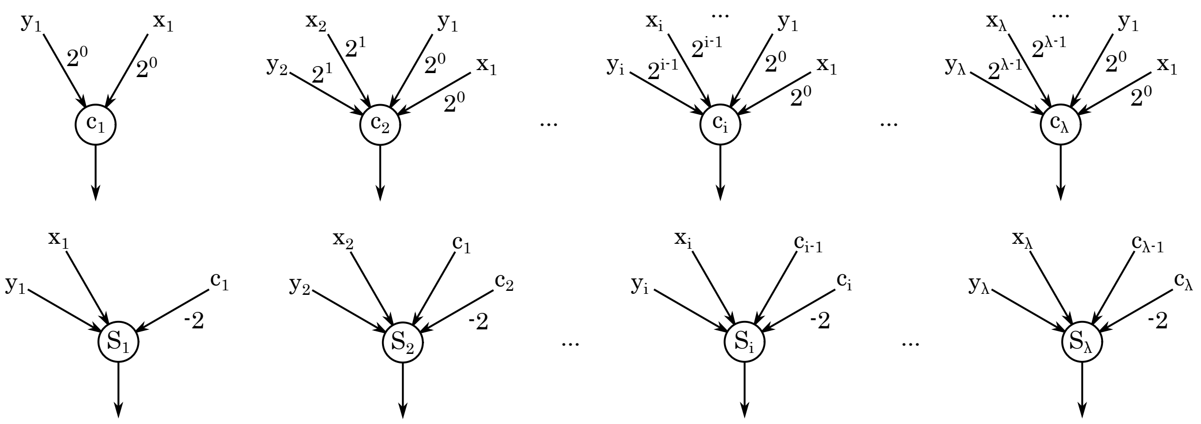
Figure 4: Adder circuit based on Ramos and Bhoórquez's circuit [33]

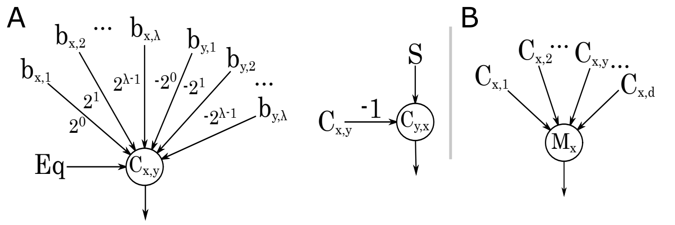
Figure 5: Brute-force comparison circuit. All thresholds are 1. A: Neuron C_{xy}, for $x < y$ (far left), fires iff $b_x \geq b_y$. Neuron C_{xy}, for $x > y$ (right), fires iff $b_x > b_y$. The input Eq is always 1 so C_{xy} fires on equality. The input S is always 1 so C_{yx} is effectively the NOT of C_{xy}. B: Neuron M_x has $d-1$ inputs: C_{xy} for all $y \neq x$. It has threshold $d-1$. It fires iff input b_x wins all its comparisons and therefore contains the maximum value.

We can compute the maximum value (rather than the index of the maximum value) using M_i the same way we used the a_{i1} neurons in the proof of Theorem 5.1. We can compute min instead of max by negating the weights of the incoming synapses of the circuits.

Sum Circuits. Two circuits in the literature add two λ-bit numbers. Siu et al. [37] calculates all the carry bits simultaneously. They give a depth-3, $O(\lambda^2)$-neuron adder using polynomially-bounded synaptic weights. Ramos and Bohórquez [33] use a carry lookahead. They give a depth-2, $O(\lambda)$-neuron adder using exponentially-bounded synaptic weights. See Figure 4 for the detailed circuits.

6 LOWER BOUNDS FOR CONVENTIONAL ALGORITHMS

In this section, we prove data-movement lower bounds for well-known conventional algorithms in the DISTANCE model, which was justified and defined in Section 2.3. In Section 4, we showed that proposed algorithms on neuromorphic architectures improve on the conventional results. Moreover, we tied this improvement to the advantages of neuromorphic architectures. Are we really observing a neuromorphic advantage, or can conventional algorithms do better via better algorithms or analysis? To answer this question, we study lower bounds for conventional methods. We start with a generic bound on the data-movement costs and then study the data-movement cost for the k-hop shortest path problem.

6.1 Lower bound on data movement costs

All distances in this section are ℓ_1 (Manhattan), because we assume data is stored in arrays of memory and is only accessible across rows or columns. We now define the *movement cost of an operation.* Suppose that prior to the operation, value v_1 is stored at position p_1 and value v_2 is stored at position p_2. Suppose further that the operation computes a value $f(v_1, v_2)$ for some function f at a register location p_r and stores it at a position p_3. Then the movement cost of the operation is the ℓ_1 distance from p_1 to p_r, plus the ℓ_1 distance from p_2 to p_r, plus the ℓ_1 distance from p_r to p_3. Positions p_1, p_2, p_r, and p_3 need not be distinct. The *movement cost of an algorithm* is the sum of the movement costs of its operations.

Theorem 6.1. *Suppose there are c registers and the input has size m. Then any algorithm that reads the entire input must incur $\Omega(m^{3/2}/\sqrt{c})$ movement cost in the DISTANCE model.*

Proof. Each of the m bits of input must be moved to at least one register over the course of the computation. A square of side length $\frac{\sqrt{m/c}}{2}$ has area $\frac{m}{4c}$. Thus at most $\left(\frac{m}{4c}\right)(c) < m/2$ of the input bits are within distance $\frac{\sqrt{m/c}}{4}$ of their nearest register. This means that at least $m/2$ of the input bits are at distance at least $\frac{\sqrt{m/c}}{4}$ from their nearest register. It follows that the total movement cost is at least $\left(\frac{m}{2}\right)\left(\frac{\sqrt{m/c}}{4}\right) = \Omega\left(\frac{m^{3/2}}{\sqrt{c}}\right)$. □

If c is a constant, then we get the desired $\Omega(m^{3/2})$ lower bound for Dijkstra's algorithm. A similar argument yields an $\Omega(m^{4/3})$ lower bound even when we only assume that the registers and disk reside in three dimensions and $c = O(1)$.

6.2 Lower bound for conventional k-hop shortest path algorithm

The best-known conventional algorithm for this problem is based on the Bellman-Ford algorithm and runs in $O(km)$ time (ignoring movement cost). Let $dist_k(v)$ be the length of the shortest path from v_s to v that has at most k edges. The algorithm computes all $dist_k(v)$ values, and it does so in $k+1$ rounds, where in the i-th round it computes $dist_i(v)$ for all v. In the zeroth round, the algorithm sets $dist_0(s)$ to be 0 and $dist_0(v) = \infty$ for all other vertices v. Each subsequent round consists of *relaxing* all edges, and relaxing a single edge $e = uv$ in the i-th round consists of performing the following assignment.

$$dist_i(v) \leftarrow \min\{dist_{i-1}(v), dist_{i-1}(u) + \ell(e)\}$$

The algorithm then returns $dist_k(v)$ for all v.

Theorem 6.2. *Suppose there are c registers. The above algorithm incurs $\Omega(km^{3/2}/\sqrt{c})$ movement cost.*

Proof. It suffices to show that each round except the zeroth round incurs $\Omega(m^{3/2}/\sqrt{c})$ movement cost. The proof is similar to that of Lemma 6.1. Since all edges need to be relaxed, each of the m edge lengths must be moved to at least one register over the course of the round. A square of side length $\frac{\sqrt{m/c}}{2}$ has area $\frac{m}{4c}$. Thus at most $\left(\frac{m}{4c}\right)(c) < m/2$ of the edge lengths are within distance $\frac{\sqrt{m/c}}{4}$ of their nearest register. This means that at least $m/2$ of the edge lengths are at distance at least $\frac{\sqrt{m/c}}{4}$ from their nearest register. It follows that the total movement cost of the round is at least $\left(\frac{m}{2}\right)\left(\frac{\sqrt{m/c}}{4}\right) = \Omega\left(\frac{m^{3/2}}{\sqrt{c}}\right)$. □

The desired lower bound on the running time of the algorithm follows if we assume $c = O(1)$.

7 AN APPROXIMATION ALGORITHM FOR k-HOP SSSP

In this section we describe a spiking algorithm that $(1+o(1))$-approximately solves the k-hop SSSP problem, based on a known algorithm in the CONGEST model. As we mentioned in Section 2.2, in general, algorithms in the CONGEST model cannot be simulated by SNNs without significant increases in space and time complexity, since CONGEST allows an individual node to perform any computation instantly. However, algorithms in the CONGEST model that do little computation at nodes can be simulated by SNNs fairly efficiently. One example of such an algorithm is a $(1+o(1))$-approximation algorithm for single-source k-hop shortest paths due to Nanongkai [30]. We will now describe a spiking version of Nanongkai's algorithm. Without loss of generality, let all edge lengths be at least 1. Let $dist(v)$ be the length of the shortest path from v_s to v, and let $dist_k(v)$ be the length of the shortest path from v_s to v with at most k edges. Nanongkai's algorithm relies on the following theorem, which he proved:

Theorem 7.1 ([30]). *Let $\epsilon = 1/\log n$. Let $D_i = 2^i$ and $\ell_i(uv) = \left\lceil \frac{2k\ell(uv)}{\epsilon D_i} \right\rceil$. Let $dist^{\ell_i}(v)$ be the distance from v_s to v when each edge e has length $\ell_i(e)$. For any node v, if*

$$\widetilde{dist}_k(v) = \min_i \left\{ \frac{\epsilon D_i}{2k} dist^{\ell_i}(v) : dist^{\ell_i}(v) \le \left(1 + \frac{2}{\epsilon}\right) k \right\},$$

then

$$dist_k(v) \le \widetilde{dist}_k(v) \le (1+\epsilon) dist_k(v).$$

Thus to compute a $(1+o(1))$-approximation of the distance from v_s to v_t, it suffices to compute $\widetilde{dist}_k(v_t)$. To do this, it suffices to compute $dist^{\ell_i}(v_t)$ for all i satisfying $dist^{\ell_i}(v_t) \le (1+2/\epsilon)k$.

Fix i. Our goal is to compute $dist^{\ell_i}(v_t)$ assuming that $dist^{\ell_i}(v_t) \le (1+2/\epsilon)k$. Note that ℓ_i is an integer-valued function. Thus we can run a modified version of our pseudopolynomial-time spiking SSSP algorithm (described in Section 3) to find $dist^{\ell_i}(v_t)$. Specifically, we run the algorithm on input graph G with length function ℓ_i, but terminate the algorithm early at time $\lceil (1+2/\epsilon)k \rceil$. This means that if $dist^{\ell_i}(v_t) \le (1+2/\epsilon)k$, then we have computed $dist^{\ell_i}(v_t)$. The modified spiking SSSP algorithm takes $O((1+2/\epsilon)k+m) = O(k\log n + m)$ time when data movement takes $O(1)$ time, and $O((1+2/\epsilon)kn + m) = O(kn\log n + m)$ time otherwise.

Note that if $i \ge \log(2kU/\epsilon)$, then $\ell_i(e) = 1$ for all edges e. Thus, to compute a $(1+o(1))$-approximation of the distance from v_s to v_t, it suffices to execute the modified spiking SSSP algorithm for all $i \in \{0, ..., \log(2kU/\epsilon)\}$, i.e., for $O(\log(kU\log n))$ different values of i. The total running time of the spiking approximation algorithm is thus $O((k\log n + m)\log(kU\log n))$ time when data movement takes $O(1)$ time, and $O((kn\log n + m)\log(kU\log n))$ time otherwise. We have proved the following theorem:

Theorem 7.2. *For an n-node, m-edge graph with longest edge length U, there is a neuromorphic $(1+o(1))$-approximation algorithm for k-hop SSSP that runs in $O((k\log n + m)\log(kU\log n))$ time when data movement takes $O(1)$ time, and in $O((kn\log n + m)\log(kU\log n))$ time otherwise.*

Comparing this to Table 1, we see that the running time of the spiking approximation algorithm is within polylogarithmic factors of the running time of the polynomial-time exact k-hop spiking SSSP algorithm. The main advantage of the approximation algorithm is in the neuron count: it uses n neurons for each value of i, or $O(n\log(kU\log n))$ neurons total, while the exact algorithm uses $O(m\log(nU))$ neurons (as calculated in Section 4.5).

8 CONCLUSIONS AND FUTURE WORK

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. Tidal flow [11] may be a promising starting point for a neuromorphic network-flow algorithm. Each iteration of tidal flow has a forward sweep from the source (breadth-first-search-like messages), a backward sweep from the sink and some local computation. Though it does not have the best-known complexity (currently $O(nm^2)$), it may be practical on neuromorphic systems. Efficient neuromorphic algorithm design may also draw inspiration from distributed computing, as illustrated in Section 7.

Our conventional data-movement lower bounds in Section 6 assume a constant number of registers. Processing-in-memory (PIM) models allow for the number of registers or computational units to grow with memory size; however, we do not know of any physically-viable and scalable implementations of PIM systems that are not subject to the demise of Moore's Law. One allure of neuromorphic computing is that scalability beyond Moore's Law is expected, with plausible physical examples such as the adult human brain (as outlined in Section 2.3).

Table 3: Selection of Current Scalable Neuromorphic Platforms

Platform	TrueNorth [28]	Loihi [10]	SpiNNaker 1 [31, 38]	SpiNNaker 2 [20]	Core i7-9700T [21]
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

APPENDIX

A Neuromorphic platform details

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 [6] 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 3, 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 6 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 [22, 29].

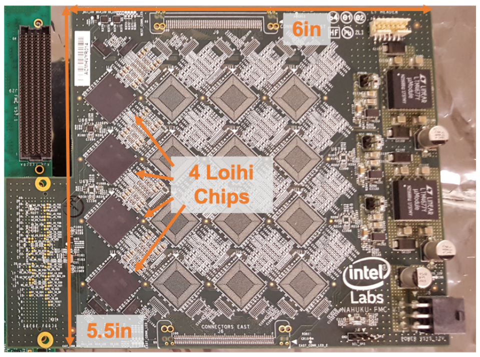

Figure 6: An Intel Loihi Nahuku form factor, which can support up to 32 chips (16 on the underside) attached to a host FPGA board (left). Each chip offers 128K neurons.

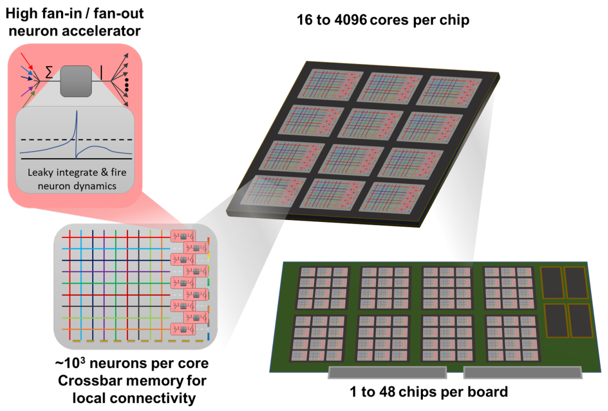
High fan-in / fan-out neuron accelerator
Leaky integrate & fire neuron dynamics
16 to 4096 cores per chip
~10³ neurons per core
Crossbar memory for local connectivity
1 to 48 chips per board

Figure 7: Current neuromorphic architectures aggregate many-core chips into boards.

ACKNOWLEDGMENTS

This work was funded 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 thank Michael Bender and David Tench from Stony Brook University for helpful suggestions and Craig Vineyard and the Neural Exploration and Research Lab at Sandia for access to Loihi hardware.

REFERENCES

[1] Agarwal, S., Quach, T.-T., Parekh, O., Hsia, A. H., DeBenedictis, E. P., James, C. D., Marinella, M. J., and Aimone, J. B. Energy scaling advantages of resistive memory crossbar based computation and its application to sparse coding. *Frontiers in neuroscience 9* (2015).

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

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

[4] Aimone, J. B., Parekh, O., Phillips, C. A., Pinar, A., Severa, W., and Xu, H. Dynamic programming with spiking neural computing. In *Proceedings of the International Conference on Neuromorphic Systems* (New York, NY, USA, 2019), ICONS '19, Association for Computing Machinery.

[5] Ali, A., and Kwisthout, J. A spiking neural algorithm for the network flow problem, 2019.

[6] Amir, A., Taba, B., Berg, D. J., Melano, T., McKinstry, J. L., Di Nolfo, C., Nayak, T. K., Andreopoulos, A., Garreau, G., Mendoza, M., et al. A low power, fully event-based gesture recognition system. In *CVPR* (2017), pp. 7388–7397.

[7] Azevedo, F. A. C., Carvalho, L. R. B., Grinberg, L. T., Farfel, J. M., Ferretti, R. E. L., Leite, R. E. P., Jacob Filho, W., Lent, R., and Herculano-Houzel, S. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *The Journal of Comparative Neurology 513*, 5 (Apr. 2009), 532–541.

[8] 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 (2014), 699–716.

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

[10] Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., Dimou, G., Joshi, P., Imam, N., Jain, S., et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro 38*, 1 (2018), 82–99.

[11] Fontaine, M. C. Tidal flow: a fast and teachable maximm flow algorithm. *Olympiads in Informatics 12* (2018), 25–41.

[12] Furst, M., Saxe, J. B., and Sipser, M. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory 17*, 1 (Dec. 1984), 13–27.

[13] Ghaffari, M., and Li, J. New distributed algorithms in almost mixing time via transformations from parallel algorithms, 2018.

[14] Hajnal, A., Maass, W., Pudlák, Szegedy, M., and Turán, G. Threshold circuits of bounded depth. *Journal of Computer and System Sciences 46* (1993), 129–154.

[15] Hamilton, K. E., Mintz, T. M., and Schuman, C. D. Spike-based primitives for graph algorithms, 2019.

[16] Hassabis, D., Kumaran, D., Summerfield, C., and Botvinick, M. Neuroscience-inspired artificial intelligence. *Neuron 95*, 2 (2017), 245–258.

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

[18] Hitron, Y., and Parter, M. Counting to Ten with Two Fingers: Compressed Counting with Spiking Neurons. In *27th Annual European Symposium on Algorithms (ESA 2019)* (Dagstuhl, Germany, 2019), M. A. Bender, O. Svensson, and G. Herman, Eds., vol. 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 57:1–57:17.

[19] Hitron, Y., Parter, M., and Perri, G. The Computational Cost of Asynchronous Neural Communication. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)* (Dagstuhl, Germany, 2020), T. Vidick, Ed., vol. 151 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 48:1–48:47.

[20] Höppner, S., Partzsch, J., Mayr, C., and Furber, S. 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.

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

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

[23] Kent, A., and Williams, J. G., Eds. *Encyclopedia of Computer Science and Technology.* Marcel Dekker, Inc., 1992. Volume 26, Supplement 11.

[24] Khan, M. M., Lester, D. R., Plana, L. A., Rast, A., Jin, X., Painkras, E., and Furber, S. B. 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* (2008), Ieee, pp. 2849–2856.

[25] Kwisthout, J., and Donselaar, N. On the computational power and complexity of Spiking Neural Networks. *arXiv:2001.08439 [cs]* (Jan. 2020). arXiv: 2001.08439.

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

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

[28] Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., Jackson, B. L., Imam, N., Guo, C., Nakamura, Y., et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science 345*, 6197 (2014), 668–673.

[29] Moore, S. 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-2020.

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

[31] Painkras, E., Plana, L. A., Garside, J., Temple, S., Galluppi, F., Patterson, C., Lester, D. R., Brown, A. D., and Furber, S. B. Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits 48*, 8 (2013), 1943–1953.

[32] 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, pp. 67–76.

[33] Ramos, J. F., and Bohórquez, A. G. Two operand binary adders with threshold logic. *IEEE Transactions on Computers 48*, 12 (1999), 1324–1337.

[34] 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, pp. 1947–1950.

[35] Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., and Plank, J. S. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963* (2017).

[36] Siu, K., and Bruck, J. On the power of threshold circuits with small weights. *SIAM J. Discrete Math. 4*, 3 (1991), 423–435.

[37] Siu, K.-Y., Roychowdhury, V. P., and Kailath, T. Depth-size tradeoffs for neural computation. *IEEE Transactions on Computers*, 12 (1991), 1402–1412.

[38] Stromatias, E., Galluppi, F., Patterson, C., and Furber, S. Power analysis of large-scale, real-time neural networks on spinnaker. In *The 2013 International Joint Conference on Neural Networks (IJCNN)* (2013), IEEE, pp. 1–8.

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

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

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

[42] Yao, A. C. C. Separating the polynomial-time hierarchy by oracles. In *, 26th Annual Symposium on Foundations of Computer Science, 1985* (Oct. 1985), pp. 1–10.