Shortest Path and Neighborhood Subgraph Extraction on a Spiking Memristive Neuromorphic Implementation

Catherine D. Schuman
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Kathleen Hamilton
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Tiffany Mintz
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Md Musabbir Adnan
University of Tennessee
Knoxville, Tennessee

Bon Woong Ku
Georgia Institute of Technology
Atlanta, Georgia

Sung-Kyu Lim
Georgia Institute of Technology
Atlanta, Georgia

Garrett S. Rose
University of Tennessee
Knoxville, Tennessee

ABSTRACT

Spiking neuromorphic computers (SNCs) are promising as a post Moore's law technology partly because of their potential for very low power computation. SNCs have primarily been demonstrated on machine learning and neural network applications, but they can also be used for applications beyond machine learning that can leverage SNC properties such as massively parallel computation and collocated processing and memory. Here, we demonstrate two graph problems (shortest path and neighborhood subgraph extraction) that can be solved using SNCs. We discuss the approach for mapping these applications to an SNC. We also estimate the performance of a memristive SNC for these applications on three real-world graphs.

CCS CONCEPTS

• **Theory of computation** → **Shortest paths**; • **Hardware** → **Neural systems**.

KEYWORDS

neuromorphic computing, graph algorithms, shortest path, neighborhood, memristors

Notice: This manuscript has been authored in part by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
NICE '19, March 26–28, 2019, Albany, NY, USA

https://doi.org/10.1145/3320288.3320290

ACM Reference Format:
Catherine D. Schuman, Kathleen Hamilton, Tiffany Mintz, Md Musabbir Adnan, Bon Woong Ku, Sung-Kyu Lim, and Garrett S. Rose. 2019. Shortest Path and Neighborhood Subgraph Extraction on a Spiking Memristive Neuromorphic Implementation. In *Neuro-inspired Computational Elements Workshop (NICE '19), March 26–28, 2019, Albany, NY, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3320288.3320290

1 INTRODUCTION

Spiking neuromorphic computing systems (SNCs) are compelling as a post Moore's law technology for a variety of reasons, including the potential for significantly lower power consumption than traditional computing architectures for certain workloads. The most natural set of applications to map to SNCs are neural network applications, because SNCs implement spiking neural networks in hardware. As a result, much of the work in neuromorphic computing applications thus far has been targeted towards neural network computation for both training and inference [17].

There are other application types beyond machine learning and neural network computation that can be implemented on SNCs that exploit the inherent architectural characteristics of SNCs in order to solve problems faster or more efficiently. An SNC is simply a specialized hardware system with certain characteristics. Communication is relatively simple and usually takes the form of spikes. SNCs have collocated processing and memory. Finally, SNCs are massively parallel- with many simple computational units (usually in the form of neurons and synapses) operating simultaneously. However, it is often non-trivial to map a non-neural network application onto an SNC.

In this work, we discuss how to map two graph problems, single-source shortest path finding and neighborhood subgraph extraction, onto a simulated memristive SNC system. We discuss how to configure the network topologies and parameters on the neuromorphic implementation, as well as how to apply input and interpret output from an SNC to solve these two problems. Finally, we estimate how the SNC will perform on three real-world graphs and discuss the energy estimates to perform those tasks.

2 NEUROMORPHIC APPLICATIONS BACKGROUND

The most natural set of applications to map to spiking neuromorphic systems are neural network applications. The types of neural network applications that have been shown either on physical neuromorphic hardware implementations or demonstrated using simulations of neuromorphic hardware include:

- Deployment of spiking convolutional neural networks for inference that were trained using conventional hardware (e.g., GPUs) [20, 21]
- Unsupervised and supervised on-chip training using plasticity mechanisms such as spike timing dependent plasticity (STDP) [11, 23]
- For both training and inference using reservoir computing or liquid state machines [10, 22]
- For training spiking neural networks using genetic algorithms or evolutionary computation [15]

For neural network applications, there is a significant amount of existing software and/or algorithms to be leveraged for building the appropriate "programs" (neural networks) to be deployed for real applications on neuromorphic systems. There is a much lower barrier of entry for non-experts to use neuromorphic systems as neural network accelerators than for non-neural network applications.

There have been relatively few use cases of neuromorphic systems for non-neural network applications. In [13], Monaco and Vindiola present a use case of neuromorphic systems as part of the sieving process for integer factorization. Severa et al. describe several neuromorphic algorithm approaches for particle image velocimetry in [19], and demonstrate that deciding how to map a problem to neuromorphic systems is non-trivial. In [18], Severa, et al., also demonstrate that neuromorphic systems can be used to generate Markov process random walks. In [2], Aimone, et al., proposes the spiking neural threshold gate model as a framework for realizing non-neural network applications for spiking neuromorphic systems. In [3], Araújo, et al., demonstrate that a spiking neuromorphic system can simulate the aggregate motion of a flock of birds.

One type of algorithm or application that is likely well suited to SNCs is graph algorithms. This is because SNCs natively implement spiking neural networks, which are essentially just specialized graphs. In [8], Hamilton, et al., describes a mapping of community detection in graphs to spiking neuromorphic hardware. They have also been used for other graph problems by mapping either constraint satisfaction problems to spiking neural networks [7] or mapping Ising models to spiking neural networks [5]. In [6], Davies, et al., note that there are methods (specifically [14]) for calculating shortest weighted path using SNCs; we discuss a variation of this approach further in Section 3.

One thing that is abundantly clear in all of these approaches is that mapping non-neural network applications to neuromorphic computing systems is likely to be non-trivial. Furthermore there will likely be trade-offs in different algorithmic approaches in the network size needed to implement the algorithm, the energy required (often mapped to the number of events in the system), and/or the wall clock run time of the system.

3 APPROACH

We configure spiking neural network "programs" such that the spiking dynamics can be used to extract relevant information from a graph without the need for extensive rate or temporal decoding. We construct an SNC similar to that described in [16]. This is a mixed analog-digital neuromorphic implementation where each synapse is implemented with two memristors (in order to allow for both positive and negative weight values). Our SNC is defined by a set of leaky integrate-and-fire neurons $\{n(v_{th}, t_R)\}$ with programmable thresholds (v_{th}) and refractory periods (t_R), and a set of synapses $\{s(\delta, s_w)\}$ with programmable delays (δ) and weights (s_w). We require that the synapses in the SNC can realize a form of spike timing dependent plasticity (STDP) and for the two graph problems discussed in this work, we rely on 1-step STDP. That is, only activity that occurs within a single time step around a neuron's firing can contribute to the plasticity process.

An undirected graph $G(V, E)$ is defined by a vertex set $V(G)$ and an edge set $E(G)$. A directed graph $D(V, E)$ is defined by a vertex set $V(D)$ and a set of directed arcs $E(D)$. The notation E can refer to either a set of undirected edges or directed arcs, but directed edges are labeled by the arc direction $e_{i\to j} \neq e_{j\to i}$ while undirected edges are labeled by the two terminal nodes $e_{ij} = e_{ji}$. We convert unweighted graphs into weighted graphs by assigning each $e \in E$ a length of 1. To avoid confusion with synaptic weights, we refer to graph edges as having lengths rather than weights.

The first step in configuring an SNC for graph applications is embedding the graph into a spiking neural network. We describe a relatively general embedding process in which a graph (G or D) is directly mapped to a SNC: each $v_i \in V$ defines a neuron $n_i \in n$ and each edge $e \in E$ defines a synapse $s \in S$. Directed arcs are mapped to directed synapses, and undirected edges are mapped to symmetric pairs of synapses $e_{ij} \to (s_{i\to j}, s_{j\to i})$. We assume that the graphs can be embedded directly into an SNC such that there are sufficient neurons and synapses and connectivity between them to realize the graph in the SNC. In future work, we plan to investigate adaptations of these algorithms in which the graphs cannot be directly embedded. Once the graph is embedded, the system parameters $(v_{th}, t_R, \delta, s_w)$ are defined by the specific application.

3.1 Shortest Path

We implemented a single-source multiple-destination shortest path finding algorithm on a simulated SNC using an approach similar to that described in [14]. The shortest path between a source vertex v_s and a destination vertex (or set of destination vertices) are non-backtracking and thus cannot return to any previously visited vertex. To implement this non-backtracking constraint in spiking dynamics we configure the SNC such that any neuron that fires only fires once. After directly mapping a given graph G or D into a spiking neural network we set the system parameters $(v_{th} = 0, t_R = \alpha, \delta = len(e) + 1, s_w = 1)$, which we derive below.

For each $v \in V$, we create a neuron $n(v_{th} = 0, t_R = \alpha)$ where α is defined as:

$$\alpha = \left(\sum_{e \in E} len(e) + 1 \right) + 1. \tag{1}$$

By setting the refractory period to be greater than the sum of the edge lengths in the graph, we force each neuron in the network to fire at most once. Our simulated SNC allows for very large refractory periods, so this can be easily realized. If a neuromorphic implementation cannot support arbitrarily large refractory periods, additional structure (more neurons and synapses) can be added for each vertex to force the corresponding neuron for that vertex to fire exactly once. This will inflate the size of the network required to implement the graph on the SNC, and it will also increase the number of accumulate and fire events in the implementation. However, neurons that only implement relatively short refractory periods may have much simpler circuitry and thus may reduce the energy required to perform each accumulate and fire event in the neuron.

For each $e \in E$ we create synapses $s(\delta = len(e) + 1, s_w = 1)$. In particular, note that the delay on a synapse is proportional to the length of the corresponding edge in the graph. However, all of the synapses in the initial spiking neural network will have the same weight value.

Once the spiking neural network is configured we load it onto the SNC device. Then, we choose source vertex v_s and stimulate its corresponding neuron n_s in the network at network time 0. We then simulate the activity in the network for α time steps, which is equivalent to running the SNC for the same amount of time. After simulation, we can determine the length of the shortest path from n_s to each neuron in the network (and thus the corresponding length of the path from the source node to every node in the graph) by looking at the fire times of each neuron. If a neuron does not fire in the allotted simulation time, then there is no path from the source node to that neuron. We can then determine the shortest paths in the network by reading out the updated network parameters. The synapses that have potentiated (increased weights) are those that caused their post-synaptic neuron to fire, and thus, are part of the shortest path to that neuron from n_s. If there are multiple shortest paths (i.e., the shortest paths are not unique), this approach will find all of the shortest paths. It is worth noting that simulating for α time steps is a worst-case scenario for single-source shortest path. Stopping the run after all neurons have fired can reduce the required simulation time and the resulting energy cost.

3.2 Neighborhood Subgraph Extraction

A neighborhood of vertex v_s in a graph G, $N_G(v_s)$, is defined as the subgraph made up of all vertices $v_j \in V(G)$ such that $e_{i,j} \in E(G)$ and the set of all edges that connect the vertices in this set. Examples of neighborhoods in a graph are shown in Figure 1. To extract $N_G(v_s)$ from spiking dynamics we need to configure the SNC such that only a subset of neurons fire throughout the "program." The extraction is done in two stages: first the vertices of $N_G(v_s)$ are identified, then the edges are identified. After directly mapping a graph into a spiking neural network, we load the network into the SNC and configure it with the following parameters: $(v_{th} = 0.5, t_R = 1)$ for all neurons, and $(\delta = 2, s_w = 1)$ for all synapses. In particular, the neuron refractory periods and the synapse delays are set such that if a neuron fires at time t, it will still be able to fire at time $t + \delta$ due to arriving charge from another neuron.

To identify the vertices, we stimulate neuron n_s at time step 0 and simulate the network for exactly two time steps (or long enough for charge to propagate from the source neuron along all of its outgoing synapses). The neurons that fire in those two time steps will be $\{n' \cup n_s\}$, the set of neurons that are adjacent to n_s and n_s itself, which correspond directly to the vertices in our desired subgraph. It is not required to "read" the updated network during this step, since all we are interested in obtaining in this step is which vertices are part of the neighborhood, which is determined entirely via output spikes. To identify the edges, we increase the spike threshold for all of the neurons outside the neighborhood subgraph, or $\{n\} \setminus \{n' \cup n_s\}$. The new threshold value for these neurons is set to be $|E(G)| + 1$. We then reset the simulation, clearing all activity, and load the new graph with updated thresholds and $s_w = 1$. We simultaneously stimulate every neuron in the set $\{n' \cup n_s\}$, which is the source neuron and all of its neighbors (as determined in the previous step). After two time steps we "read" the network parameters and determine all of the synapses that have $s_w > 1$ (the original weight). These are the synapses that have potentiated during the last run of the SNC, which means that they connect two neurons that are part of the neighborhood of the source neuron. Thus, these synapses correspond to the edges of $N_G(v_s)$.

Table 1: Energy Estimate per Event Type

	Accumulation	Fire	Learning	Idle
Neuron	9.81 pJ	12.5 pJ	-	7.2 pJ
Synapse	1.45 pJ	-	2.58 pJ	0.07 pJ

3.3 Estimating Energy Usage

To determine approximate energy usage of a memristive SNC on these tasks, we collect event information from the simulation of each network on the SNC. In particular, we track the number of times each neuron accumulates, the number of times each neuron fires, the number of times each synapse receives a fire, and the number of times each synapse potentiates or depresses for each network. We have previously discussed the details of this simulation code in [16], and we have discussed how we estimated energy usage for each event for this neuromorphic implementation in [1, 4, 16]. A summary of the approximate energy usage of those events is given in Table 1, where the neuron estimates come from [4] and the synapse estimates come from [1]. Because the behavior of the neuromorphic system is simulated using a discrete event simulation, we do not explicitly calculate the number of idle cycles for each neuron and synapse during the simulation. Instead, we use the number of simulation time steps (or simulated clock cycles), the number of neurons and synapses in the networks, and the other event counters to calculate how many idle cycles there are for neurons and synapses in the network.

4 RESULTS

To understand how a SNC may behave on real-world problems, we use three graphs from the Stanford Large Network Dataset Collection [12] to evaluate performance on a simulated SNC. These three graphs are: a California road network graph (roadNet-CA), a collaboration network graph from high energy physics on Arxiv (ca-HepPh), and a co-purchasing network from Amazon for June

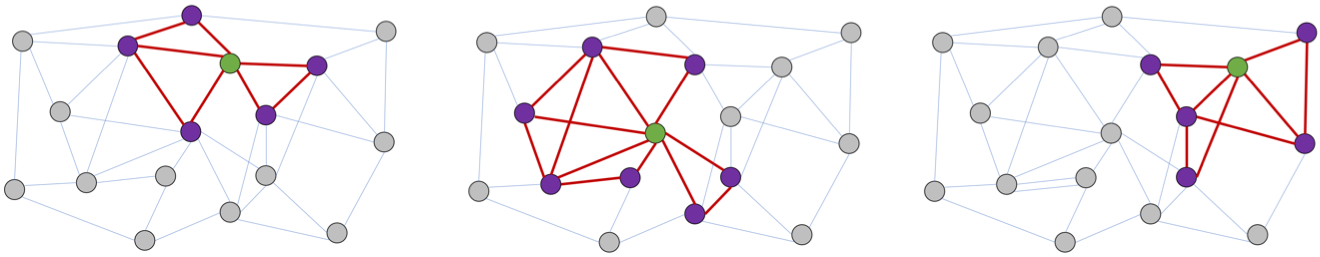

Figure 1: Examples of three neighborhoods in a graph. The source vertex v_s for each neighborhood is shown in green. The neighboring vertices are shown in purple, and the induced edges are highlighted and shown in red.

Table 2: Real-World Graphs [12]

Graph	Type	Vertices	Edges
roadNet-CA	Undirected	1,965,206	2,766,607
ca-HepPh	Undirected	12,008	118,521
amazon0601	Directed	403,394	3,387,388

1, 2003 (amazon0601). Table 2 gives information about each graph, including the number of vertices and edges, as well as whether the graph is directed or undirected. None of these graphs are weighed (i.e., the edges do not have associated lengths), so we implicitly include lengths of 1 for each edge.

To obtain performance for single-source shortest path and neighborhood subgraph extraction, we select the source vertex to be the vertex with the highest degree. To confirm the output of the neuromorphic implementation, we implemented naive shortest path and neighborhood subgraph extraction implementations for CPU. We confirmed that the output of the CPU implementations and the simulated SNC on these tasks are identical. In particular, for single source shortest path, the neuromorphic implementation output in both the lengths of the paths found and the paths themselves matched exactly. For neighborhood subgraph extraction, the neighborhoods (the vertices and edges) matched exactly for each approach.

Table 3 summarizes the differences between the two approaches in terms of SNC usage and traditional processor usage, as well as communication between the two. The number of clock cycles required on the SNC for shortest path relies on the size of the graph, while the number of clock cycles required for neighborhood subgraph extraction is fixed for any graph size. However, the shortest path method requires significantly less traditional processor computation for extraction. In particular, it requires a single $O(|E(G)|)$ sweep through the graph to determine the paths in post-processing. For neighborhood extraction, it requires $O(|V(G)| + |E(G)|)$ processing, where the $O(|V(G)|)$ sweep comes from resetting thresholds between neuromorphic calls and the $O(|E(G)|)$ sweep comes from determining which edges have been potentiated (which correspond to the edges that are in the subgraph).

Table 3 also summarizes the number of network loads and network reads required for each method as well. We contrast network reads and loads with simply applying spiking input to and reading spiking output from the SNC. Network "loads" require setting up a new network topology and/or parameters of the network, such as neuron thresholds or synapse weights. Network "reads" require reading the network parameters back off the SNC. In this case, the only parameters that are updated over the course of a simulation that might need to be read are the synaptic weights, which may be updated due to STDP. Though we note these communications in Table 3, we do not include those as part of the energy estimates. However, these communications are not likely to be optimized for most neuromorphic implementations, since for neural network implementations a network "load" will only occur once and a network "read" may never be required. Because they are not optimized, they may require a significant amount of time to complete on physical implementations.

Table 4 gives a breakdown of estimated energy usage per graph per task. Because of the O($|E|$) clock cycles required for shortest path (i.e., the number of clock cycles required depends on the number of edges in the graph), significantly more time is spent on the SNC than in the neighborhood subgraph extraction task, and thus, more energy usage is required. It is worth noting that these energy estimates do not capture the co-processor computation cost or communication cost associated with implementing these algorithms. However, they do give a rough estimate on what performance to expect from an SNC on this task.

As can be seen in Figure 2, the vast majority of the energy cost for both tasks comes from neuron and synapse idle states. In fact, it is only in the case of the ca-HepPh graph for the neighborhood subgraph extraction task that other event types contribute similarly to the energy estimate. For all other tasks, the idle costs are orders of magnitude more than the other event types. In the single-source shortest path task, because each neuron fires at most once, neurons and synapses are idle for most of the execution of the task. In the case of neighborhood subgraph extraction, neighborhoods in large graphs may be relatively small with respect to the larger graph, which again results in most neurons and synapses remaining idle for even the short execution time. If the idle costs in the SNC can be reduced, significantly less energy will be required to complete the task.

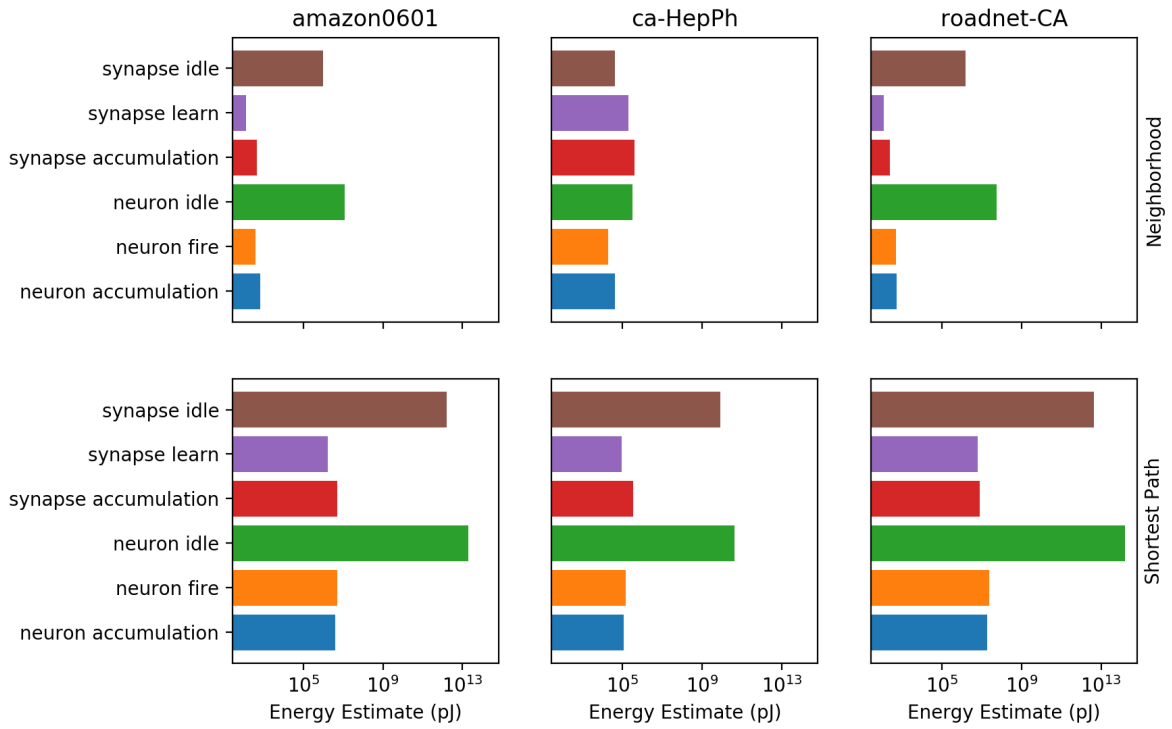

Figure 2: Energy breakdown by event type in simulation. The three graphs are shown in the three columns (left to right: amazon0601, ca-HepPh, and roadNet-CA). The first row shows results for neighborhood subgraph extraction task and the second row shows results for the single source shortest path calculation task. Note that the x-axis (the energy estimates) is on a log scale. The different event types are shown as bars of different colors.

Table 3: SNC/CPU Co-Processor Breakdown

	Shortest Path	Neighborhood
SNC Runtime	O($\|E(G)\|$)	O(1)
CPU Runtime	O($\|E(G)\|$)	O($\|V(G)\| + \|E(G)\|$)
CPU– >SNC Net Loads	1	2
SNC– >CPU Net Reads	1	1

Table 4: Estimated SNC Energy Usage Results

	roadNet-CA	ca-HepPh	amazon0601
Shortest Path	161.35 J	48.85 mJ	21.28 J
Neighborhood	58.32 μJ	999.14 nJ	12.56 μJ

5 CONCLUSION

Though mapping non-machine learning and non-neural network tasks to SNCs may be non-trivial, this is an area that has been relatively under-explored and has the potential for significant algorithmic development. In this work, we show how two graph problems can be mapped to a memristive SNC and give energy estimates on performance for three real-world graphs. These results illustrate how one might map a non-standard task onto an SNC. The results also show that the two tasks have different performance and ways of utilizing the characteristics of the hardware, despite both being graph-related tasks. It is worth noting that the goal of this work is not to give a direct comparison of how a neuromorphic system behaves with respect to a CPU. Instead, our goal is to illustrate that there are non-neural network applications that can be mapped to neuromorphic systems, how those mappings might occur, and how their performance characteristics may show properties of existing neuromorphic implementations that may not emerge in neural network applications.

An interesting consequence of this work is that both of these applications resulted in energy usages that were dominated by idle costs of the neuromorphic system. In our previous uses of neuromorphic systems, this was not a characteristic that was observed. In particular, either the networks used were run for fairly short amounts of time or there were much fewer neurons and synapses in the networks and most of the neurons and synapses were more active during simulation than the activity profiles for these applications. It is worth noting that non-neural network applications such as these will likely have very different performance characteristics than the neural network uses of neuromorphic systems. Thus, we should consider these types of applications in the design of neuromorphic systems moving forward, as there are likely to be more

and more non-neural network use cases of neuromorphic systems as they become more widely available.

In future work, we plan to explore direct comparisons with other compute systems (e.g., CPUs and GPUs) in terms of power and time-to-solution on non-neural network applications such as these two graph algorithms. We also intend to investigate implementing other graph algorithms; in this work we assumed the SNC had specific capabilities (e.g. synaptic plasticity, long refractory times). In particular, we are defining a variety of spike-based primitives that can be composed to produce more complicated graph algorithms and can be adapted to a wider variety of SNCs [9]. Finally, we intend to investigate the implementation of these algorithms on physical neuromorphic hardware systems, such as Intel's Loihi [6], so that we can quantify the effect of communication costs and embedding costs in an existing neuromorphic platform.

ACKNOWLEDGMENTS

This research is sponsored in part by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy. This material is based in part upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725.

REFERENCES

[1] Md Musabbir Adnan, Sagarvarma Sayyaparaju, Garrett S Rose, Catherine D. Schuman, Bon Woong Ku, and Sung Kyu Lim. 2018. A Twin Memristor Synapse for Spike Timing Dependent Learning in Neuromorphic Systems. In *Proceedings of the 31st IEEE International System-On-Chip Conference (SOCC)*. IEEE, To appear.

[2] James B Aimone, Ojas Parekh, and William Severa. 2017. Neural computing for scientific computing applications: more than just machine learning. In *Proceedings of the Neuromorphic Computing Symposium*. ACM, 7.

[3] Rui Araújo, Nicolai Waniek, and Jörg Conradt. 2014. Development of a Dynamically Extendable SpiNNaker Chip Computing Module. In *International Conference on Artificial Neural Networks*. Springer, 821–828.

[4] Gangotree Chakma, Md Musabbir Adnan, Austin R Wyer, Ryan Weiss, Catherine D Schuman, and Garrett S Rose. 2018. Memristive Mixed-Signal Neuromorphic Systems: Energy-Efficient Learning at the Circuit-Level. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8, 1 (2018), 125–136.

[5] Kevin Corder, John V Monaco, and Manuel M Vindiola. 2018. Solving Vertex Cover via Ising Model on a Neuromorphic Processor. In *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, 1–5.

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

[7] Gabriel A Fonseca Guerra and Steve B Furber. 2017. Using Stochastic Spiking Neural Networks on SpiNNaker to Solve Constraint Satisfaction Problems. *Frontiers in neuroscience* 11 (2017), 714.

[8] Kathleen E Hamilton, Neena Imam, and Travis S Humble. 2017. Community detection with spiking neural networks for neuromorphic hardware. In *Proceedings of the Neuromorphic Computing Symposium*. ACM, 9.

[9] Kathleen E. Hamilton, Tiffany M. Mintz, and Catherine D. Schuman. 2019. Spike-based primitives for graph algorithms. *arXiv e-prints*, Article arXiv:1903.10574 (Mar 2019).

[10] Dhireesha Kudithipudi, Qutaiba Saleh, Cory Merkel, James Thesing, and Bryant Wysocki. 2016. Design and analysis of a neuromemristive reservoir computing architecture for biosignal processing. *Frontiers in neuroscience* 9 (2016), 502.

[11] Chankyu Lee, Priyadarshini Panda, Gopalakrishnan Srinivasan, and Kaushik Roy. 2018. Training deep spiking convolutional neural networks with stdp-based unsupervised pre-training followed by supervised fine-tuning. *Frontiers in neuroscience* 12 (2018).

[12] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[13] John V Monaco and Manuel M Vindiola. 2017. Integer factorization with a neuromorphic sieve. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE, 1–4.

[14] Filip Jan Ponulak and John J Hopfield. 2013. Rapid, parallel path planning by propagating wavefronts of spiking neural activity. *Frontiers in computational neuroscience* 7 (2013), 98.

[15] Catherine D Schuman, James S Plank, Adam Disney, and John Reynolds. 2016. An evolutionary optimization framework for neural networks and neuromorphic architectures. In *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 145–154.

[16] Catherine D Schuman, Raphael Pooser, Tiffany Mintz, Md Musabbir Adnan, Garrett S Rose, Bon Woong Ku, and Sung Kyu Lim. 2017. Simulating and Estimating the Behavior of a Neuromorphic Co-Processor. In *Proceedings of the Second International Workshop on Post Moores Era Supercomputing*. ACM, 8–14.

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

[18] William Severa, Rich Lehoucq, Ojas Parekh, and James B Aimone. 2018. Spiking Neural Algorithms for Markov Process Random Walk. *arXiv preprint arXiv:1805.00509* (2018).

[19] William Severa, Ojas Parekh, Kristofor D Carlson, Conrad D James, and James B Aimone. 2016. Spiking network algorithms for scientific computing. In *Rebooting Computing (ICRC), IEEE International Conference on*. IEEE, 1–8.

[20] William Severa, Craig M Vineyard, Ryan Dellana, Stephen J Verzi, and James B Aimone. 2019. Training deep neural networks for binary communication with the Whetstone method. *Nature Machine Intelligence* 1, 2 (2019), 86.

[21] Sumit Bam Shrestha and Garrick Orchard. 2018. Slayer: Spike layer error reassignment in time. In *Advances in Neural Information Processing Systems*. 1412–1421.

[22] Nicholas Soures, Corey Merkel, Dhireesha Kudithipudi, Clare Thiem, and Nathan McDonald. 2017. Reservoir Computing in Embedded Systems: Three variants of the reservoir algorithm. *IEEE Consumer Electronics Magazine* 6, 3 (2017), 67–73.

[23] Johannes C Thiele, Olivier Bichler, and Antoine Dupret. 2018. A Timescale Invariant STDP-Based Spiking Deep Network for Unsupervised Online Feature Extraction from Event-Based Sensor Data. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.