SAND2022-2007C

Neural Mini-Apps as a Tool for Neuromorphic Computing Insight

Craig M. Vineyard
Sandia National Laboratories
Albuquerque, NM, United States
cmviney@sandia.gov

Suma Cardwell
Sandia National Laboratories
Albuquerque, NM, United States
sgcardw@sandia.gov

Frances Chance
Sandia National Laboratories
Albuquerque, NM, United States
fschanc@sandia.gov

Srideep Musuvathy
Sandia National Laboratories
Albuquerque, NM, United States
smusuva@sandia.gov

Fred Rothganger
Sandia National Laboratories
Albuquerque, NM, United States
frothga@sandia.gov

William M. Severa
Sandia National Laboratories
Albuquerque, NM, United States
wmsever@sandia.gov

J. Darby Smith
Sandia National Laboratories
Albuquerque, NM, United States
jsmit16@sandia.gov

Corinne Teeter
Sandia National Laboratories
Albuquerque, NM, United States
cmteete@sandia.gov

Felix Wang
Sandia National Laboratories
Albuquerque, NM, United States
felwang@sandia.gov

J. Brad Aimone
Sandia National Laboratories
Albuquerque, NM, United States
jbaimon@sandia.gov

ABSTRACT

Neuromorphic computing (NMC) is an exciting paradigm seeking to incorporate principles from biological brains to enable advanced computing capabilities. Not only does this encompass algorithms, such as neural networks, but also the consideration of how to structure the enabling computational architectures for executing such workloads. Assessing the merits of NMC is more nuanced than simply comparing singular, historical performance metrics from traditional approaches versus that of NMC. The novel computational architectures require new algorithms to make use of their differing computational approaches. And neural algorithms themselves are emerging across increasing application domains. Accordingly, we propose following the example high performance computing has employed using context capturing mini-apps and abstraction tools to explore the merits of computational architectures. Here we present Neural Mini-Apps in a neural circuit tool called Fugu as a means of NMC insight.

ACM Reference Format:
Craig M. Vineyard, Suma Cardwell, Frances Chance, Srideep Musuvathy, Fred Rothganger, William M. Severa, J. Darby Smith, Corinne Teeter, Felix Wang, and J. Brad Aimone. 2021. Neural Mini-Apps as a Tool for Neuromorphic Computing Insight. In *Proceedings of Neuro-Inspired Computational Elements Workshop (NICE2022).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Neuromorphic computing (NMC) is an actively evolving field with considerable uncertainty in how it may evolve over the next few years. While the primary focus on NMC has been on edge computing and real-time processing of artificial intelligence (AI) algorithms, there are increasing reasons to believe that neural approaches may be broadly useful for scientific computation [1]. NMC is somewhat unique among current computing technologies in that it is a novel computing paradigm. While the pursuit of algorithms and architectures inspired by the brain is not a new aspiration, the 'End of the Line' for conventional approaches has motivated the investigation of many Beyond Moore's Law technologies [10; 15]. Accordingly, NMC offers several exciting paths forwards as new enabling devices and technologies are developed, but also investigating how to change the computational paradigms to make use of this novel paradigm of computation. As such, the potential of NMC is not simply as an acceleration alternative, improving upon canonical approaches, but it also stimulates novel solutions. However, understanding the tradeoffs of these approaches is a complex challenge.

One complicating aspect of forecasting the long-term value and suitability of NMC in HPC domains is the uneven maturity across the different levels of the technology stack. That is, the compilers to NMC, the programming layer, and the neural algorithms themselves are simultaneously evolving at varying timescales. This presents a challenge to assessing the potential of a NMC solution to a problem: if a scoping study fails to deliver a promising result, is that due to a fundamental limitation of a hardware system?, of the architecture?, of the mapping of the algorithm to the hardware?, or is it a failure of the algorithm design itself? Alternatively, how do we attribute the benefits of an observed performance advantage across the different types of implementation?

To address this, we take inspiration from the example of high-performance computing (HPC) mini-apps which compactly encompass a computational workload [16]. This concept includes relevant context and details and enables cross platform compatibility investigations. Accordingly, here we introduce 'Neural Mini-Apps' as a mechanism for assessing NMC advantages. While benchmarking offers performance insight into specific algorithm-architecture

NICE2022, March 28 – April 1, 2022, San Antonio, TX, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

combinations, mini-apps offer a broader understanding. The context of the surrogate application can identify broader insights such as whether communication or I/O limit the acceleration of a key kernel. And in prescribing tune-able parameters pertinent to the application, neural mini-apps can offer understanding of the architectures themselves. We motivate this paradigm for neural algorithm and architecture understanding by presenting three diverse neural mini-apps using the Fugu neural circuit composition tool [4]. Fugu serves as an intermediary between neural algorithms, computational workloads such as mini-apps, and NMC hardware, and it facilitates performing studies to assist in the evaluation of emerging NMC technologies.

As follows, first we introduce the Fugu tool, then after a brief background on other NMC benchmarking considerations we expand upon how mini-apps offer insight into the interplay of algorithms and emerging computer architectures, we introduce three neural mini-apps which vary in numerical complexity and application, present results of how these neural mini-apps can characterize NMC architectures, and conclude with how this approach enables NMC computing insight.

2 BACKGROUND

2.1 Fugu

To help address the challenges in algorithm design and implementation, our Fugu tool aims to facilitate programming NMC hardware by combining two ideas: 1) component-based assembly of large solutions from small solutions, and 2) cross-platform portability [4]. Fugu attains cross-platform portability by establishing a minimalist neuron model which is generally supported by most neuromorphic devices. The minimalist approach is somewhat different from a least-common-denominator approach. The Fugu neuron model does support some features which are not always present. However, these are carefully selected based on their computational necessity, and there is often a way to substitute functionality on platforms that lack a given feature. Roughly speaking, Fugu is capable of describing any of the features used in threshold gates (fan-in, fan-out, threshold, synapse weights) and most features of spiking neurons (synapse delays, decay, noise). Learning is not currently specifically implemented within Fugu, but the framework for Fugu is compatible with having learning as an attribute of synapses.

Fugu attains component-based hierarchical assembly using the concept of a “brick”. A brick is essentially a section of network with well-defined inputs and outputs. Bricks with compatible pins can be connected together into larger bricks or into a final application. Fugu is primarily realized as a Python package. This includes a set of basic bricks, a brick assembly system called the “scaffold”, and backends. Bricks are implemented as Python classes which follow a defined interface in order to work within the scaffold. Their primary job is to generate sections of the network using Fugu neurons. Fugu is also realized with a graphical user interface using the N2A modeling system. N2A (“Neurons to Algorithms”) is an object-oriented language for creating large-scale neuroscience models. It includes equivalent representations of each Fugu brick, along with the ability to export Python code.

To support its minimalist neuron model, the Fugu software suite provides a “backend” for NMC hardware either translating a network into functioning code on a supported NMC platform or emulating the computation on a traditional computational platform. A Fugu backend is a Python class which handles both the task of compiling the network definition for a specific execution environment and the task of actually running the program and collecting results. Currently, backends exist for PyTorch (for conventional CPU), PyNN (compatible with SpiNNaker), and Intel Loihi.

Each backend takes as input the built scaffold and a dictionary of compiler options. Most of the work of a backend is to examine the generated neural network and translate it to the target. Fugu uses the NetworkX library to represent the network as a graph object. The nodes represent neurons and the edges represent synapses. Both are decorated with appropriate parameters.

For the purposes of algorithm prototyping as well as having a reference simulation, we provide Fugu with two CPU-based backends, SNN and DS. As reference simulators, they emphasize correctness and precision with less of a focus on optimal performance. Accordingly, these baselines offer single core, single threaded, commodity class baseline execution.

While Fugu speaks to the challenge of how to represent a computational abstraction for NMC, we next consider how to assess the performance of algorithms and architectures.

2.2 Machine Learning Benchmarks

To assess performance characteristics of neural networks and their execution, many measures have been considered. This ranges from application specific measures such as the accuracy of a neural network classifier to device or architectural properties like thermal design power and operation counts. Effectively, various approaches have been proposed by academic as well as industry efforts. Of the various approaches, a prominent recent effort MLCommons has garnered broad community support [21; 25]. This effort establishes various categories of assessment, defining a few neural network workloads and the required performance quality. For example, a training task must meet a minimum accuracy threshold for the particular model on the specified dataset. The benchmark also establishes closed and open divisions. The former requires the same model be used across the different systems, whereas the latter allows novel solutions suited for the computational architecture to most efficiently perform the task.

Despite the commonality of neural network computation, the precedence of MLCommons and related benchmark considerations (which are well posed for neural network accelerators, GPUs, etc.) does not directly translate to the alternative computing paradigm NMC pursues. In response, prior efforts have motivated the need for new considerations in benchmarking NMC [8; 23; 34–36]. This includes introducing new metrics and spiking datasets as well as explorations into how to characterize specific NMC architectures. Each of these efforts (as well as many others not highlighted here) have been valuable in advancing the understanding of the NMC field. However, a general limitation of the efforts to date has been a broadly applicable and well adopted NMC benchmarking paradigm. Accordingly, we are proposing to follow the lead of best practices established by traditional computing and next introduce the mini-app concept followed by our neural extension.

2.3 Mini-Apps

The concept of mini-applications was created in the context of understanding emerging computing architectures for high performance computing. Taken from Heroux 2009 [16], "*there is a middle ground for small, self-contained programs that, like benchmarks, contain the performance-intensive computations of a large-scale application, but are large enough to also contain the context of those computations.*" We expect that understanding the value of neuromorphic hardware for various applications will similarly benefit from the development of quantifiable surrogate applications that sit between benchmarks evaluating single kernels (like FLOPS or matrix operations) and full scale applications that are infeasible on emerging hardware prototypes.

Heroux et al. further lists as potential values of mini-apps as helping to enable:

- Interaction with external research communities
- Development of simulators
- Early node architecture studies
- Network scaling studies
- New language and programming models
- Compiler tuning

The 2009 work introduced mini-apps for finite element solving as well as contact analysis, molecular dynamics, parallel circuit simulation, and a collection of computational kernels. From the initial release, the concept has continued to grow with now over a dozen mini-apps available.[1]

Further enabling the mini-app construct to assess the merits of diverse computational architectures are capabilities such as Kokkos [11; 32] and RAJA [6]. These tools offer software abstractions enabling program portability for HPC codes. Figure 1 illustrates how mini-apps in conjunction with computational workload representation tools can engage with emerging computational platforms. The left half of the figure corresponds to the Mantevo Mini-Apps which through abstractions such as Kokkos/RAJA can then be assessed on emerging systems. Analogously, the Neural Mini-Apps construct presented here, in conjunction with Fugu, can bring this capability to NMC.

Combined, these benefits of mini-apps are well-suited to help assess novel technologies such as NMC. Accordingly, we see additional advantages such as:

- Influence future platforms
- Analyze testbed systems
- Tailor algorithms to emerging platforms

It is important to emphasize the relative maturity of emerging conventional technologies that may be considered for traditional computational platforms, such as FPGAs or ARM processors, is considerably higher than neuromorphic hardware. While emerging neuromorphic platforms are architecturally very sophisticated and advanced, they lack much of the software and algorithms infrastructure that benefit conventional systems. As a result, the last item in the list above—tailor algorithms to emerging platforms—is perhaps a greater part of why the Neural Mini-Apps effort that are described here than would be expected from those that target conventional platforms.

[1]https://mantevo.github.io/packages.html

3 NEURAL MINI-APPS

The Neural Mini-Apps we have targeted here are designed to provide some structure in assessing both hardware and software tools for NMC. The mini-apps themselves vary in numerical complexity and application, but their purpose is to fix the concrete numerical task so as to allow the different facets of the technology stack to be evaluated and improved.

Here, we introduce an initial set of three mini-apps that are intended to span the overall range over which NMC computing may contribute to computation. These initial mini-apps capture a diversity of computing applications that NMC researchers have identified as being 'wins' for the field, and there exist explicit claims (based on theory or lightweight scaling studies) for each of these applications that they can outperform conventional computing. The first mini-app, focused on random walk methods for solving partial differential equations, represents an NMC approach to a classic scientific computing challenge that is reformulated to use neuron-based computation. The exploration of NMC approaches to this computation includes [7; 17; 29; 30]. The second mini-app, focused on the sparse coding algorithm for image processing, represents an NMC approach to solving machine learning tasks for which neurons are typically used as logic elements [9; 13; 26; 28]. Finally, our third mini-app, focused on shortest path search on graphs, represents a class of graph analytics algorithms that are natively suited for NMC architectures. Explorations of various graph algorithms on NMC architectures includes [9; 14; 18; 27].

As shown in Fig. 2, the general workflow of a Neural Mini-App starts with the user identifying which backend should be run, and parameterizing the simulation according to domain specific parameters.

3.1 Neuromorphic Random Walk

Random walks are a powerful tool for utilization in large scale modeling and simulation as well as in randomized algorithms. Advantages of using random walks include that they may often be computed independently in parallel, and are computationally simple. However, their use in Monte Carlo methods may be less preferred in some instances due to their slow convergence. NMC hardware combats these reservations by providing a platform for cost-effective and efficient simulation of random walks. More efficient simulation means gaining more random walks or samples for the same cost. While this does not change the convergence rate of Monte Carlo methods, it does imply that better results can be achieved for the same cost as conventional random walk computation. Our neural random walk Mini-App may describe simple diffusion or be used to describe a discrete time Markov chain (DTMC). These Markov chains can be used in conjunction with Feynman-Kac style formulas to solve particular partial differential equations with non-local terms.

For our Random Walk Mini-App, we specifically consider a Particle Angular Fluence computation as the exemplar task. Particle angular fluence is the time-integrated flux of particles traveling

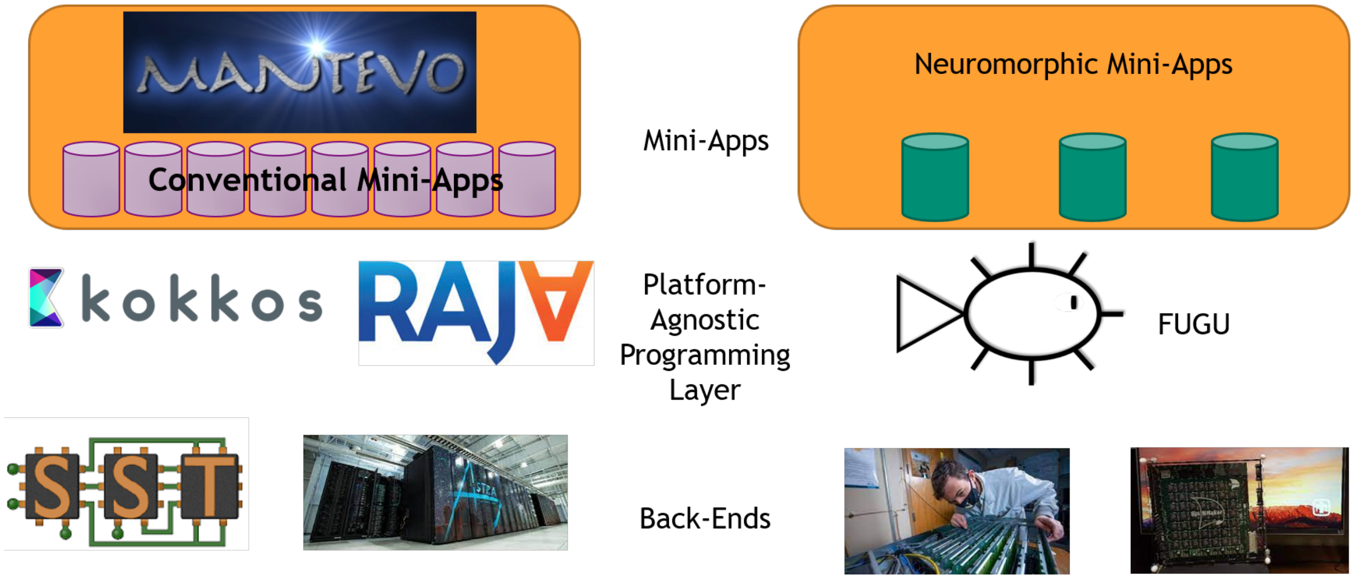


Figure 1: Overview of how mini-apps in conjunction with tools like Kokkos have enabled investigations into conventional computational platforms (left) and how the proposed development of Neural Mini-Apps in Fugu can bring similar insight into emerging neuromorphic computing platforms (right).

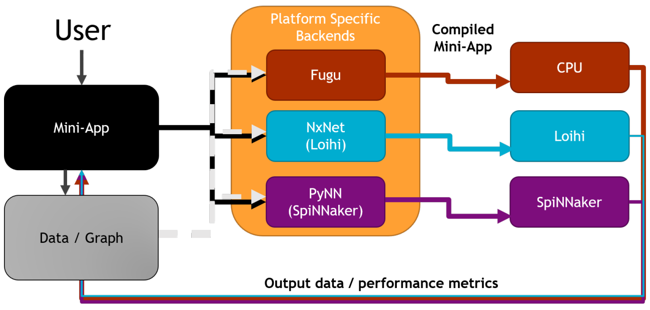


Figure 2: Neural Mini-App Conceptual Flow.

through media given as a function of position and velocity. Our particular task considers the case where particles travel at a constant speed and experience relative velocity scattering over a small region of space, ultimately exiting the area of interest at some angle. For this Mini-App, we can leverage the algorithm described in [29] and [30].

Mini-App Paramaterization

This Mini-App has several parameters that can be explored that relate to both potential physics considerations (particle absorption, velocities, etc), but our use-case of the Mini-App prioritizes an initial exploration on controlling the scale of the simulation as follows:

- Number of total walkers to use.
- Size of direction/relative velocity/angular discretization: this parameter controls the number of relative velocities available to particles in the DTMC simulation.
- Time discretization size: the time step size of simulation (To be probabilistically sound, this choice should be made so that it is reasonable that no more than a single scattering event could occur within the time step).
- Domain of interest: controls the size of the state space.
- Size of positional discretization: this parameter, ds, must be chosen so that minimal error arises from the DTMC approximation.

Note, the scale of the simulation can also be controlled through parallelization. If multiple copies of the mesh could be made, then parallel runs of walkers could occur for each of the state space locations within the sensor. Furthermore, additional parameters exist, but do not greatly affect the scaling of the Mini-Apps which is an emphasis here to explore the impact of enabling NMC architectures.

Mini-App Scaling

The primary dimension of scaling of relevance to DTMC simulations is the number of random walkers.

- General problem size scaling can be tested by adding more and more walkers M.
- A scaling related to conventional weak scaling can also be tested by increasing the mesh size with growing number of walkers M.

Mini-App Metrics

The radiation transport problem discussed for this mini-app does not typically have an analytically tractable solution. Therefore, comparison in output can be compared in aggregate to a large-scale simulation of the equivalent problem solved via the canonical method.

The performance of neuromorphic at the task is a far more interesting measure of success. Some ways we can measure success are:

- Cost of walkers: compare the energy efficiency of doubling the number of walkers on neuromorphic compared to doubling the number of walkers in the canonical method on conventional.
- Time to run: compare the additional time it takes to run double the number of walkers on neuromorphic versus the amount of time it takes to double the walkers in the canonical method on conventional.
- Space to run: compare the number of neuromorphic resources required to implement larger increased DTMC simulation meshes.

3.2 Neural Sparse Coding

Sparse coding or sparse dictionary learning is a way of modeling data by decomposing it into sparse linear combinations of elements of a given overcomplete basis set [22]. That is, a data vector $y \in \mathbb{R}^m$ may be approximated as multiplying a dictionary matrix $D \in \mathbb{R}^{m \times k}$ with a sparse vector $x \in \mathbb{R}^k$: $y \approx Dx$. When a dictionary or basis set has already been learned, the process of finding the sparse vector x for a given data vector y is called sparse coding. This is traditionally found through regression analysis by formulating the problem as the L_1-regularized optimization,

$$l(y, D) = \min_{x \in \mathbb{R}^k} \frac{1}{2} ||y - Dx||_2^2 + \lambda ||x||_1$$

where the cost may be understood as the contributions of the reconstruction error $\frac{1}{2}||y - Dx||_2^2$ and a sparsity penalty $\lambda||x||_1$, where λ is a regularization parameter determining the degree of sparsity imposed. This particular formulation is known as the LASSO (least absolute shrinkage and selection operator) [31].

On neuromorphic, the LASSO computation for sparse coding can be approximated with the spike-based locally competitive algorithms (LCA). These are a class of algorithms where a shallow network (i.e. single layer) of spiking neurons are laterally connected such that their dynamics are mutually inhibitory [26]. This gives the resulting competitive behavior where higher activation in a given neuron will suppress the activation of its connected neighbors. In approximating LASSO, the spiking neurons correspond to dictionary elements, and the amount of inhibition depends on the overlap between dictionary elements, as computed by their dot product. For the input, the overlap between the data vector and the dictionary element applies a current to the neuron, resulting in a "baseline" spike rate in the absence of inhibition. As the network activity approaches equilibrium between these two factors, the spike rates of the neurons can be read out as the sparse coding vector of the input with respect to the corresponding dictionary.

Our Sparse Coding Mini-App focuses upon the LASSO problem as it is solved through LCA in the data domain of natural images. Due to scaling limitations on the size of dictionaries that may be learned with respect to the size of the inputs, an image to be reconstructed through LASSO will be split up into smaller, multiple overlapping image patches, where each patch shares the same dictionary. For the full reconstruction, each image patch is first reconstructed individually and then weighted with their overlapping neighbors.

Mini-App Paramaterization

There are several ways to parameterize a given LASSO reconstruction for images:

- Size of the image: height and width in pixels of the image to be reconstructed.
- Size of the image patch: height and width in pixels of the image patch. This also provides the size of the input layer.
- Size of the dictionary: how many dictionary elements to be used to reconstruct a given image patch. This also provides the size of the output layer.
- Stride of the image patch (or overlap): this defines the tiling of the image patch over the image and along with the size of the image patch, provides the number of image patches in the image. For parallelization, this also provides the number of copies of the network needed.
- Desired sparsity: this is the sparsity (e.g. percentage of active neurons) of the resulting coding vector, and influences the inhibitory weights, spiking thresholds, and overall activity of the network.

Mini-App Scaling

We can scale the LASSO reconstruction problem for images along multiple dimensions:

- Primarily, we can increase the problem size by simply increasing the number of image patches to be reconstructed. Due to the independently computed image patches, this is an embarrassingly parallel form of weak scaling.
- Alternatively, we may increase parameters such as the input or dictionary sizes.

Mini-App Metrics

Solving the reconstruction through LCA on NMC can be measured in a number of ways (and with respect to both methods of scaling):

- Time for setup: while we can assume that a dictionary is already known beforehand, there will also be a number of "one-time" setup steps to initialize/load this to neuromorphic platform.
- Time to reconstruction: this can be an overall application metric (similar to inferences/second) that simply measures the total time needed to solve the LASSO reconstruction problem.
- Reconstruction performance: similar to accuracy in most classification problems, the reconstruction error is a measure of how well the LCA algorithm performed for solving the reconstruction problem.
- Reconstruction sparsity: this is the other term in the loss function, where we also care about the sparsity of the resulting sparse coding vector with respect to our desired sparsity.
- Compute resource usage: this should be a measure of how much we saturate the computing resources of the neuromorphic platform.
- Energy resource usage: similar to the above, this can measure how much energy the neuromorphic platform uses for its compute (potentially related to the number of spikes).

Because the reconstruction occurs over time, there are some other interesting ways we can break down the problem:

- Reconstruction performance curve: instead of simply waiting for the network to reach equilibrium, we can compute the reconstruction error along the way as if the intermediate spiking rates were the sparse coding vector. Knowing the rate at which the reconstruction problem is solved allows potentially early stopping of the algorithm by tuning for the reconstruction error to fall below some acceptable threshold.
- Sparsity performance curve: similar to the above, but for the sparsity (i.e. percentage of active output neurons) of the sparse coding vector.

3.3 Neural Graph Analysis

Graph analytics represents a wide range of algorithms and approaches used to analyze, process, and predict features of graphs. Recent research momentum suggests that graph algorithms may be well-suited for highly parallel neuromorphic platforms [2; 3; 14]. We concern ourselves with a standard graph algorithm–Single source shortest path (SSSP). That is, between a source and target node, what is the shortest path (and path length) that connects the two. More formally, if you have a vertex set V and a weighted edge set $E \subset \{(v, u, w) : (v, u) \in V^2, w \in \mathbb{Z}^+\}$, then a path from a source $s \in V$ to a target $t \in V$ is a finite list of edges $(v_0, u_0, w_0), \ldots (v_k, u_k, w_k)$ satisfying $u_0 = s$, $v_k = t$, $u_i = v_{i+1}$, for $0 \leq i < k$. The length of a path is defined as $\sum_{i=0}^{k} w_i$, and the shortest path is one with a minimum length. In general, the shortest path may not be unique.

A spiking neural network algorithm to solve the length of the SSSP is straightforward and well-known. Each vertex in the source graph is instantiated as neuron, and each edge is a synapse between the corresponding neurons. The weights of the source graph are converted to delays on the neural network. Each neuron is defined so that it spikes with any input. At run time, the source neuron is driven to fire by external input and the shortest path length is determined by the first spike time of the target neuron. This produces the shortest path length in an obvious way—the spikes that arrive first are those that took less time and therefore underwent the least aggregate delay.

For our Graph Analysis Mini-App, we used two extensions from this base algorithm. First, each neuron has a self-inhibition loop. This prevented neurons from firing more than once and increases the sparsity of spike activity. Second, we also have additional neurons essentially monitoring each of the edges. This allows a reconstruction of the path itself and not just the path length.

Mini-App Paramaterization

- Graph generator: the graph can be generated either as a uniformly random tree or using a small world graph generator.
- N: in the case of a tree, N represents the number of nodes. In the case of a small world graph N is the side length of a square lattice.
- Weight range: weights are assigned randomly to the edges of the graph. The weights are sampled uniformly from the weight range.
- Max runtime: amount of time to run the simulator or neuromorphic hardware.
- Source: source node on the graph for the shortest path.
- Target: target node on the graph for the shortest path.

Mini-App Scaling

The SSSP Mini-App can be scaled in multiple ways:

- Particularly relevant is the scale of the underlying graph. For example, we can scale N as noted above, altering the size of the graph in terms of nodes or connectivity depending upon the graph type.
- Relatedly, we can also adjust the weight range as it has a direct impact on the range of delay values of the network.

Mini-App Metrics

- Total time: measuring the total time includes the setup time and the execution time of the algorithm.
- Time for setup: mapping graph nodes to neurons with the appropriate connectivity and temporal delays.

4 RESULTS

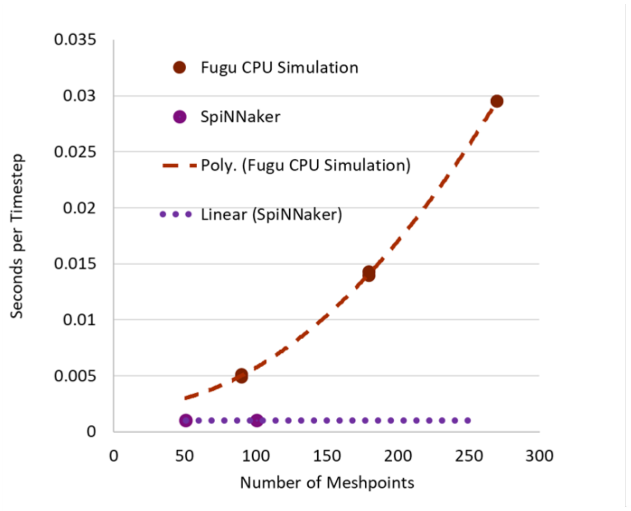
The core contribution of this paper is the presentation of Neural Mini-Apps as a tool for neuromorphic computing insight, rather than any particular application and architectural analysis itself. Nevertheless, to demonstrate the utility of this NMC assessment capability as follows we show example insights for how these initial Neural Mini-Apps are able to articulate differences in NMC computation.

4.1 Neuromorphic Random Walk

We have previously identified that NMC platforms, such as Loihi and TrueNorth, can be shown to be advantageous on DTMC random walks in the abstract, especially when required energy and time are both considered [29]. The specific nature by which this NMC approach parallelizes random walks is complex and beyond the scope of this paper, but a simple description is that the NMC model distributes the random walk state space over the available physical hardware (each NMC chip is responsible for a number of mesh points), while random walkers themselves travel across the hardware. Because the movement of a walker from one state to another can be represented as a discrete event, its walkers can very cheaply traverse over the available NMC state space. Thus, adding more walkers into this paradigm (classically the biggest cost in Monte Carlo simulations) is essentially free, but we have to pay directly for the state space that we simulate over.

For this advantage to be impactful for real-world applications, it is important to understand how the NMC implementation will scale with larger model sizes (which the algorithm to span across multiple hardware chips). While current test platforms are still relatively limited in size, we can begin to use this Mini-App to understand how increasing the state space impacts the overall computation time of Monte Carlo updates (the walkers themselves can move in a nearly embarrassingly parallel manner, but they still must be communicated between neurons across chips).

To illustrate one such preliminary result, we show in Fig. 3 that Loihi and SpiNNaker show promising scaling with larger model sizes. While our 48-chip SpiNNaker implementation was relatively limited in the overall model size that could be implemented, there did not appear to be any significant penalty for using an increased number of chips. Largely, this is due to the slow underlying system clock speed of SpiNNaker ($\approx$ 1 kHz) that ensures that all within chip computation and all chip-to-chip communication can be accomplished. Due partly to its more modern CMOS technology, Loihi has a faster underlying clock speed and can fit considerably more neurons on each chip, allowing much larger model sizes to be implemented. Because of Loihi's barrier synchronization, each system clock cycle updates when all other cores are ready, providing a slight penalty for increasing the number of chips being used, but this paradigm also permits the algorithm to update very rapidly, especially if there are no probes being used for continual I/O.



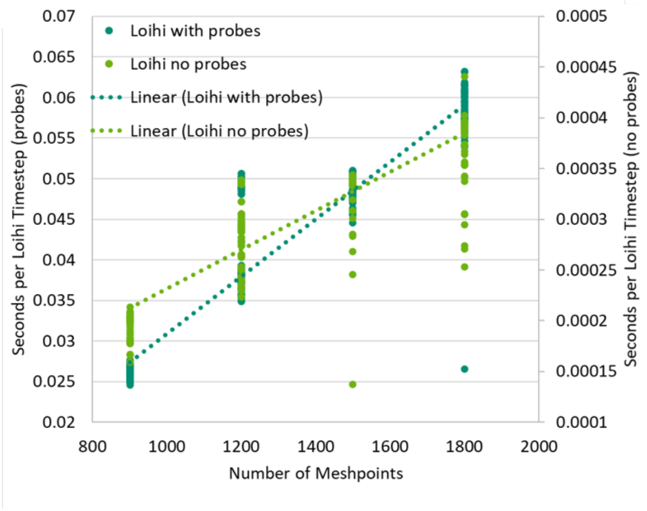


Figure 3: Scaling of random walk Mini-App on different neuromorphic platforms. Top: Timesteps are constant time despite increase state space size on SpiNNaker platform, whereas CPU simulation of neural algorithm shows polynomial scaling. Bottom: Scaling state space over an increased number of Loihi chips yields linear time scaling of random walk updates

4.2 Neural Sparse Coding

To illustrate architectural tradeoffs highlighted by weak scaling of the Sparse Coding Mini-App, we increased the problem size with an equivalent increase of compute resources. This was done by increasing the number of image patches to be reconstructed while keeping other algorithmic parameters (e.g. dictionary size) and compute parameters (e.g. resource saturation) fixed. Because image patches were reconstructed individually, the workload for LCA is considered embarrassingly parallel, which means that ideal scaling should be constant.

In addition to the neuromorphic implementations, we also considered conventional CPU-based solvers from the SPAMS optimization toolbox [20], namely the spams.lasso and spams.fistaFlat solvers: spams.lasso is a fast implementation of the LARS (least-angle regression) algorithm for LASSO, and is well adapted to small and medium-scale sparse decomposition problems; spams.fistaFlat is part of the proximal toolbox that implements FISTA (fast iterative shrinkage-thresholding algorithm), and is in a class of more general solvers adapted for a wide range of possibly large-scale learning problems. These approaches offer a reference of comparison with leading conventional algorithms [5; 12].

Due to the differences in resources available for the different platforms, the number of image patches per-platform were adjusted accordingly. The dictionary size was kept at 112 elements across all platforms. The default resource utilization parameters were used on Loihi and SpiNNaker, and for the CPU algorithms, the number of available threads was increased.

Individual timing results for each platform are shown in Figure 4. Error bars correspond to one standard deviation. On Loihi (without probes) and SpiNNaker, timings demonstrated ideal near-constant scaling of computation time with respect to the number of patches, up until resource limitations. However, a linear scaling trend was seen on Loihi when activity probes were turned on for Loihi, indicative of an I/O bottleneck. The CPU-based methods also showed poorer scaling, with FISTA showing near linear scaling (which meant quadratic scaling in compute requirements). However, this may likely have been due to the way threads are utilized by these methods as opposed to being weak scaling in the embarrassingly parallel sense.

4.3 Neural Graph Analysis

The initial graph type examined in the Mini-App are random trees, such as those shown in Figure 5. These are very sparse un-directed graphs for which we assign random weights to the tree's edges. Additionally, small world networks were explored as a different class of graph structure. These graphs are characterized such that most nodes are not direct neighbors, but due to the connectivity structure of the network most nodes can be reached from other nodes via a small number of edge traversals. Figure 6 illustrates examples with increasing side length of square small world lattice.

Using the SSSP Graph Analysis Mini-App to explore how scaling impacts computational architectures, we ran the same experiment on on both Loihi and CPU simulation (DS). Measuring the time for 1000 timesteps while searching for the shortest path between 2 nodes we then scaled by increasing the size of NetworkX random trees. As shown in the top of Fig. 7, Loihi is much faster than the

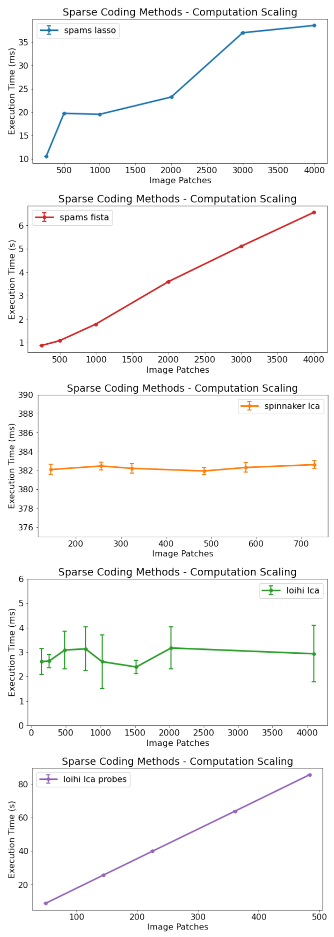


Figure 4: Weak scaling sparse coding experiments for Loihi, SpiNNaker, and CPU

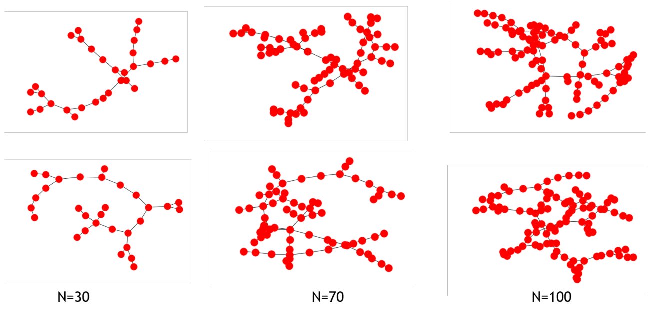


Figure 5: Example graphs of NetworkX random trees with increasing node counts (N).

Fugu CPU simulation for the shortest path task. Further, it is notable that the Loihi execution time scaling (which ignores setup time) is effectively independent of the size of the graph. This is largely due to the fact that the Loihi embedding limits the number of neurons per core, so the algorithm is mostly operating in an extremely

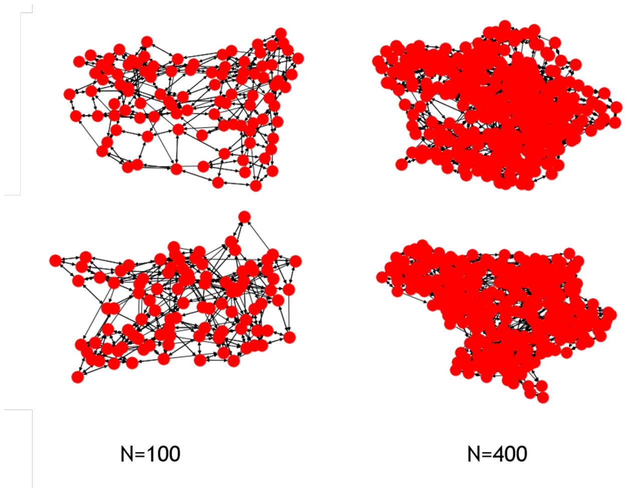


Figure 6: Example graphs of NetworkX small-world networks with increasing square lattice side length (N).

parallel mode. Additionally, we have also explored the impact of the scale of the weight range as the runtime is dependent upon the length of the longest path. The bottom of Fig. 7 compares Loihi and Fugu CPU simulations while increasing the sampling range of the randomly selected edge weights. As shown, the impact of Loihi is minimal, whereas the Fugu CPU simulation incurs an increase in number of timesteps and the time for each timestep.

5 CONCLUSION

In considering the advantages Beyond Moore's Law technologies such as NMC may have, we see the inclusion of the *'context of the computation'* as incredibly important. Not only does the context provide a more realistic assessment of the technology, but can also capture the broader impact where an advantage may come from more than a minimal core computation. Or even when a key computation can be performed efficiently, other factors can be limiting.

Accordingly, Fugu design principles such as compositionality allow us to explore not only core computations, but also application context. This includes incorporating rigorous physics simulation details such as demonstrated in the Neuromorphic Random Walk (Particle Angular Fluence) Mini-App, as well as exploring scaling of the application. The latter is not only important to understand realistically sized scientific computations, but additionally because prior work has shown a neuromorphic advantage may require considering problem setup and scale as well as the computation [24; 33].

Benchmarking is data from an architecture whereas mini-apps yield data about an architecture. While the former is a subset of the latter, they offer different insights. For example, running a ResNet50 neural network (with a specified input size) on an architecture provides performance detail about that one network - how fast an inference may be computed, the associated energy consumption, etc. A broader mini-app however, can directly enable scaling properties of the application such as the size of input,

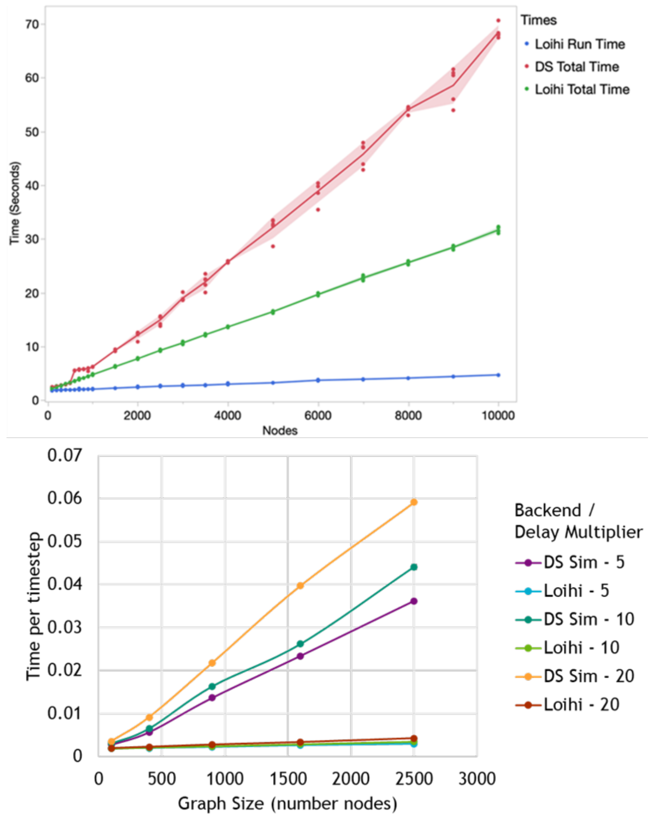


Figure 7: Scaling of Graph Analytics Mini-App comparing CPU (DS) and NMC. Top: comparison of run time as a function of increased tree size. Bottom: comparison of run time with increased graph edge distances.

depth of the network, or various other meaningful ways a Neural Mini-App may be adjusted (each of which would be an individual neural network configuration). Together this data yields broader insight into the performance of the architecture. Given that NMC architectures are actively being explored, Neural Mini-Apps for NMC are an important contribution for understanding the architectures more thoroughly, and not just their performance. For example, consider the various ways in which a neuromorphic architecture may progress. Starting from a baseline many-core approach, the next generation may improve one or many performance factors. Scaling may be enabled by employing a multi-chip architecture which connects several of the original many-core chips together, or by increasing the neural density. Alternatively, increased communication may be the emphasis of architectural advancement. Or overall, the progression may be a compound change improving several features or even moving to new material or manufacturing technologies. Any of the many architectural changes will impact application performance, and the ability to tune the configuration of a Neural Mini-App offers a means of understanding the impact. Alternatively, while benchmarks provide very precise information about a specific computation, that narrow detail is more pertinent for established architectures or applications whose precise configuration is important. While this concept has been more intrinsically understand in domains like scientific computing, where scalability is a key performance indicator, it is less common for neural networks where many metrics pertain to the computation of a specific neural network on a dataset.

The Neural Mini-Apps introduced here are by no means intended to be exhaustive. They emphasize different computational properties, applications, and complexity. However, we envision Neural Mini-Apps to be an evolving and growing tool set to help advance the understanding of neural algorithms, architectures, and their intersections. To enable this growth by the NMC community we are open sourcing the Fugu tool so that others may contribute Neural Mini-Apps as a growing toolset for researching NMC advantages. Future integration of Fugu with tools like the Intel Lava framework [19] will also enable community development of other NMC libraries as well as hardware backends to further enable NMC comparisons and assessments. In this regard, we envision Neural Mini-Apps to be a community driven and evolving tool for exploring neural algorithms and architectures.

ACKNOWLEDGMENT

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

REFERENCES

[1] James B Aimone, Kathleen E Hamilton, Susan Mniszewski, Leah Reeder, Catherine D Schuman, and William M Severa. 2018. Non-neural network applications for spiking neuromorphic hardware. In *Proceedings of the Third International Workshop on Post Moores Era Supercomputing*. 24–26.

[2] James B Aimone, Yang Ho, Ojas Parekh, Cynthia A Phillips, Ali Pinar, William Severa, and 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*. 35–47.

[3] James B. Aimone, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Helen Xu. 2019. Dynamic Programming with Spiking Neural Computing. In *Proceedings of the International Conference on Neuromorphic Systems* (Knoxville, TN, USA) *(ICONS '19)*. Association for Computing Machinery, New York, NY, USA, Article 20, 9 pages. https://doi.org/10.1145/3354265.3354285

[4] James B Aimone, William Severa, and Craig M Vineyard. 2019. Composing neural algorithms with Fugu. In *Proceedings of the International Conference on Neuromorphic Systems*. 1–8.

[5] Amir Beck and Marc Teboulle. 2009. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences* 2, 1 (2009), 183–202.

[6] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 71–81.

[7] Bingjie Dang, Keqin Liu, Jiadi Zhu, Liying Xu, Teng Zhang, Caidie Cheng, Hong Wang, Yuchao Yang, Yue Hao, and Ru Huang. 2019. Stochastic neuron based on IGZO Schottky diodes for neuromorphic computing. *APL Materials* 7, 7 (2019), 071114.

[8] Mike Davies. 2019. Benchmarks for progress in neuromorphic computing. *Nature Machine Intelligence* 1, 9 (2019), 386–388.

[9] Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R Risbud. 2021. Advancing neuromorphic computing with Loihi: A survey of results and outlook. *Proc. IEEE* 109, 5 (2021), 911–934.

[10] Jeff Dean, David Patterson, and Cliff Young. 2018. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro* 38, 2 (2018), 21–29.

[11] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[12] Bradley Efron, Trevor Hastie, Iain Johnstone, and Robert Tibshirani. 2004. Least angle regression. *The Annals of statistics* 32, 2 (2004), 407–499.

[13] Kaitlin L Fair, Daniel R Mendat, Andreas G Andreou, Christopher J Rozell, Justin Romberg, and David V Anderson. 2019. Sparse coding using the locally competitive algorithm on the TrueNorth neurosynaptic system. *Frontiers in neuroscience* 13 (2019), 754.

[14] K. E. Hamilton, C. D. Schuman, S. R. Young, N. Imam, and T. S. Humble. 2018. Neural Networks and Graph Algorithms with Next-Generation Processors. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1194–1203.

[15] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[16] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574* 3 (2009).

[17] Oleksandr Iaroshenko and Andrew T Sornborger. 2021. Binary Operations on Neuromorphic Hardware with Application to Linear Algebraic Operations and Stochastic Equations. *arXiv preprint arXiv:2103.09198* (2021).

[18] Bill Kay, Prasanna Date, and Catherine Schuman. 2020. Neuromorphic graph algorithms: Extracting longest shortest paths and minimum spanning trees. In *Proceedings of the Neuro-inspired Computational Elements Workshop*. 1–6.

[19] Lava-Nc. [n. d.]. lava-nc/lava: A Software Framework for Neuromorphic Computing. https://github.com/lava-nc/lava

[20] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. 2010. Online Learning for Matrix Factorization and Sparse Coding. *Journal of Machine Learning Research* 11 (January 2010), 19–60.

[21] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. 2020. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro* 40, 2 (2020), 8–16.

[22] Bruno A. Olshausen and David J. Field. 1997. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research* 37, 23 (1997), 3311 – 3325. https://doi.org/10.1016/S0042-6989(97)00169-7

[23] Christoph Ostrau, Christian Klarhorst, Michael Thies, and Ulrich Rückert. 2020. Benchmarking of neuromorphic hardware systems. In *Proceedings of the Neuro-inspired Computational Elements Workshop*. 1–4.

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

[25] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.

[26] Christopher J Rozell, Don H Johnson, Richard G Baraniuk, and Bruno A Olshausen. 2008. Sparse coding via thresholding and local competition in neural circuits. *Neural computation* 20, 10 (2008), 2526–2563.

[27] 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 *Proceedings of the 7th Annual Neuro-inspired Computational Elements Workshop*. 1–6.

[28] Samuel Shapero, Christopher Rozell, and Paul Hasler. 2013. Configurable hardware integrate and fire neurons for sparse approximation. *Neural Networks* 45 (2013), 134–143.

[29] J Darby Smith, Aaron J Hill, Leah Reeder, Brian Franke, Richard B Lehoucq, Ojas D Parekh, William Severa, and James B Aimone. 2021. Neuromorphic scaling advantages for energy-efficient random walk computation. *Arxiv* (2021).

[30] J Darby Smith, William Severa, Aaron J Hill, Leah Reeder, Brian Franke, Richard B Lehoucq, Ojas D Parekh, and James B Aimone. 2020. Solving a steady-state PDE using spiking networks and neuromorphic hardware. In *International Conference on Neuromorphic Systems 2020*. 1–8.

[31] Robert Tibshirani. 1994. Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society, Series B* 58 (1994), 267–288.

[32] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The kokkos ecosystem: Comprehensive performance portability for high performance computing. *Computing in Science & Engineering* 23, 5 (2021), 10–18.

[33] Stephen J Verzi, Fredrick Rothganger, Ojas D Parekh, Tu-Thach Quach, Nadine E Miner, Craig M Vineyard, Conrad D James, and James B Aimone. 2018. Computing with spikes: The advantage of fine-grained timing. *Neural computation* 30, 10 (2018), 2660–2690.

[34] Craig M Vineyard, Sam Green, William M Severa, and Çetin Kaya Koç. 2019. Benchmarking Event-Driven Neuromorphic Architectures. In *Proceedings of the International Conference on Neuromorphic Systems*. 1–5.

[35] Craig M Vineyard, Mark Plagge, and Sam Green. 2020. Comparing Neural Accelerators & Neuromorphic Architectures The False Idol of Operations. In *Proceedings of the Neuro-inspired Computational Elements Workshop*. 1–6.

[36] Qu Yang, Jibin Wu, and Haizhou Li. 2021. Rethinking Benchmarks for Neuromorphic Learning Algorithms. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.