Caspian: A Neuromorphic Development Platform

J. Parker Mitchell
mitchelljp1@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Robert M. Patton
pattonrm@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Catherine D. Schuman
schumancd@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Thomas E. Potok
potokte@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee

ABSTRACT

Current neuromorphic systems often may be difficult to use and costly to deploy. There exists a need for a simple yet flexible neuromorphic development platform which can allow researchers to quickly prototype ideas and applications. Caspian offers a high-level API along with a fast spiking simulator to enable the rapid development of neuromorphic solutions. It further offers an FPGA architecture that allows for simplified deployment – particularly in SWaP (size, weight, and power) constrained environments. Leveraging both software and hardware, Caspian aims to accelerate development and deployment while enabling new researchers to quickly become productive with a spiking neural network system.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; • **Hardware** → **Emerging technologies**.

KEYWORDS

neuromorphic, reconfigurable computing, spiking neural network

ACM Reference Format:

J. Parker Mitchell, Catherine D. Schuman, Robert M. Patton, and Thomas E. Potok. 2020. Caspian: A Neuromorphic Development Platform. In *NICE '20: Neuro-Inspired Computational Elements Workshop, March 17–20, 2020, Heidelberg, Germany*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3381755.3381764>

1 INTRODUCTION

There has been a significant rise in neuromorphic computing research in recent years [21]. The diversity in neuromorphic research

efforts has led to different hardware implementations, different software frameworks, and a variety of algorithms. Though there have been some efforts to unify these disparate efforts under a single umbrella (e.g., Applied Brain Research’s Nengo [3]), there is still a significant barrier to entry for those who are new to neuromorphic computing to be able to quickly be able to experiment with a neuromorphic computing framework and apply it to their own applications. With this in mind, we propose Caspian, a neuromorphic development platform with both software and hardware components, that is specifically targeted for new users to rapidly get started using a neuromorphic computing system.

Our focus is on developing a platform that is targeted for a non-neuroscience user. That is, we assume that the user who is interested in understanding and utilizing a neuromorphic system is not interested in running a biologically accurate simulation, but is simply interested in understanding how a neuromorphic platform might be useful to their application. We offer this distinction because those users may be well-served by other software packages, such as Brian [11], NEST [10], etc. For a non-neuroscientist, however, these software packages can be quite difficult to use to simply start investigating the capabilities of spiking neural networks.

We envision that there are two types of ways non-neuroscience users will want to interact with a neuromorphic system: training a spiking neural network using existing training approaches and deploying them, or hand-tooling spiking neural networks to investigate how they might be used for non-machine learning tasks, such as those described in [1]. For the first group, we focus on supporting several existing algorithms and software packages within the Caspian framework, including SLAYER [23], EONS [20], and liquid state machines. For the second group of use cases, we focus on providing an easy-to-use network representation and a highly optimized hardware-accurate software simulator that allows for rapid SNN simulation and prototyping.

In Section 2 we discuss other software and hardware efforts in neuromorphic computing, with a specific focus on how and why they can be difficult to use, particularly for newcomers to neuromorphic computing, motivating the need for a neuromorphic development platform such as Caspian. In Section 3, we discuss the SNN model that Caspian implements and why that model was chosen. Sections 4 and 5 discuss the Caspian software and hardware, respectively, providing details on the current state of the system, as well as the plans for future development. In Section 6, we demonstrate the performance of the Caspian simulator, as well

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>).

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

NICE '20, March 17–20, 2020, Heidelberg, Germany

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. <https://doi.org/10.1145/3381755.3381764>

as results using Caspian with training approaches such as SLAYER and EONS.

2 BACKGROUND AND RELATED WORK

For a newcomer to neuromorphic with a computer science background, the landscape can seem daunting. There are a number of different software packages and hardware implementations each with their own terminologies and APIs. There are a few efforts to help unify multiple simulators and hardware architectures under a common API. One such effort is PyNN. PyNN is a Python API for designing neural networks independent of the simulator or hardware used [6]. It currently supports NEURON, NEST, and Brian as simulation engines, and it can use SpiNNaker and BrainScaleS hardware. PyNN along with its associated simulators are heavily influenced by neuroscience [10, 11]. In order to use PyNN, the user must already have significant knowledge of spiking neural networks, and the user is required to build out the structure of their network explicitly. PyNN is not designed to be a compute-oriented framework, and perhaps as a result, it does not offer high level machine learning algorithms.

Another prominent framework for neuromorphic development is Nengo. Nengo is developed by Applied Brain Research (ABR) as an all encompassing Python library built on the principles of the Neural Engineering Framework (NEF). It offers multiple simulation engines including a native Python implementation and an OpenCL accelerated implementation. Nengo supports different learning algorithms as well as deep learning integration with Tensorflow. While Nengo is more compute oriented than PyNN, it is heavily tied to the NEF and may be impractical or impossible to use when the NEF is not the desired methodology.

In addition to software frameworks, there are many different neuromorphic hardware platforms including FPGA-based designs, digital VLSI designs, and mixed-signal designs. At one end of the spectrum, SpiNNaker is a multi-core ARM-based digital design with custom networking designed to allow highly scalable spiking simulations. By using ARM cores [9], SpiNNaker can execute a wide range of neurons and synapse models. Other digital designs such as IBM TrueNorth [2] and Intel Loihi [5] instead favor a neurosynaptic core approach in which the compute elements are customized for neuromorphic. However, analog and mixed-signal hardware is also a promising approach for neuromorphic computing. For example, BrainScaleS uses wafer-scale integration of analog neurons and synapses to perform large spiking neural network simulations at faster than real-time speeds [14]. Unfortunately, these specialized integrated circuit designs are expensive and require manufacturing and distributing custom hardware, so one alternative is to target FPGA platforms. There are several examples of FPGA targeted designs including DANNA [7], DANNA2 [15], NeuroFlow [4], and NengoFPGA [16]. These designs can be upgraded over time if requirements change and can also be deployed using commodity hardware.

3 CASPIAN

The primary goal for Caspian is to provide not only easy-to-use software and hardware, but also a simple enough model for new users to understand that also allows for sufficient complexity in order to perform SNN computation. In particular, we want a user to

Parameter	Range	Bits
Threshold	[0, 255]	8
Axonal Delay	[0, 15]	4
Leak Time Constant	None, 1, 2, 4, 8, or 16	3

Table 1: The default neuron parameter ranges for Caspian.

Parameter	Range	Bits
Weight	[-128, 127]	8
Synaptic Delay	[0, 15]	4

Table 2: The default synaptic parameter ranges for Caspian.

be able to easily conceptualize what effect their parameter settings will have on a neuron or synapse, without having a background in neuroscience.

When determining a suitable neuron model for Caspian, it was critical for the model to be both simplistic enough to understand and also computationally fast to evaluate. With this criteria, we selected a Leaky Integrate and Fire (LIF) neuron model. As our goals are focused solely on computation with SNNs, we represent neuron and synapse parameters as unit-less integer quantities. Table 1 gives the default neuron parameter ranges for Caspian. There are three parameters in the Caspian LIF neuron: threshold, axonal delay, and the leak time constant. The threshold value represents the amount of charge required for the neuron to fire and is a non-negative integer value. The axonal delay parameter allows the user to specify a time delay on when the neuron’s output affects the neuron’s outgoing synapses. The leak time constant controls the rate of charge decay over time. Caspian evaluates charge leak with $2^{-t/\tau}$ dynamics rather than $e^{-t/\tau}$, and values of τ are constrained to powers of two. This design choice allows for useful hardware optimizations when using event-based evaluation.

The default Caspian synapse parameter ranges are given in Table 2. There are two parameters for Caspian synapses: weight and synaptic delay. The synaptic weight is given in signed integer quantities. The synaptic weight is directly added to the neuron’s charge value, which is then compared against the neuron’s threshold. For example, if a synapse’s weight value is 10, the synapse’s post-synaptic neuron’s charge value is 0 and threshold is 5, then when the event is received at the post-synaptic neuron, the neuron’s charge value will increase to 10 (larger than the threshold) and cause that neuron to fire. For flexibility in implementation of algorithms, we also include a synaptic delay, so that the user may choose to use the synaptic delay and/or the axonal delay to delay neuron fires traveling through the SNN.

We chose these unit-less representations of threshold, charge, and weight in Caspian’s neurons and synapses to abstract away the underlying details of how these dynamics are captured in bio-inspired simulations (where a neuron’s threshold may be represented by a voltage and a synapse’s weight may be represented by a current). In particular, we abstract away these details because Caspian is not meant to be a tool for simulating biologically realistic neurons and synapses. Instead, we want to make the Caspian model accessible to those who are not familiar with the details of

the operation of neurons and synapses in the brain. Though the dynamics are simplified, they are still complex enough to capture behaviors necessary to implement neuromorphic algorithms that require LIF-like behavior.

4 SOFTWARE

4.1 Network Representation

Caspian networks are represented as a directed graph in which a node represents a neuron and an edge represents a synaptic connection. Although each neuron is assigned a unique integer identifier, the identifiers are not required to be a contiguous sequence. To allow for this flexibility, neurons are stored in a hash table data structure keyed on its unique ID. Each neuron then has adjacency lists for its incoming and outgoing synaptic connections. Currently, each neuron and each synapse contain their unique parameters, but future iterations of Caspian may benefit from adding parameter sharing mechanisms like explicit support for convolutional filters.

Caspian networks can be serialized to a simple JSON file format. This format provides great flexibility and interoperability during development. External tools can easily be created just by using a JSON library to read or write network files. By using JSON, the files also allow for some level of human readability if the network is small. In addition to JSON serializations, Caspian easily integrates with the `networkx` package from Python. This provides a wide set of graph related capabilities when manipulating Caspian networks.

4.2 Simulation

Simulation is an important part of the Caspian platform to enable quick development and prototyping of spiking neural networks. Currently, Caspian offers a single threaded discrete event simulation which offers exact cycle accuracy with hardware implementations. This simulation engine is built on the principles of the high performance DANNA2 simulator [15].

The simulation works as follows. Input fires are queued prior to the start of a simulation call. For each time step, any inputs followed by any internal fires for the given step are processed. To process a fire, the simulator looks up the target neuron as well as the amount of charge to accumulate. If this is the first spike handled for the target neuron during the current step, the neuron must calculate any charge leakage. Then, the neuron may accumulate the incoming spike. If the updated charge exceeds the neuron's programmable threshold, the neuron might spike, but this is not definitely known until all spikes for the current time have been processed. In order to deal with this scenario, the simulator tracks all neurons that may spike in a given step, and it checks the final charge at the end of the step before issuing outgoing spikes. Once a neuron spikes, its charge will reset. The default behavior is to reset to zero. However, Caspian also supports soft reset which decrements the charge by the neuron's threshold value possibly leaving residual charge.

With further development, the Caspian platform may support different simulation engines beyond the current single threaded event simulator. For example, larger networks and densely connected networks may benefit from multi-threaded or GPU-based simulators. One goal for the platform is to allow this implementation flexibility while hiding associated complexity from the end user.

```
1 import caspian
2 net = caspian.Network()
3 net.add_neuron(0, threshold=100, delay=1, leak=2)
4 net.add_neuron(7, threshold=50, delay=0, leak=2)
5 net.add_synapse(0, 7, weight=64, delay=15)
6 net.set_input(input_id=0, nid=0)
7 net.set_output(output_id=0, nid=7)
```

Figure 1: Example code to instantiate neurons and synapses

```
1 sim = caspian.Simulator()
2 sim.configure(net)
3 sim.apply_input(0, charge=120, t=0)
4 sim.apply_input(0, charge=50, t=2)
5 sim.simulate(100)
6 spike_cnt = sim.get_output_count(0)
```

Figure 2: Example code to simulate a Caspian network

4.3 Programming API

All performance-critical code for Caspian is developed in C++ to allow for extensive optimization, but similar to other machine learning frameworks, the primary development API is Python-based. This offers a few key benefits. In particular, a Python API allows for integration with a wide variety of existing libraries of interest. Additionally, it tends to lead to an increased ease of use and faster prototyping than a C or C++ based API.

The Caspian API offers abstractions at multiple levels. A user can choose to instantiate a `Network` object and manually add neurons and synapses using function calls. This gives complete control and facilitates non-machine learning tasks such as spiking-based graph algorithms [12, 18]. Figure 1 demonstrates the basics of creating neurons and synapses with Caspian. The result of this code is a two neuron network with one input neuron, one output neuron, and one synapse connecting the input to the output. Figure 2 shows how one may choose to simulate the network from the prior example. Networks may be executed by configuring either a software simulator or a hardware implementation. The user then must queue inputs to the network and set the length of time to run. After execution, the user is free to access the spike data from any output neuron including precise spike timings or just spike counts. While this allows a great deal of flexibility, many users may prefer to directly use the algorithm-level abstractions.

One available algorithm is a genetic algorithm-based approach called EONS [20]. Given a fitness function to score the suitability of a network, EONS may determine both the topology and parameters for a spiking neural network. Using EONS, a user can train a Caspian network for both classification and control style tasks. The resulting networks from EONS tend to be small, sparsely connected networks well suited to Caspian's design.

Caspian also supports the spiking backpropagation algorithm SLAYER. SLAYER is built on top of PyTorch and uses its own simulation engine based on the Spike Response Model. It allows users

to train deep neural networks with temporal dynamics. To adapt SLAYER to Caspian, we added a spike response kernel to match Caspian's neuron model. We also included weight quantization to generate the necessary integer weights for Caspian.

5 HARDWARE

In addition to offering a software simulation engine, Caspian aims to provide hardware accelerated backends through the same Python API. Similar to how deep learning libraries allow GPU acceleration with minimal code changes, Caspian can provide high level abstractions requiring the user to have no specific hardware knowledge. The hardware accelerators can serve a variety of purposes depending on the desired use case. For example, a user may wish to deploy a trained network to an edge application. For this, Caspian can offer a small, efficient implementation suitable to a size, weight, and power (SWaP) constrained environment. At the other end of the spectrum, a user may wish to accelerate the EONS training process which requires running thousands of networks for a given task. Such a task would have very different requirements and would prioritize high throughput over low SWaP.

Caspian aims to offer different implementations to cater to both sets of needs. Namely, we are developing μ Caspian to enable highly efficient edge deployments of trained spiking neural networks. We also have a roadmap to expand to larger implementations including some which could accommodate running multiple networks in parallel or run large spiking convolutional neural networks generated by SLAYER. With a focus on FPGA designs, it is easy to distribute Caspian to other users who can take advantage of this work using only off-the-shelf components. Further, FPGAs allow us to have a variety of designs and to iterate through those designs with a high degree of freedom.

5.1 μ Caspian

μ Caspian, or "micro-Caspian", is a very small IP core suitable for the smallest FPGAs available. Our primary target device is the Lattice iCE40 UP5k which is an FPGA with 5280 4-input lookup tables (LUTs) and 120 kilobits of embedded block RAM [22]. This is an exceptionally small FPGA, and as a result, it offers an inexpensive entry point for integrating Caspian into a system.

Our early implementation of this architecture can support 256 neurons and 4096 synapses. Each neuron is allowed arbitrary connectivity up to the synaptic limit, and each neuron and synapse can have unique, independent parameters. Neurons may have an 8 bit threshold, configurable leak, and a 4 bit axonal delay. Each synapse has an 8 bit weight, but synaptic delays are not allowed due to increased overhead on this platform. Due to the small size and embedded nature of this FPGA, we expect power consumption to be in the low milliwatt range for a μ Caspian deployment.

While μ Caspian can be accessed and used from the Python API, it really is designed such that an end user can integrate it into their overall system design as a low SWaP neuromorphic system. In such a scenario, the user would either integrate the IP core into a larger FPGA design, or the user may deploy an iCE40 FPGA along with a simple communication link (e.g., UART) to a microcontroller.

5.2 Future Architectures

Beyond μ Caspian, we plan to scale to larger architectures. There are a variety of use cases which we aim to accommodate. For example, networks trained with backpropagation methods tend to have a very large number of neurons and synapses, but these networks also follow connectivity patterns. By leveraging a range of FPGA-based designs, we can build customized variants to accommodate these characteristics while maintaining a common neuron model and common programming API.

Another avenue for future work is a training accelerator. When solving a problem with the EONS algorithm, thousands of networks must be evaluated. Typically, these networks are tested against their target application in populations ranging from 100 to 1000. A large FPGA, or perhaps multi-FPGA, system may be able to accelerate the evaluation process by running an entire population simultaneously through many parallel Caspian cores.

6 RESULTS

In this section, we explore a few results demonstrating the Caspian simulator as well as two compatible algorithms. All tests were executed on a workstation machine with dual Intel Xeon E5-2690v4 processors running Ubuntu 18.04 LTS, gcc 7.4, and Python 3.6.8.

6.1 Simulator Performance

To demonstrate the performance of the Caspian simulator, we compared Caspian to DANNA2. DANNA2's event-driven simulator operates similarly to Caspian, and because of this, we are able to create a test which generates identical activity patterns in both DANNA2 and Caspian.

Our first test is inspired by simulation performance testing in [13]. In this test, we create random networks of N neurons that are fully connected to all other neurons in the network. Synaptic weights are randomly sampled and scaled to an 8-bit signed integer range. All neurons in the network have external input firing rates assigned from a uniform distribution from 0 to 100 Hz. We treat each step as 1 millisecond and simulate the network for a total of 1 second (1000 steps). This test serves as an almost worst case scenario for this simulation methodology due to a lack of network sparsity paired with high firing rates. The result is a very high number of events to separately process. We ran this test for networks of $N = 5$ to $N = 1000$ across 10 random seeds each to evaluate the simulator's scaling behavior. Figure 3 shows the results plotted as synaptic events versus time. Caspian demonstrates improved performance and scalability over DANNA2. For the highest activity networks, Caspian is nearly three times as fast as DANNA2.

As a follow-up test, we used a fixed number of neurons and then varied the probability of a synaptic connection from 0.1 to 1.0 in steps of 0.1. As shown in Figure 4, Caspian and DANNA2 have similar performance for sparse networks, but Caspian scales better as connectivity increases. Though DANNA2 and Caspian share a very similar event driven simulation algorithm, Caspian's implementation improvements, including more efficient and scalable data structures, likely explain the difference in performance.

Figure 3: Demonstration of how execution time scales with the number of synaptic events across a range of network sizes

Figure 4: Simulation times for randomly generated networks with 1000 neurons with varying amounts of synaptic connectivity

6.2 EONS

In order to demonstrate Caspian with EONS, we selected the MAGIC dataset from the UCI machine learning repository [8]. The MAGIC dataset is a Monte Carlo generated dataset simulating high energy gamma particles in a gamma telescope. It consists of 19,020 labeled samples with 10 features per sample. This is a binary classification task to determine if the data sample is signal or background.

Each feature is encoded in a sequence of spikes using a combination of spike count and charge injection. We use up to 10 spikes with charge values ranging from 0 to 255 which is the maximum neuron threshold. This type of spike encoding is further explained in [19]. Each possible output is assigned to a particular output neurons, and a prediction is chosen by selecting the output neuron which spiked the most during simulation.

For training, we used a population of 500 networks with a duration of 100 epochs. We ran 40 separate training sessions with the

Classifier	Training Accuracy	Testing Accuracy
Perceptron	77.41%	77.79%
MLP (100 neurons)	83.96%	82.80%
MLP (500 neurons)	84.86%	83.45%
AdaBoost	85.4%	84.14%
SVM Classifier	82.32%	82.61%
K Nearest Neighbors	85.71%	81.41%
Caspian + EONS	84.65%	83.28%

Table 3: Accuracy of various machine learning algorithms on MAGIC dataset

Figure 5: A Caspian network produced by EONS

same parameters but different starting seeds. The dataset was divided into a 75% training portion and a 25% testing portion. Across the 40 runs, the best network yielded 83.28% accuracy on the testing set, which is comparable with other machine learning and neural network approaches (see Table 3). This network is shown in Figure 5. In the figure, orange nodes represent inputs, blue nodes represent outputs, and green nodes are hidden neurons. Black edges are excitatory synapses while red edges are inhibitory. The resulting structure is quite small and sparse but is able to leverage irregular connectivity patterns to solve the task.

6.3 SLAYER

Using SLAYER with Caspian primarily required two modifications to the standard SLAYER configuration. We updated the spike response kernel to mirror the Caspian neuron dynamics, and we also enabled a form of weight quantization because synaptic weights must be integers. To verify these changes were both correct and would not have a significant negative impact to the training performance, we used the MNIST classification task.

NMNIST is a spiking vision dataset equivalent to MNIST [17]. Each digit is captured with a spiking vision sensor. The sensor sits on a motorized pan and tilt platform which moves in a predefined way for each digit. Each sample is 34 by 34 pixels with two channels of "on" and "off" polarity spikes. The digit is shown for a total of 300 milliseconds. As with standard MNIST, there are 60,000 training examples and 10,000 testing examples.

We trained 10 networks for 20 epochs each using the full 60,000 sample training set. We performed no additional preprocessing or stabilization of the data. For proof of concept purposes, we used a simple network topology of two hidden layers of 500 neurons each.

The average final testing accuracy reported by SLAYER was **98.341%**. After moving the model to the Caspian simulation, the average accuracy dropped by 0.057% to **98.284%**. This small difference is likely due to the integer-based computation within Caspian versus floating point for SLAYER.

Although the Caspian simulator was not originally envisioned with this purpose, the average inference time per instance was **41.3ms** which is **7.26×** real-time inference speed on this task. The simulator had an average throughput of 126 million synaptic events per second, averaged across the testing set on all 10 networks.

7 CONCLUSION

In this work, we introduce Caspian as a compute-oriented neuro-morphic platform. We show how Caspian provides both simplicity and flexibility for prospective neuromorphic developers by using a LIF-style neuron model by default, by offering varying levels of abstractions through a Python API, and by including a high performance event driven simulator. Further, we discuss how FPGA-based hardware implementations integrate into this ecosystem starting with μ Caspian for embedded deployments of Caspian networks. We demonstrate Caspian's compatibility with two vastly different training algorithms – EONS and SLAYER – and we also show simulation scaling improvements over a past event-based simulation effort.

ACKNOWLEDGMENTS

This material is based 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, and by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory.

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 *3rd International Workshop on Post-Moore's Era Supercomputing (PMES 2018)*, Dallas, TX.
- [2] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. 2015. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 10 (2015), 1537–1557.
- [3] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C Stewart, Daniel Rasmussen, Xuan Choo, Aaron Russell Voelker, and Chris Eliasmith. 2014. Nengo: A Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics* 7, 48 (2014). <https://doi.org/10.3389/fninf.2013.00048>
- [4] Kit Cheung, Simon R. Schultz, and Wayne Luk. 2016. NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors. *Frontiers in Neuroscience* 9 (2016), 516. <https://doi.org/10.3389/fnins.2015.00516>
- [5] 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.
- [6] Andrew P. Davison, Daniel Brüderle, Jochen M. Eppler, Jens Kremkow, Eilif Müller, Dejan Pecevski, Laurent U. Perrinet, and Pierre Yger. 2008. PyNN: A Common Interface for Neuronal Network Simulators. *Frontiers in Neuroinformatics* 2 (2008), 3637 – 3642.
- [7] A. Disney, J. Reynolds, C. D. Schuman, A. Klibisz, A. Young, and J. S. Plank. 2016. DANNA: A Neuromorphic Software Ecosystem. *Biologically Inspired Cognitive Architectures* 9 (July 2016), 49–56.
- [8] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. <http://archive.ics.uci.edu/ml>
- [9] Steve B Furber, Francesco Galluppi, Steve Temple, and Luis A Plana. 2014. The spinnaker project. *Proc. IEEE* 102, 5 (2014), 652–665.
- [10] Marc-Oliver Gewaltig and Markus Diesmann. 2007. Nest (neural simulation tool). *Scholarpedia* 2, 4 (2007), 1430.
- [11] Dan FM Goodman and Romain Brette. 2009. The brian simulator. *Frontiers in neuroscience* 3 (2009), 26.
- [12] Kathleen E Hamilton, Tiffany M Mintz, and Catherine D Schuman. 2019. Spike-based primitives for graph algorithms. *arXiv preprint arXiv:1903.10574* (2019).
- [13] Hananel Hazan, Daniel J Saunders, Hassaan Khan, Darpan T Sanghavi, Hava T Siegelmann, and Robert Kozma. 2018. BindsNET: A machine learning-oriented spiking neural networks library in Python. *Frontiers in neuroinformatics* 12 (2018), 89. <https://doi.org/10.3389/fninf.2018.00089>
- [14] K. Meier. 2015. A mixed-signal universal neuromorphic computing system. In *2015 IEEE International Electron Devices Meeting (IEDM)*. 4.6.1–4.6.4. <https://doi.org/10.1109/IEDM.2015.7409627>
- [15] J. P. Mitchell, M. E. Dean, G. Bruer, J. S. Plank, and G. S. Rose. 2018. DANNA 2: Dynamic Adaptive Neural Network Arrays. In *International Conference on Neuromorphic Computing Systems*. ACM, Knoxville, TN. <https://doi.org/10.1145/3229884.3229894>
- [16] Benjamin Morcos, Terrence C Stewart, Chris Eliasmith, and Nachiket Kapre. 2018. Implementing NEF Neural Networks on Embedded FPGAs. In *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 22–29.
- [17] Garrick Orchard, Ajinkya Jayawant, Gregory K. Cohen, and Nitish Thakor. 2015. Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades. *Frontiers in Neuroscience* 9 (2015), 437. <https://doi.org/10.3389/fnins.2015.00437>
- [18] 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*. ACM, 3.
- [19] Catherine D Schuman, James S Plank, Grant Bruer, and Jeremy Anantharaj. 2019. Non-Traditional Input Encoding Schemes for Spiking Neuromorphic Systems. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–10.
- [20] 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.
- [21] 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).
- [22] Lattice Semiconductor. [n.d.]. iCE40 UltraPlus. <http://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40UltraPlus>
- [23] Sumit Bam Shrestha and Garrick Orchard. 2018. SLAYER: Spike layer error reassignment in time. In *Advances in Neural Information Processing Systems*. 1412–1421.