This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LOCS.2018.2885976, IEEE Letters of the Computer Society

The TENNLab Exploratory Neuromorphic Computing Framework

James S. Plank, *Member, IEEE,* Catherine D. Schuman, *Member, IEEE,* Grant Bruer, *Student Member, IEEE,* Mark E. Dean, *Fellow, IEEE,* and Garrett S. Rose, *Member, IEEE*

Abstract—Spiking, neuromorphic computing systems are in a period of active exploration by the computing community. While they feature computational expressiveness beyond both von Neumann computing models and feed-forward neural networks, they are also challenging to design and program. The TENNLab exploratory neuromorphic computing framework is a software infrastructure, soon to be open-source, whose goal is to enable potential users of spiking, neuromorphic computing systems to develop applications and evaluate computing architectures, and for architecture researchers to develop and evaluate their architectures with a variety of applications. In this letter, we present the software architecture of the TENNLab framework.

Index Terms—Neuromorphic computing, spiking recurrent neural networks, machine learning, beyond Moore's Law.

1 Introduction

With the demise of Moore's Law and the success of Deep Learning has come a renewed interest in exploring unconventional, but powerful computing architectures. One class of these architectures is termed "Neuromorphic Computing Systems," because of their inspiration from the human brain. Neuromorphic computing systems process temporal spiking events in place of the fetch-and-execute cycle of a von Neumann computer, or static assignment of input values in a Deep Learning system. The spikes have values (amplitudes) and are processed by a fabric of neurons, which accumulate values from their incoming spikes, until their accumulators reach predetermined thresholds, at which point they produce spiking "fire" events. Neurons are connected by synapses, which carry outgoing spikes from one neuron to be applied as incoming spikes to another neuron. While there are additional features which can enrich a neuromorphic computing system, such as leaky neurons or plastic synapses, all spiking neuromorphic computing systems share this fabric of neurons, synapses and spikes.

Neuromorphic computing systems feature a high amount of parallelism, and their computing power is rich, being termed "Super-Turing" by Cabessa and Siegelmann [1]. As an example, certain classification applications have been developed on neuromorphic systems that achieve comparable accuracy to Deep Learning systems, but with over 100 times fewer components [17]. Additionally, neurons and synapses are typically simple to build. For example, recent research projects have explored memristors, biomemetic substrates and optoelectronics, among other devices, to implement neurons and synapses in neuromorphic processors [3], [13], [19]. A more complete listing of neuromorphic hardware research projects has been provided by Schuman *et al* in 2017 [18]. To summarize, neuromorphic processors are attractive as low-power devices with high computational complexity. Application areas often focus on real-time control, IOT, or data processing at a data source, as these areas feature the requirements of complex processing and low power.

The biggest challenge with neuromorphic systems is how to program them. There have been both research and commercial products which employ the Deep Learning approach of programming with backpropagation [6], [14]; however, these approaches limit the networks to being feed-forward, which jettisons the computational advantages of highly recurrent networks. Current exploratory approaches for programming recurrent neuromorphic systems include competitive algorithms [20], reservoir computing [7], genetic algorithms [21], spike timing-dependent plasticity [8] and custom algorithm design [11], [12]. Unlike Deep Learning, which has a host of programming environments such as TensorFlow, Microsoft Cognitive Toolkit and Keras, there are no general software environments for neuromorphic computing. PyNN [5] is a general Python-based interface to neuromorphic systems which has been inspirational to our work. We intend to explore PyNN when we develop Python front-end interfaces to our framework. The works cited above all employ custom-built software.

In this paper, we describe the TENNLab exploratory neuromorphic computing framework. The framework provides interfaces and software support for the development and testing of both neuromorphic applications and neuromorphic devices. The programming approach utilizes a genetic algorithm called *Evolutionary Optimization of Neuromorphic Systems (EONS)* [16], [21], which requires minimal support from the application and the device, but otherwise is a general purpose approach. The framework has already been employed to develop over twenty neuromorphic applications and six neuromorphic devices. One feature of the framework is that applications and devices program to a general model, and therefore applications can run on all

- *J. Plank, G. Bruer, M. Dean and G. Rose are with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996.*
  E-mail: jplank@utk.edu
- *C. Schuman is with Oak Ridge National Laboratory.*
- *Manuscript received 15-Aug-2018. Manuscript accepted: 12-Nov-2018. Final manuscript received: 14-Nov-2018.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LOCS.2018.2885976, IEEE Letters of the Computer Society

architectures, and architectures can support all applications. In this paper, we describe the software architecture of the framework. We plan to support the framework as open source software, starting in 2019, and welcome collaborators who wish to explore neuromorphic applications and devices within the framework.

2 The Structure of an Application

Figure 1 shows the main loop of an application running on a neuromorphic device. The application communicates its state, which is composed of values, to the device. This communication is aided by a module in the framework which converts values to spikes and back again. The device accepts input spikes and then processes for a period of time, producing output spikes. These spikes are converted to values which are then interpreted as input to the application, and the loop continues until the application is complete.

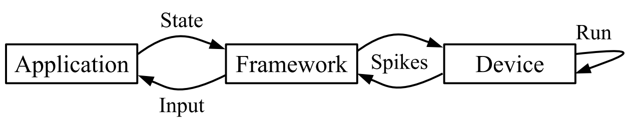

Fig. 1. The main loop of an application running on a neuromorphic device within the TENNLab framework. Applications express states and interpret inputs using values, whereas the devices process spikes.

The framework supports many encodings of values to spikes, including the following:

- **Direct** encoding of the value as spike amplitude.
- **Binning**, by using multiple input neurons for a value, and partitioning the values into bins, each bin going to a specific input neuron.
- **Rate Coding** the value into multiple spikes, where higher values are represented with more spikes.
- **Stochastic Logic**, where values are converted into spike trains, and the decision to spike at each point along the train is assigned randomly with a probability based on the value.
- **Temporal Coding**, where values are converted into two spikes, and the interval between the spikes is determined by the value.
- Combinations of **Direct**, **Binning** and **Rate Coding**.

The framework supports the same encodings for output, except there is no direct encoding, because a neuron or synapse's spike value typically does not change. For **Binning**, multiple output neurons partition each output's value into bins, and the bin that spikes the most is used as the output. Similarly with **Rate Coding**, the number of spikes determines the output value.

The encodings and their various parameterizations may be assigned by the application at runtime.

2.1 An Example Application - Sense-and-Avoid

To help guide the explanation, we present an example TENNLab application, which we call *Sense-and-Avoid*. This is a control application, where a vessel is traveling through space, equipped with a fixed array of LIDAR sensors to detect obstacles. The vessel starts traveling forward, and may boost its power by a fixed amount along any of its (x, y, z) axes. The space through which it travels is populated by moving objects. The goal of a neuromorphic device that controls the vessel is to have it to travel as far forward as it can, while avoiding obstacles and staying within a certain threshold along the y-axis. Figure 2 shows a screen shot of the application.

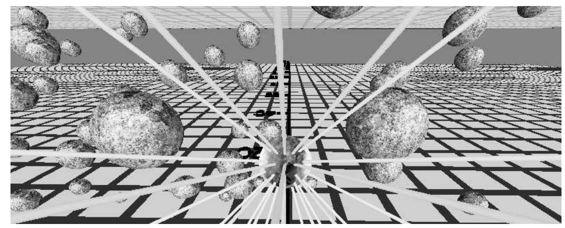

Fig. 2. A screen shot of the Sense-and-Avoid application, where a vessel equipped with LIDAR sensors travels through space (toward the reader), avoiding moving obstacles.

2.2 Application Software Components

Figure 3 shows the software components that an application must implement within the TENNLab framework, and how the components fit in with the other software modules. The application must implement three libraries and two driver programs. The framework implements a separate driver program for training, using the application's libraries. The arrows in the figure denote compilation dependencies of the various drivers and libraries.

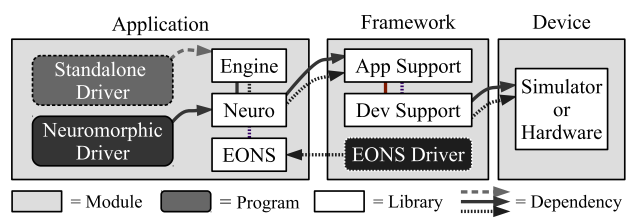

Fig. 3. TENNLab software modules from the application perspective.

The first application library is named **Engine**. It implements functionality specific to the application that is independent of anything neuromorphic. In the case of Sense-and-Avoid, this library implements the physics behind the simulation, the LIDAR sensors and the moving obstacles. A **Standalone Driver** program must also be written so that the application may be executed and tested independent of any neuromorphic components. In the case of control applications like Sense-and-Avoid, a visualization component may be included.

The second library is named **Neuro**. It performs any interaction that the application may have with a neuromorphic device, such as sending state (see Figure 1), receiving input and instructing the device to run. As depicted in Figure 3, the application interacts with an **App Support** module within the framework, which performs the relevant input/output encodings and interactions with the device. For Sense-and-Avoid, the state is composed of the readings from the LIDAR array, and the neuromorphic device controls which of the six directions (if any) to boost. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LOCS.2018.2885976, IEEE Letters of the Computer Society

specification of the neurons, synapses and their relevant parameters is in the form of a network serialization, which may be stored in a file or communicated as a character string. A **Neuromorphic Driver** program must be written, which loads one of these networks onto the device, and runs the application using the device. The network has been created by EONS or by another means of machine learning.

The last application library is named **EONS**. This library must implement a *fitness function*, which receives a network from the framework's **EONS Driver**, and then runs training tests using the **Neuro** library. From the tests it returns a fitness score, which the **EONS Driver** uses to create further networks for the application. For Sense-and-Avoid, the fitness function is composed of several independent runs of the application, each time with a different seed. Each run ends after a time threshold, or when the vessel collides with an object or goes out of field. The runs are scored on their success in getting to the goal, and scores are averaged for a resulting fitness value.

All three libraries must be written to facilitate multithreading from the **EONS Driver**, which can perform multiple fitness runs in parallel. To do so, each library must be partitioned into read-only state that may be shared across fitness runs, and instance-only state that is specific to an individual fitness run. In that way, EONS may leverage all of the cores (and hyperthreads) of a given machine.

3 Device Software Components

Figure 4 shows the software components from the perspective of a device. The device module must implement three libraries that interact with the framework. The first, named **Network**, manages the neuromorphic networks that are sent to the device by the application. Network management tasks include loading networks onto the device, and serializing networks for storage. This library is necessary so that the applications and the framework do not need to understand any device specifics in order to load and store networks.

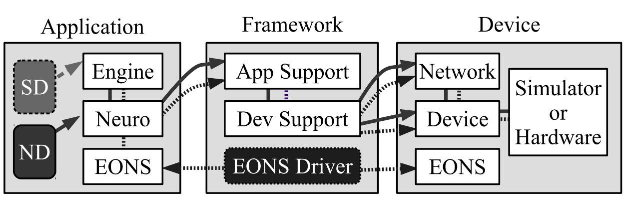

Fig. 4. TENNLab software modules from the perspective of a device.

The second library is named **Device**, and manages the interactions of the neuromorphic device and the framework. These interactions include processing input spikes, communicating output spikes, and having the device run for a specified number of cycles. The spikes are specified by the framework as having floating point values between -1 and 1, which are applied at floating point times relative to the current time. It is up to the **Device** library to convert these numbers to the spike and time units specific to the device. For example, the memristor-based neuromorphic device mrDANNA converts the numbers to discrete values and the times to discrete cycles [2]. In contrast, the NIDA device allows for the floating point values to be sent directly to the device [17].

The **Device** library is the interface to the actual neuromorphic device, whether implemented in simulation or in hardware. For EONS, simulation is typically preferable, to leverage available computing resources [15]. However, for some devices, like Intel's Loihi (as of this writing), hardware is the only available option [4].

The last library is named **EONS**. It interfaces with the framework's **EONS Driver**, by implementing conversions between EONS' representation of a graph and the networks supported by the device (please see the next section).

4 EONS

The **EONS Driver** is a program within the TENNLab framework, whose goal is to train networks for a given application and device. EONS first reads parameters that are specific to an application (e.g., number of inputs and outputs) and to a device (e.g., dimensionality, size), and then it generates an initial population of random graphs according to those parameters. These graphs are the entities on which EONS acts, rather than on networks that are specific to a device. The reason that EONS uses graphs is discussed below. The graphs are then converted into networks for the given device by the device's **EONS Library**. The networks are then passed to the application's **EONS Library** so that the application can determine the fitness of each network. The networks are converted back to graphs, and EONS then orders the graphs by their fitnesses. It generates the next population by selecting graphs of the ordered population and having them reproduce, either by duplicating them, merging them, or having them crossover. Following reproduction, they may mutate as well. These operations are diagrammed in Figure 5. The resulting population is converted to networks, and the process repeats, either until a desired fitness is achieved, or the EONS process is terminated due to time.

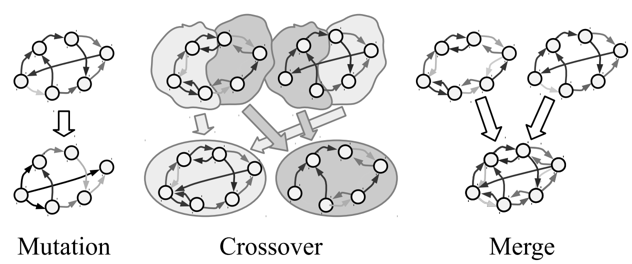

Fig. 5. The genetic operators of EONS, which transform one population of graphs into another.

EONS works on a general graph representation of a neuromorphic network. The graph contains neurons (nodes) and synapses (edges). Each neuron may have any number of parameterized values, as may each synapse. The entire graph may have parameterized values as well. These parameters are defined by the device at the startup to EONS. EONS then applies its genetic operators to the graph structure (the neurons and synapses) and to all of the parameters. In this way, EONS does not have to concern itself with the meaning of the various parameters. For example, neurons can have leak rates and plasticity parameters, and these are optimized by EONS just like any other parameter (such as the threshold) of a neuron.

The decision to have EONS work on a general graph representation was made from experience. At first, the genetic operators had to be written as part of the device module, which allowed them to be customized for each device. This led to very poor genetic operations, because device module authors were not experienced with genetic algorithms, and either wrote very limited genetic operators, or copied them from another device's module. With the general graph representation, device module authors only have to write conversion routines for their network to and from the general graph representation. The "smarts" of the genetic operators are then the purview of the framework. This allows for genetic algorithm features to be written once, within the framework, and apply to all devices implemented within the framework.

Because the EONS process may be computationally expensive, the framework includes a distributed EONS driver that runs over MPI on a cluster environment. We employed this to do a 24-hour EONS run for a robot surveillance application on 18,000 cores of the Titan supercomputer at Oak Ridge National Laboratory [9].

5 Discussion

The primary advantage of the TENNLab framework is its generality. EONS is capable of operating with a variety of devices, architectures, and applications without changing its underlying algorithm. As such, it is easy to apply to new neuromorphic implementations as well as new applications. This reduces the burden on both neuromorphic hardware developers and neuromorphic application developers. Additionally, EONS can be used to investigate the initial capabilities of a hardware platform without requiring the hardware developer to have a broader understanding of training or learning methodologies, either those inspired by machine learning or those inspired by neuroscience. This is important for a software framework, because much of the neuromorphic community is made up of researchers who may not have a background in those fields.

The framework can be applied to explorations of low-power neuromorphic devices for IOT and edge computing. One example is the robot surveillance application mentioned above, where device and robotic simulators were employed to train a network for the DANNA neuromorphic device. The network was loaded onto a battery-powered robot and FPGA-implemented DANNA device, successfully performing an autonomous room surviellance, exclusively under neuromorphic control [9]. This demonstrates a workflow for neuromorphic processors in the field, that incorporates the TENNLab framework.

6 Status

The software framework is written entirely in C++. Its size is roughly 15,000 lines of code. We have developed over 20 applications in the framework, mostly in the domain of control applications; however, we also have general-purpose applications for classification and for event detection in time-series data. There are six devices currently implemented within the framework, with hardware implementations on FPGA/VLSI [10], memristors [15], oil-lipid bimembranes [13], and optoelectronics [19]. The last of these is important because the device module was written by researchers at NIST and not by the TENNLab team. We have funding from Intel to implement a device module for their Loihi neuromorphic processor [4].

The application modules range from 1,000 to 6,000 lines of code, and the device modules average around 5,000 lines of code. The framework contains a simple application (streaming exclusive-or) with a long tutorial walk-through, to aid those developing new applications, and a simple device (NIDA [17]), again with a long tutorial walk-through, to aid those developing new devices. We plan to post the code as open source in 2019.

We are actively researching all facets of the framework, including the effectiveness of other encoding techniques, such as rank order and sparse population encoding. One advantage of the EONS approach is that it composes well with other learning methodologies, such as unsupervised and/or online learning. We are undergoing two explorations in this arena. The first inserts an unsupervised autoencoder between the application's state and the device's inputs. The second explores the effectiveness of STDP as an unsupervised, online learning technique to "hone" the EONS-produced networks, and allow them to adapt.

7 Conclusion

We have described the software architecture of the TENNLab exploratory neuromorphic computing framework. The framework's goal is to support research on applications and devices for spiking, neuromorphic computing systems. The approach to programming applications is a genetic algorithm called EONS, which requires minimal application and device support, but is otherwise general purpose. We plan to post the framework as open source in 2019, and welcome collaboration.

Acknowledgments

This research is supported in part by an Air Force Research Laboratory Information Directorate grant (FA8750-16-1-0065), a grant from Intel Corporation, and 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.[1]

References

[1] J. Cabessa and H. T. Siegelmann. The super-turing computational power of plastic recurrent neural networks. *International Journal of Neural Systems*, 24(8), December 2014.

[2] G. Chakma *et al*. Memristive mixed-signal neuromorphic systems: Energy-efficient learning at the circuit-level. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, PP(99), 2017.

1. Notice: This manuscript has been authored 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.)

[3] S. G. Dahl, R. Ivans, and K. D. Cantley. Modeling memristor radiation interaction events and the effect on neuromorphic learning circuits. In *Inter. Conf. on Neuromorphic Comp. Sys.* ACM, 2018.

[4] M. Davies *et al*. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[5] A. P. Davison *et al*. Pynn: a common interface for neuronal network simulators. *Frontiers in neuroinformatics*, 2, 2009.

[6] S. K. Esser *et al*. Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems*, 2015.

[7] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11), 2002.

[8] T. Masquelier, R. Guyonneau, and S. J. Thorpe. Spike timing dependent plasticity finds the start of repeating patterns in continuous spike trains. *PLoS ONE*, 3(1), 2008.

[9] J. P. Mitchell *et al*. NeoN: Neuromorphic control for autonomous robotic navigation. In *IEEE 5th Int. Symp. on Robotics and Intelligent Sensors*, pages 136–142, October 2017.

[10] J. P. Mitchell *et al*. DANNA 2: Dynamic adaptive neural network arrays. In *Inter. Conf. on Neuromorphic Comp. Sys.* ACM, 2018.

[11] J. V. Monaco and M. M. Vindiola. Integer factorization with a neuromorphic sieve. *CoRR*, abs/1703.03768, 2017.

[12] D. J. Mountain, M. M. McLean, and C. D. Krieger. A comparison between single purpose and flexible neuromorphic processor designs. In *Inter. Conf. on Rebooting Computing*. IEEE, 2017.

[13] J. S. Najem *et al*. Memristive ion channel-doped biomembranes as synaptic mimics. *ACS Nano*, 12(5), March 2018.

[14] E. O. Neftci *et al*. Event-driven random back-propagation: Enabling neuromorphic deep learning machines. *Frontiers in neuroscience*, 11:324, 2017.

[15] J. S. Plank, G. S. Rose, M. E. Dean, C. D. Schuman, and N. C. Cady. A unified hardware/software co-design framework for neuromorphic computing devices and applications. In *Inter. Conf. on Rebooting Computing*. IEEE, 2017.

[16] C. D. Schuman, J. S. Plank, A. Disney, and J. Reynolds. An evolutionary optimization framework for neural networks and neuromorphic architectures. In *International Joint Conference on Neural Networks*, Vancouver, July 2016.

[17] C. D. Schuman *et al*. Neuromorphic computing for temporal scientific data classification. In *Neuromorphic Computing Symposium*, July 2017.

[18] C. D. Schuman *et al*. A survey of neuromorphic computing and neural networks in hardware. arXiv:1705.06963, May 2017.

[19] J. M. Shainline, S. M. Buckley, R. P. Mirin, and S. W. Nam. Superconducting optoelectronic circuits for neuromorphic computing. *Physical Review Applied*, 7:034013, 2017.

[20] S. Shapero and P. Hasler. Neuromorphic hardware for rapid sparse coding. In *BioCAS: IEEE Biomedical Circuits and Systems Conference*, November 2012.

[21] K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.