HRLSim: A High Performance Spiking Neural Network Simulator for GPGPU Clusters

Kirill Minkovich, Corey M. Thibeault, *Student Member, IEEE*, Michael John O'Brien, Aleksey Nogin, Youngkwan Cho, and Narayan Srinivasa, *Senior Member, IEEE*

***Abstract*— Modeling of large-scale spiking neural models is an important tool in the quest to understand brain function and subsequently create real-world applications. This paper describes a spiking neural network simulator environment called HRL Spiking Simulator (HRLSim). This simulator is suitable for implementation on a cluster of general purpose graphical processing units (GPGPUs). Novel aspects of HRLSim are described and an analysis of its performance is provided for various configurations of the cluster. With the advent of inexpensive GPGPU cards and compute power, HRLSim offers an affordable and scalable tool for design, real-time simulation, and analysis of large-scale spiking neural networks.**

***Index Terms*— Distributed graphical processing units (GPUs) programming, general purpose GPU (GPGPU), large scale, neuron simulation, spike timing-dependent plasticity (STDP), spiking neural simulation.**

I. Introduction

COMPUTER modeling of spiking neural networks has an important role in the understanding of brain function. Neurons communicate primarily through action potentials or spikes that are generated by the integration of dendritic synaptic currents caused by spikes from other neurons. There have been several techniques developed to measure neural activity to understand brain function and most approaches are local and limited in their ability to simultaneously measure a large number of neurons [1]. While there have been recent advances in these measurement techniques [2], it is very difficult to simultaneously obtain precise neural data at multiple spatial and temporal scales. The capacity to model biologically plausible models of large-scale neural networks in real-time offers an alternate solution to improve the understanding of brain function. While these models cannot capture all the details of biology, they operate by abstracting the phenomenology of the various cellular and network functions. Using these models, it is possible to analyze behavioral implications of network dynamics and thus make testable predictions for various network topologies and learning dynamics. These models can also offer a path to develop neuromorphic systems with real-world applications.

There are many difficulties to overcome in the modeling of large-scale neural systems such as numerically integrating the governing equations for neurons and synapses, as well as efficiently communicating spiking information between neurons. These inherent difficulties can be further compounded by the need for high-performance and real-time simulations. Although the numerical modeling of spiking neural systems appears to be highly parallel, the models generally do not scale linearly with the number of compute elements because of the strong interdependence of the neurons. We present a general framework for large-scale neural simulations that demonstrates a high level of scaling on general computing architectures. The simulation environment, named HRL Spiking Simulator (HRLSim), was designed for both parallel central processing unit (CPU) architectures and parallel general purpose graphical processing unit (GPGPU) cluster computers.

The motivation for creating a new neural simulator was driven by the need to support the neuromorphic hardware of the SyNAPSE project [3]. A key goal of the project is to implement, in a square centimeter of CMOS, 10^6 neurons with 10^{10} synapses, and an average connectivity of 10^4 synapses per neuron. Recently, as a part of this effort, the HRL SyNAPSE team published a compiler for the automatic translation of a given neural architecture into custom neuromorphic hardware [4]. HRLSim was developed to verify the functional performance of large-scale spiking models. The verified spiking models are then ported onto hardware using the neuromorphic compiler.

A. GPGPU Programming With Compute Unified Device Architecture

GPUs have a large number of single-instruction multiple-data (SIMD) processors capable of efficiently processing huge

Manuscript received February 1, 2013; revised April 19, 2013 and July 1, 2013; accepted July 26, 2013. Date of publication August 15, 2013; date of current version January 10, 2014. This work was supported by the Defense Advanced Research Projects Agency SyNAPSE under Contract HR0011-09-C-001. This work is approved for public release and distribution is unlimited. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the DARPA or the Department of Defense.

K. Minkovich was with the Center for Neural and Emergent Systems, Information and System Sciences Department, HRL Laboratories LLC, Malibu, CA 90265 USA. He is now with Mentor Graphics, Fremont, CA 94538 USA (e-mail: kirill.minkovich@gmail.com).

C. M. Thibeault is with the Center for Neural and Emergent Systems, Information and System Sciences Department, HRL Laboratories LLC, Malibu, CA 90265 USA, and also with the Department of Electrical and Biomedical Engineering, The University of Nevada 89557, Reno (e-mail: cmthibeault@hrl.com).

M. J. O'Brien is with the Center for Neural and Emergent Systems, Information and System Sciences Department, HRL Laboratories LLC, Malibu, CA 90265 USA, and also with The University of California, Los Angeles, Los Angeles, CA 90095 USA (e-mail: mjobrien@hrl.com).

A. Nogin, Y. Cho, and N. Srinivasa are with the Center for Neural and Emergent Systems, Information and System Sciences Department, HRL Laboratories LLC, Malibu, CA 90265 USA (e-mail: anogin@hrl.com; ykcho@hrl.com; nsrinivasa@hrl.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2013.2276056

amounts of data in parallel. In addition, the cost associated in creating clusters of GPUs is considerably lower than CPU-based supercomputers capable of comparable performance [5]. It is not surprise that they are being exploited for general purpose computing.

In the computational neuroscience community, there have been a large number of projects focused on single or dual-GPUs localized to a single compute node. However, there are no GPU-cluster-based neural simulation environments openly available for studying large-scale neural models.

B. Spiking Neural Simulators

For many researchers the choice of neural simulation environment can be tricky due to a tradeoff between biological realism and computational complexity. If access to high-performance computing resources is unavailable, large-scale modeling may be out of reach. In addition, the time investment required for installing and learning a new simulator is a hindrance. With the relatively low cost of GPGPU computing more researchers now have the ability to explore large-scale models. However, the difficulty persists in adopting a new simulation environment.

There are a number of general CPU-based simulators that support large-scale neural models. NEURON [6], [7] and GENESIS [8], [9] are the two most popular simulators. Both offer CPU versions for single and distributed computer environments. Other simulators include: 1) NEST [10]; 2) NCS [11], [12]; and 3) PCSIM [13] (the parallel successor of CSIM [14]). Each of these provides a parallel CPU implementation that is well suited for many distributed environments. However, they do not yet offer a GPU compatible version that can take advantage of large-scale GPGPU clusters. The lack of GPU support for neural simulation resulted in a number of projects focused on creating general environments specific to GPU implementations. Nageswaran *et al.* [15] developed a single GPU spiking neural simulator with a C++ user interface for creating networks. An enhanced version of this single GPU simulator was described in [16]. However, both target a single GPU. Thibeault *et al.* [17] presented a proof-of-concept simulator that targeted multiple GPUs within a single computer. They used a method for distributing the simulations on multiple nodes, which was based on a novel spike message passing scheme that represented the neuron states using individual bits. Izhikevich neurons [18] were supported along with features such as conductance-based synapses and spike-timing-dependent plasticity (STDP).

Similarly, a number of projects have resulted in model-specific GPU implementations. Scorcioni *et al.* [19] presented a single GPU simulator capable of modeling 100 000 Izhikevich neurons, with a fan-out of 100 randomly connected STDP synapses, in real time. Tiesel *et al.* [20] created a single planar network of integrate-and-fire (I&F) neurons using the OpenGL graphics application programming interface (API). Along the same lines, [21] presented a two-layer input–output networks specific to image recognition. Igarashi *et al.* [22] in 2011 developed a heterogeneous model of action selection in the basal ganglia. Two different neuron types are simulated in the model including Izhikevich neurons for the striatum and leaky I&F (LIF) with Hodgkin–Huxley channels for the other areas. The simulation was executed on a single CPU–GPU combination in real time. Richmond *et al.* [23] presented a model with two layers of I&F neurons with recurrent connections. The resulting code demonstrated a speedup as high as 42× over a comparable Python implementation. In this case, the parallelism of the GPU was exploited for parametric optimization. Fidjeland *et al.* [24] presented results for a single GPU simulation similar to [15]. The system could not simulate as many neurons in real time but did demonstrate higher throughput, defined as spike arrivals/s.

Yudanov *et al.* [25] presented a single GPU simulation of Izhikevich neurons integrated using an adaptive Parker–Sochacki method. Emphasis was placed on submillisecond event tracking and accuracy between CPU and GPU implementations. A speedup of 9× was achieved for a comparable CPU implementation. Nere *et al.* [26] presented an extension to a learning model of the mammalian neocortex. The simulation abstracted the neural activity up to the level of neocortical minicolumns using a rate-based model. Synaptic plasticity is only applied to active columns and follows a Hebbian learning rule where the weight matrix between columns is increased if the input is active and decreased if it is not. Simulations were distributed between a single CPU and either a single GPU card or dual GPU cards. The resulting implementation demonstrated a 60× speedup over the single-threaded implementation. De Camargo *et al.* [27] created a GPU simulation of multicompartment conductance-based neurons. The test network was comprised of excitatory pyramidal and inhibitory cells. Each neuron contained two channel conductances modeled using Hodgkin–Huxley dynamics. Variations in the number of connections, weights, and neural activity were explored resulting in a speedup of 40× in some cases over the serial CPU implementation.

In addition to present the first GPU-cluster-based spiking neural simulator, this paper presents several novel concepts in neural simulation that increase both performance and scalability. A description of the modeling and programming elements of HRLSim is provided in Section II. The design details are presented in Section III. Benchmark results for performance of HRLSim are summarized in Section IV. In Section V, possible improvements to HRLSim are discussed, and finally, conclusion is summarized in Section VI.

II. Simulator Description

A. Neural Model Description

HRLSim currently supports two different point neuron implementations: 1) the LIF model and 2) the simple Izhikevich model. The LIF model is defined by

$$C_m \frac{dV}{dt} = g_{\text{leak}}(E_{\text{rest}} - V) + I. \tag{1}$$

C_m is the membrane capacitance; I is the sum of external and synaptic currents; g_{leak} is the conductance of the leak channels; and E_{rest} is the resting potential of the neuron.

As the current input into the model neuron is increased, the membrane voltage increases until the threshold voltage

(V_{thresh}) is reached. At this point, an action potential is fired and the membrane voltage is reset to the voltage reset value (V_{reset}). The neuron has a refractory period of 2 ms. If the current is removed before reaching the threshold, the voltage decays to E_{rest}. The LIF model is one of the least computationally intensive neural models but is still capable of replicating many aspects of neural activity [28].

The Izhikevich neuron model is more costly computationally, but can recreate a larger range of neuron classes [18], [29], [30]. The model is expressed by the simple membrane voltage equation

$$\frac{dV}{dt} = 0.04V^2 + 5V + 140 - u + I \tag{2}$$

a recovery variable

$$\frac{du}{dt} = a(bV - u) \tag{3}$$

and the spike reset rules

$$\text{if V} \geq 30\text{, then} \begin{cases} V \leftarrow c \\ u \leftarrow u + d. \end{cases}$$

a) Conductance-based synapses: HRLSim allows for excitatory and inhibitory synapses. For the ith neuron, and syn $\in$ {exc, inh}, the general form of the synaptic influence on neuron i is defined as follows:

$$I_i^{\text{syn}} = g_i^{\text{syn}} \cdot g_{\max}^{\text{syn}} \cdot (E_{\text{syn}} - V_i). \tag{4}$$

$g_{\max}^{\text{syn}}$ is the maximum conductance for that particular class of synapse; g_i^{syn} is synaptic input to neuron i from that particular class of synapse; and E_{syn} is the reversal potential for that particular class of synapse.

With this notation, the current in (1) and (2) is $I_i = I_i^{\text{exc}} + I_i^{\text{inh}} + I_i^{\text{ext}}$, where the last current is from external sources.

Let $X_j(t) = \sum_k \delta(t - t_j^{\text{AP}_k})$ is the spike train of neuron j as a sum of Dirac functions over the spike times $t_j^{\text{AP}_k}$ of neuron j. With this notation, the following two models can be used for g_i^{syn}. First, users can use the simple synaptic input sum

$$g_i^{\text{syn}} = \sum_{j \in \mathcal{J}_i} W_{ij} X_j(t - \Delta_{ij}) \tag{5}$$

where $\mathcal{J}_i$ is the set of afferents to neuron i; W_{ij} is the synaptic efficacy, or weight, of the synapse from neuron j to neuron i; and Δ_{ij} is the transmission delay from neuron j to neuron i. Users also have the option to simulate the buffering and reuptake of neurotransmitters by modifying the influence a presynaptic action potential has on a neuron using

$$\tau_{\text{syn}} \frac{dg_i^{\text{syn}}}{dt} = -g_i^{\text{syn}} + \sum_{j \in \mathcal{J}_i} W_{ij} X_j(t - \Delta_{ij}). \tag{6}$$

Time constants are user specified for each of the two classes of synapses. In addition, the value of W_{ij} can be modified during the simulation by both long-term and short-term dynamics.

b) STDP rules: Long-term potentiation is modeled using the STDP rules defined by Song *et al.* [31]. The synaptic update rule for the weight between presynaptic neuron j and postsynaptic neuron i, denoted W_{ij}, is given by

$$\dot{W}_{ij} = P_{ij} X_i(t) - D_{ij} X_j(t - \Delta_{ij}) \tag{7}$$

$$\dot{P}_{ij} = -\frac{P_{ij}}{\tau_+} + A_+ X_j(t - \Delta_{ij}) \tag{8}$$

$$\dot{D}_{ij} = -\frac{D_{ij}}{\tau_-} + A_- X_i(t) \tag{9}$$

where P_{ij} is the potentiation trace, tracking the influence of presynaptic spikes, and D_{ij} is the depression trace, tracking the influence of postsynaptic spikes. W_{ij} is then hard bounded to the interval [0, 1]. Users can define A_+, A_-, τ_+, and τ_-.

c) Transmission delays: HRLSim currently supports four different transmission delays: 2, 6, 11, and 21 ms. This choice was made to be compatible with the neuromorphic hardware being developed under the SyNAPSE project, but can be readily extended to cover more fine grained delays if needed. The cost of extending to more transmission delays are in memory, and a slight speed hit. Simulation speed is slightly reduced in checking, which synapses belong to each transmission delay, extra memory management, and the number of variables that must be decayed are increased (however, variable decay has been highly optimized as discussed later). Memory depends on the maximum delay and the total number of possible delays. The spike trains are held in memory until the maximum transmission delay is reached. In addition, the memory required for the plasticity updates increases. Notice that in (9), D_{ij} does not depend on neuron j, and thus only $D_i = D_{ij}$ needs to be stored. On the other hand, in (8), P_{ij} depends on the specific delay Δ_{ij}. Therefore, for the potentiation trace, a distinct copy of the parameter is required for each transmission delay associated with neuron i. This is not overly prohibitive, and HRLSim can easily be configured through preprocessor directives to cope with more transmission delays.

B. User Network Model Description

1) Model Development (C++/PyNN Like Interface): The simulator consists of two main parts: 1) the neuron simulator and 2) the user experiments. The user experiments are further divided in two parts: 1) an .exp text file and 2) the C++ code. The .exp file is used to provide constants to the C++ file, select simulator options (such as using a LIF or Izhikevich neuron model). The simplest .exp file provides the name of the C++ file and of the base class for the user experiment. More sophisticated experiment designs can leverage the .exp file to efficiently perform parameter sweeps and can be used to specify user-defined preprocessor flags for creating highly generalized user experiments without touching any C++ code. The network building parts are described in a PyNN style [32] C++ API, which enables them to be compiled into the simulator. Each user-defined experiment can provide three types of functions: 1) functions to build the network; 2) functions to inject stimuli into the network; and 3) functions to collect statistics about the simulation. These three modules provide a good balance of functionality and complexity needed for simulations and online analysis. The simulator interfaces with the user-defined experiment through a standardized class interface.

2) C++ Preprocessor and Code Generation: The use of C++ in the user experiment allows for faster network building, input generation, and online analysis—all three of which must interface with the main simulator. On the side of the simulator, the C++ compiler allows certain optimizations to be performed such as: 1) loop unrolling; 2) precalculation of constants; and 3) unused simulator code removal. This adapts the simulator to the exact network that is being simulated. Using preprocessor flags, which can be set in the .exp file, a highly optimized simulation executable is generated for the specific experiment through the removal of conditional statements and unnecessary computations. These optimizations make a large difference in GPU performance because memory lookups and branching executions are a common performance bottleneck. The only downside is that with each change to the network, the simulator has to be recompiled, for which the OMake [33] build system is used. Consecutive build times are negligible because only the parts that were affected have to be recompiled. Once the binary is built, constructing and starting the simulation for a network with 100-k neurons and 10-M synapses can be performed in less than 4 s.

C. Input

There are two ways to provide inputs to the simulator: spike and/or current injections. Spike injection is performed through dummy input neurons, which are in turn connected to a set of regular neurons. This allows a single input to be routed to many different neurons. Current injection, on the other hand, is fed directly into the neuron as a floating point addition to the voltage. The single-threaded version of the simulator supports both spike and current injections, whereas the distributed version of the simulator only supports spike injection. This restriction is due to two factors. First, for maximizing the performance the amount of communication had to be reduced. Because current injection uses floating point precision, it would greatly increase the communication requirements in comparison with a spike injection, which can be modeled by a single bit. Second, it would have also required an additional type of message to be exchanged between GPUs, thereby further slowing down the communication. Lifting this restriction is possible, in general, but at a significant performance hit. Because HRLSim was developed to support spike-based neuromorphic hardware, and because of the performance cost of communicating current injection, the decision was made to only support spike injection on the multithreaded platform.

Despite only having spike inputs available to multithreaded simulations, integrating HRLSim with virtual environments, such as CASTLE [34] and WEBOTS [35], is straightforward. The only requirements for HRLSim to interface with an environment is that the environment models stimuli as spike trains, which can be fed into the inputs module, and that the environment can receive binned spike rate averages from the online statistics module, and interpret them as motor commands. The statistics and input generation modules can both be extended by the user (Fig. 1), allowing for easy integration with any virtual environment that fits these criteria. Furthermore, the neural models implemented on HRLSim have been successful in using spike injection (rather than current injection) to drive activity.

D. Analysis

Analysis of the simulations can be completed either while the simulation is running or after the simulation has completed. Analyzing the model during the simulation provides users with the ability to monitor and to some extent refine the model.

1) Online Analysis: The simulator provides the user with hooks for online spike analysis of the running simulation. At the end of each (user defined) statistics printing interval, users can access the total number of spikes that occurred during the interval and the synaptic weight values at the end of the interval. For convenience, this information is available at both the population level as well as the individual neuron level. Online analysis is accomplished through either the default statistics module, or by a custom user-extended statistics module.

2) Offline Analysis: Often with larger models, the statistics take a considerable amount of time to calculate. In addition, visualization offers better insight into the neural model. An offline library is provided for efficient analysis of the neural and synaptic outputs of a completed simulation. The library is written in C++ for higher performance but is accessed with Python through the Boost.Python framework [36]. This feature combines the performance of C++ with the ease of visualization and manipulation that Python provides.

The finished simulation results are analyzed by initially specifying the binary data files (a complete spike history, synaptic weights at user-defined intervals, and the network configuration) to read, and the neuron population, with corresponding indexes, to extract. The analysis object constructor is called by Python but instantiated in C++ for speed. The created object can then be used to gather statistics with the results stored in C++ STL vectors but treated in Python as list objects. These can then be further manipulated or plotted using available Python utilities.

III. Simulator Design

A. Modular Design

Fig. 1 shows the complete breakdown of all the simulation modules with the user-defined modules shown in a lighter color. A simulation starts by creating the main process object which in turn builds a statistics module for logging spikes, printing simulation status, and performing user-specified calculations; an input generation module for user-provided inputs; and a master compute module for building the network, splitting up the network, and performing the simulation. The master compute module uses the BuildNetwork module (which provides the user with APIs to construct the network) and the user experiment to build the network and state modules. The network module is the immutable portion of the network

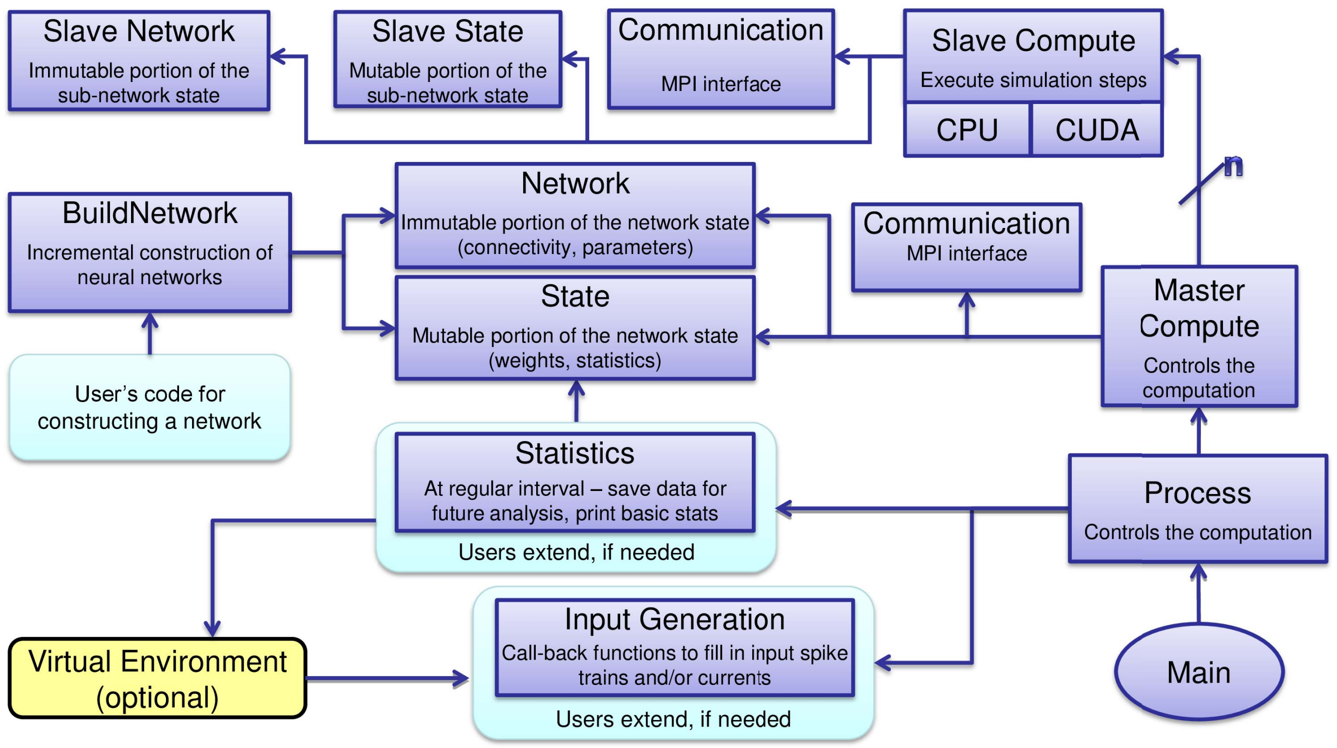

Slave Network
Immutable portion of the sub-network state
Slave State
Mutable portion of the sub-network state
Communication
MPI interface
Slave Compute
Execute simulation steps
CPU
CUDA
n
BuildNetwork
Incremental construction of neural networks
Network
Immutable portion of the network state (connectivity, parameters)
State
Mutable portion of the network state (weights, statistics)
Communication
MPI interface
Master Compute
Controls the computation
User's code for constructing a network
Statistics
At regular interval – save data for future analysis, print basic stats
Users extend, if needed
Process
Controls the computation
Virtual Environment (optional)
Input Generation
Call-back functions to fill in input spike trains and/or currents
Users extend, if needed
Main

Fig. 1. Simulator modules of HRLSim with all the interactions between them.

state, such as connectivity and parameters, whereas the state module is the mutable portion of the network state, such as weights and voltages. This distinction allows for efficient reporting as the network module is only recorded once, at the beginning of the simulation, and the state module is recorded at the end of every statistics interval. After generating the network, the master compute module then splits the network, building a set of communication modules [for performing the message passing interface (MPI) communication] and slave compute modules (for performing the actual simulation).

Once the simulation is initialized, the master compute module is responsible for providing the inputs and parsing the outputs of the neural network by running the user-provided analysis and input generation code. The slave compute modules are responsible for performing the simulation. This distribution of the work leaves the master compute module to have enough computational resources for performing user-defined tasks. Throughout the simulation, the master compute module is continuously writing out three files: 1) network state; 2) network spikes; and 3) synaptic weights. This information is only written at user-specified intervals, giving fine control over how much time is dedicated in saving the state. All information is passed to the master compute node, which then can write the requisite files while the slave nodes continue with the simulation. This allows the simulator to resume from a saved state and it also enables an offline analysis to be performed. Each of these individual features (spike analysis, weight analysis, and state saving) can be toggled on or off by the user to speedup simulations that do not require the data.

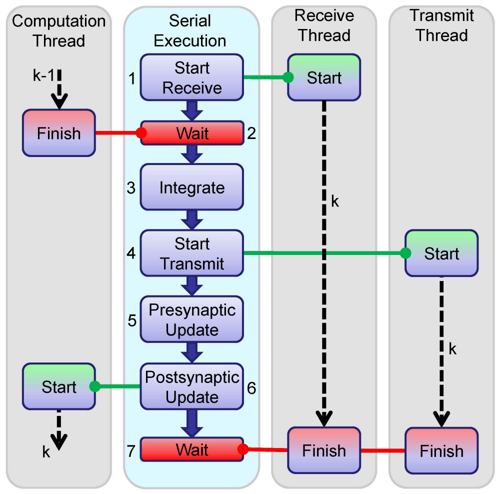


Fig. 2. Flow charts showing how the communication thread is parallelized with the computation thread.

B. Parallelizing Simulation/Communication

To achieve the best performance on the slave nodes the computation has to be parallelized as much as possible with the communication. Fig. 2 shows how the execution of the computation and the communication modules are interleaved. The three gray boxes represent the three parallelized threads: 1) computation; 2) transmitting; and 3) receiving. The blue box features the serialized tasks. When a task can be parallelized, it is branched off and executed in one of the threads, with the dotted line denoting the parallelized task's execution. We define presynaptic updates to be the simulation updates that happen upon the arrival of a presynaptic action potential. Likewise, postsynaptic updates are those that happen following a postsynaptic action potential. With this terminology, the following steps (with corresponding labels in Fig. 2) are used to maximize the amount of parallelization.

1) The receive thread starts receiving incoming spikes for iteration k.
2) The computation thread waits for the postsynaptic updates from iteration $k-1$ to finish.
3) Integration is performed to generate the outgoing spikes.
4) The transmit thread starts transmitting iteration k's outgoing spikes.
5) The presynaptic updates are computed.
6) The postsynaptic updates are computed for iteration k (which are used by the next iteration).
7) The receive and transmit threads wait for incoming spikes to be received and the outgoing spikes to be sent, synchronizing communication.

HRLSim overlaps communication and computation as much as possible. The communication threads are given maximum time to finish while hiding communication latency with computational overhead.

C. MPI Communication

Efficiently passing spiking messages in a neural simulation environment is a topic of interest to a number of simulation projects [37]–[39]. In many of these simulation environments, it is argued that the majority of time is spent in processing rather than communication, so optimizing the latter offers little to no performance benefit [37]. Although this argument is valid in traditional distributed architectures, this is not true when using high-performance GPGPUs because the latter has a much greater computational throughput than the former. Thus, to differentiate HRLSim's communication scheme three of its main aspects will be presented: 1) dummy neurons; 2) message packing; and 3) message passing. These efficient concepts allow for a significant increase in speed.

1) Dummy Neurons: Dummy neurons operate like spike relays by simply passing any incoming spikes as outputs without any integration. These neurons are used when a neuron on one computation resource has connections to neurons on other computation resources. An example of this process is shown in Fig. 3. The use of dummy neurons not only provides a mechanism for compressing the amount of data that has to be transferred, but also removes the need for extra lookup tables to deal with external efferent connections. The dummy neurons are handled just like regular neurons to reduce code complexity. These are similar to the proxy nodes introduced in [38].

2) Message Packing: In addition to reduce the cost of message passing by parallelizing the computation and communication, hybrid message passing provides a guaranteed upper bound to the amount of data sent regardless of the spike rate of the network. This is performed using HRLSim's novel dynamic spike packing, which dynamically switches between the address-event representation (AER) [40] scheme to a bit representation of the outgoing ensemble of neurons.

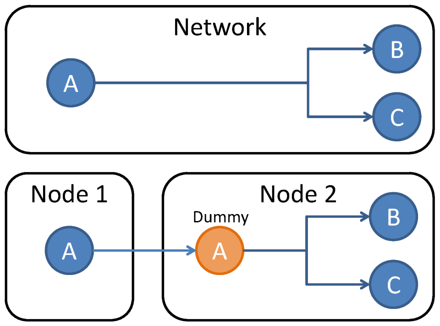


Fig. 3. Example showing how dummy neurons can be used to simplify message passing.

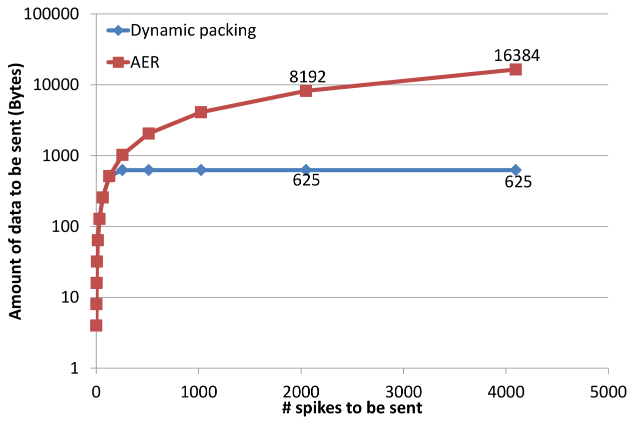


Fig. 4. Dynamic spike packing is compared with the AER approach for simulating a network with 5000 outgoing synapses.

The bit representation scheme essentially encodes the state of these neurons as single bits in an array. A 1 bit represents that the neuron fired, whereas a 0 bit represents it did not. Note that because of the 2-ms refractory period and the voltage integration time step of 1 ms (discussed below), a neuron can only spike once/time bin. Using this scheme, the entire ensemble can be encoded in $N/32$ integers, where N is the number of neurons in the ensemble. To ensure optimal performance, the simulator only uses this scheme when the activity of the ensemble is high enough that the bit representation is smaller than the AER scheme. Fig. 4 shows the difference between dynamic spike packing and AER when simulating a network with 5000 outgoing synapses. This method is similar to [17]. However, the method presented in [17] encoded the state of the entire network at each iteration rather than just the outgoing connections. In addition, the method in [17] did not dynamically switch to the most efficient encoding method. With the dummy neurons described above, the bit representation of the outgoing neurons is sorted based on their outgoing projections. The sorted list enables computation to be performed very efficiently, on GPUs, due to the nonoverlapping memory access and, on CPUs, due to localization, which reduces the number of cache misses.

3) Message Passing: The MPI [41] API facilitates the interprocess communication in HRLSim. During the network initialization phase, the network is split and assigned to the slave compute modules. Along with the requisite network pieces, each slave compute module is also given a postsynaptic target lookup table, which details the nodes that the target neurons live on. Then, after the integration part of the compute cycle, spikes are packaged together with respect to the destination node IDs. These spike packages are then sent using the MPI protocol. In using MPI, several different communication schemes were explored and it was determined that for Infiniband-based communication with GPGPU-based computation the nonblocking point-to-point (P2P) methods MPI_Isend and MPI_Irecv perform best. For the results of this paper, this is the communication method used. However, HRLSim provides four different communication schemes that the user can chose from. The three other communication methods are being further analyzed and their results will be reported in [42].

1) *Blocking P2P:* Separate calls to *MPI_Isend* and *MPI_Recv*.
2) *Nonblocking P2P:* Separate calls to *MPI_Isend* and *MPI_Irecv*.
3) *AlltoAll Collective:* Fixed size message buffers but different data is sent to all receiving nodes.
4) *AlltoAllv Collective:* Variable-sized data buffers. Relies on the MPI implementation to provide performance optimizations.

D. Simulation

1) Integration/Synaptic Updates: Neural simulations can either be time-based or event driven [14]. For time-based simulation, a timestep is selected and each component is simulated for that timestep. The simplicity of this method allows for parallelization and scalability. For event-driven simulations, each component is simulated to figure out the time of the next event, which is inserted into a priority queue. Specifically, this method would determine the next time a neuron should fire and only examine that neuron when it is expected to fire. This method can perform very well when simulating a small network with a low firing rate because there is no wasted work during low activity times. The problem with this approach is that it is difficult to parallelize and thus does not scale easily [38], [43]. For these reasons, HRLSim employs time-based simulation with a 1-ms time step. At every time step, the simulator performs a two-step Euler integration followed by synaptic updates, including the STDP and current summation calculations. In the current implementation of HRLSim, the choices of a 1-ms timestep, and the Euler integration method are project specific, made for both speed and hardware compatibility. Recall that HRLSim was developed to verify spiking models, which are capable of being ported to the neuromorphic hardware [3]. With these choices, interesting neural dynamics have been demonstrated [44], [45], and accuracy has been demonstrated as described in Section V. Despite the SyNAPSE project specific choices made here, HRLSim's high-level architecture does not preclude more sophisticated integration techniques or smaller timesteps.

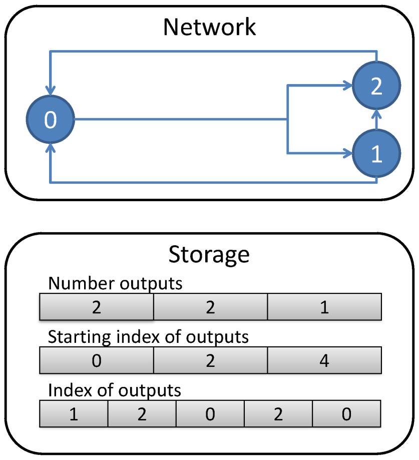


Fig. 5. Conversion from a graph representation of a network to a flattened linear array of the same network.

2) Network Storage/Layout: Using object-oriented code makes network graph creation and connectivity simple. However, in using an object-oriented build API, it is very common that at the completion of the network build, network data is distributed across noncontiguous (scattered) memory. Thus, after the user finishes creating the network graph using HRLSim's API, an additional step is performed to flatten the network into a contiguous vector representation. The transformation is shown in Fig. 5 where a network with three neurons is converted into three vectors: 1) a vector showing the number of outputs for each neuron; 2) a vector showing where to start looking in the synaptic connection vector; and 3) a vector listing the postsynaptic connections of each neuron. These three vectors are used together to describe the connectivity of the network. The master node then distributes the flattened network representation and the corresponding synapses to the slave nodes in a simple ordered way. For N nodes, neuron i is assigned to node $\lfloor i/N \rfloor$, where $\lfloor \cdot \rfloor$ is the floor function. In this distribution scheme, neurons that are in populations together are likely to be assigned to the same node. The assumption here is that a population is more highly connected to itself than to other populations. Future work will involve developing a more intelligent splitting scheme.

Though the flattened network is memory efficient, the original representation can be expansive. In benchmarking the limitations of HRLSim, it was occasionally necessary to use a big-memory node for the master node to build and flatten the network before splitting the representation across the slave nodes. Future work on HRLSim will involve parallelizing the build process across slave nodes both to avoid memory restrictions for the master node and to speedup the process.

3) CPU: HRLSim uses a standard CPU implementation with optimized memory storage (as described above). However, the biggest bottleneck of network scaling is the synaptic updates. The following optimizations were benchmarked using the 20-k network described in Section IV-A.

a) Delayed STDP computation: One of the slowest parts of the CPU simulation is the STDP [31] updates, which are performed at every timestep for each synapse. One part of the calculation that takes a large amount of time is the STDP decay of internal variables P (potentiation) and D (depression). This problem is compounded by transmission delays, which increases the number of decays of P by the number of delay lines. P and D are decayed through the following updates:

$$P \leftarrow P \cdot e^{-1.0/\tau_+}$$
$$D \leftarrow D \cdot e^{-1.0/\tau_-}$$

where τ_+ is the time constant for the LTP (long-term potentiation) window and τ_- is the time constant for the long-term depression window both in milliseconds. The interesting thing is that these variables are only needed when a neuron fires [37], [46]. By delaying the calculation until it is needed, the computation can be grouped and optimized. For example, if X is multiplicatively decayed n times by some constant ρ, then the combined decay would be reduced to

$$X \leftarrow x \cdot \rho^n. \tag{10}$$

Keeping track of n, and using (10) produces a substantial simulation speedup. Employing this optimization technique to D and P reduced the runtime by 68%.

b) Lookup tables: To efficiently compute the decay from (10), ρ^n is precalculated for all $n \leq 512$, where 512 was found empirically to be the best tradeoff. For $n \geq 512$, ρ^n was either approximated as zero or, depending on the value of ρ, the computation was broken down using the following decomposition: $\rho^n = \rho^{\lfloor n/512 \rfloor} \cdot \rho^{n \bmod 512}$. A Taylor expansion method was also considered, but found to be slower and less accurate. In benchmarking these optimizations. Using a five-term Taylor expansion produced a 17% speedup over traditional power calculations while using a lookup table with ρ^n computed for $n \leq 512$ resulted in a 30% speedup. Thus, the precomputation method was selected for both speed and accuracy (adding more terms to the Taylor expansion would yield better accuracy, but at the expense of speed, so this is not a viable solution).

These schemes lead to increased memory access because P and D have to be checked to see if they need to be decayed before they are used. This memory access is easily offset by the reduction in computation for CPUs but on GPUs it poses a large slow down in computation. Besides the additional memory access, this optimization could result in divergent computation on GPUs, which would also affect performance.

4) GPU: The difference between the computations performed on a GPU and CPU stem from the inherent difference in the architectures. The biggest deciding difference is that GPUs have up to 448 cores (NVIDIA Tesla C2075) while CPUs have up to 16 cores (AMD Opteron 2600). The GPU cores are called scalar processors (SP), and are divided into streaming multiprocessors (SM). Thus, the Tesla C2075 has 448 SPs, divided into SMs. While there are many other important differences such as clock speed, memory bandwidth, and memory structure, the number of cores is what truly separates them. An overview of the compute unified device architecture (CUDA) GPU architecture can be found in [15], but some of the terminology and high level ideas will be summarized here.

With hundreds of cores available, every large computation has to be broken down into a set of highly parallelizable tasks. Each parallelizable task is split into a separate function, denoted as a kernel. The idea is that a kernel can be run in mass parallelism with each parallel thread executing on a separate GPU core and operating on (mostly) different data. If a particular value must be updated by multiple threads, atomic operations are used to eliminate race conditions, or incorrect results can occur as a consequence of competing threads attempting to modify the same piece of memory at the same time.

A kernel is concurrently executed by processing threads, which are organized into blocks. A block can have up to 512 threads (1024 for CUDA compute capability 2.0 and above). For programming convenience, thread blocks can be of dimension one, two, or three, as long as the number of total threads does not exceed the block size limit. Because a block of threads has a size limit, many blocks are needed to execute a large number of threads. The blocks must be the same size and shape, and are organized in a grid. When a kernel execution is launched, each block will be assigned to an SM that becomes available. Blocks may execute in any order, thus kernels must be written to be correct for any block order execution. To minimize processor wait time, threads from the blocks are systematically (in a way to optimize memory access) assigned to groups, called warps. If a particular warp requires an expensive memory access, the warp is swapped out, in favor of another warp that can execute immediately, while the memory is being retrieved. To have enough threads to populate a number of warps, multiple blocks are assigned to an SM at a time.

Another capability of modern GPUs is the parallelization of certain GPU operations. To do this, we use what are termed CUDA streams. Each CUDA stream is a queue of GPU operations such as kernel launches and memory copies.

The low number of publicly available GPU-based simulator code suggests that porting existing CPU optimized code to GPUs is difficult. To overcome these differences, the above CUDA features were used to develop a suite of optimizations. These optimizations include: 1) using network statistics for selecting grid/block size; 2) dynamic kernel selection; 3) kernel parallelization; 4) integer approximation; 5) memory optimizations; and 6) communication message packing. These optimizations, described below, require CUDA compute capability 1.2 or higher, and allow for the efficient simulator scheduling as shown in Fig. 2, where each computation block is executed as a kernel.

a) Adjust grid/block size to network size and expected dynamics: GPUs employ single-instruction multiple data parallelism, which means that splitting the data is crucial in achieving the maximum performance on the GPUs. It is common for the GPU code to force the GPU architecture onto the data being processed, but to obtain the best performance the data must be partitioned to match the GPU architecture. To perform this, the structure of the neural network was used to split the data onto the GPU cores. For example, fan-out information was used to split methods that operate over fan-outs. This information is extracted before the simulation starts

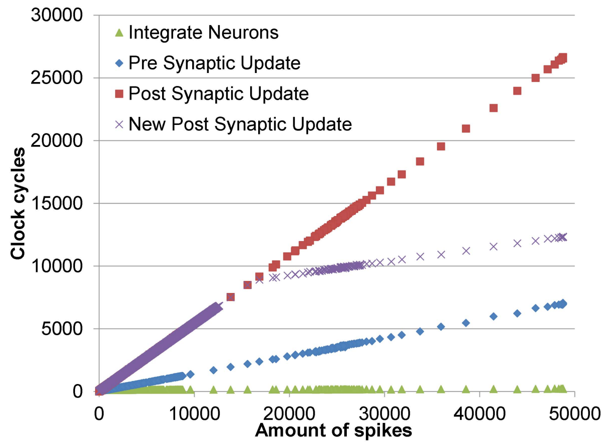


Fig. 6. Graph showing the timing breakdown of a GPU simulation with neuron integration (green), presynaptic updates (blue), simple postsynaptic updates (red), and optimized postsynaptic updates (purple). The postsynaptic updates are slower than the presynaptic updates due to memory access. The synapses are sorted in the presynaptic order, and this increases the efficiency for these updates.

and is used to determine the thread grid and block sizes of every kernel launch.

b) Dynamic method selection based on firing rate: Another way to improve execution time is to align the memory access to the thread execution where consecutive threads access consecutive memory storage. This presents a complication when memory access cannot be easily predicted (e.g., memory access is dependent on firing rate). To address this issue, a dynamic selection algorithm was developed to automatically choose between two methods for postsynaptic updates. Method 1 iterates only on the neurons that just fired while method 2 iterates over all the neurons, only updating the ones that just fired. A benefit to these methods is that the runtime is directly determined by how many neurons fired in the previous iteration. This allows the number of neurons that fired to be the selection criterion for determining, which method should be used. Because the selection is dynamic, it can adjust to changing network conditions or code improvements. To determine the initial transition point during network initialization, two firing rates are chosen to which both methods 1 and 2 are applied. The methods scale linearly as evidenced by Fig. 6 (described below), so a linear fit is constructed for both methods, and the transition point is selected as the intersection of the lines. Once a transition point is selected then there is no overhead of selecting the proper algorithm. If the runtime matches the expected value then no extra work has to happen. If the runtime does not match, the transition point is adjusted using the same technique as in initialization, but using the most recent history of data points.

Fig. 6 shows the runtime breakdown of the main kernels that make up the CUDA computation: 1) integration; 2) presynaptic; and 3) postsynaptic updating. New postsynaptic update is created by dynamically selecting between method 1 and 2. When simulating the 100-k example from Section IV-A for a 30 virtual seconds it was observed that if only the method 1 was used then the runtime would be 26 s and if method 2 was used then the runtime would be 167 s (due to the majority of iterations having a low firing rate), but if the hybrid postsynaptic method is used then the runtime would be 21 s.

c) Kernel parallelization: It was observed that for many CUDA function calls, the output was not immediately needed. For example, the output of the synaptic update functions is not needed until the next integration step. In addition, many of the memory operations were not required to be completed until the next iteration. To take advantage of these features, HRLSim uses multiple CUDA streams. If separate kernels do not depend on large memory copies, then they can be assigned to separate streams and be executed in parallel. In HRLSim, streams are used to launch all the synaptic update functions in the background and to parallelize as much of the memory access as possible. Note that this feature is extremely important because communication between the CPU and the GPU is slow, often being the main bottleneck in GPU-based computing.

d) Integer approximation: Many of the initial Tesla GPU cards, like the C1060, did not support operations such as floating-point atomic addition operating on 32-bit words in global and shared memory. These operations are crucial in many neural computation steps in HRLSim. Fortunately, there is support for integer atomic operation on the Tesla C1060, and thus many of the floating point operations were converted into integer operations by determining the necessary precision. Although, this resulted in a small functional difference from CPU computation, the alternative of performing the computation serially or in a tree fashion would have been unacceptably slow. As HRLSim is currently implemented using atomic integer operations, a card with CUDA compute capability of 1.2 or higher is required. The Tesla C1060 can support CUDA compute capability up to 1.3, though much of the code could be simplified if atomic float operations are available (CUDA compute capability 2.× or higher).

e) Memory optimization: HRLSim optimizes memory access in two ways. First, all the synapse related data structures are aligned to 128-bits because single instruction memory reads can be up to 128 bits. By preventing overlapping access to memory, the total runtime was reduced by about 1%. Second, all the memory allocations are based on the assumed sizes and not on the maximum size. This allows the memory footprint to be greatly reduced, with the rare penalty that the memory might need to be copied to a larger vector. This optimization was essential for efficiently packing the outgoing spike vectors.

f) Communication message packing: When comparing the amount of data that is produced by the GPU computation unit, versus a CPU one, the ratio is over 10 times. The GPU computation has to undertake some of the communication complexity to prevent the communication thread from being overwhelmed. A special kernel is employed to take the newly generated spikes and pack them into messages to be sent to other nodes.

IV. Performance Evaluation

A. Large-Scale Neural Model

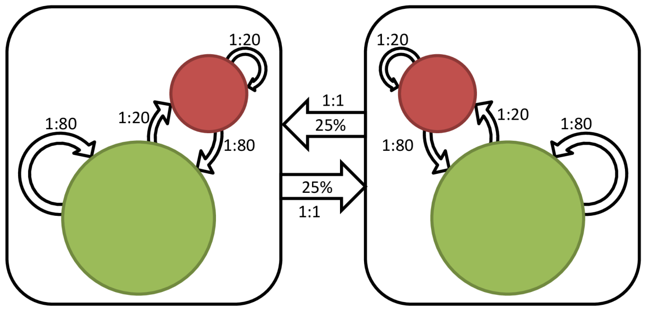
The selection of a neural network architecture has a large effect on how well a simulator can perform. For example,

TABLE I
NETWORK PARAMETERS USED IN THE BENCHMARKS

All Networks	
$C_{\mathrm{m}} = 200$ pF	$g_{\mathrm{leak}} = 10$ nS
$E_{\mathrm{inh}} = -80$ mV	$E_{\mathrm{exc}} = 0$ mV
$V_{\mathrm{thresh}} = -54$ mV	$V_{\mathrm{reset}} = -60$ mV
$E_{\mathrm{rest}} = -74$ mV	fanout = 100
$\tau_{\mathrm{exc}} = 5$ ms	$\tau_{\mathrm{inh}} = 100$ ms
$A_{+} = .025$	$A_{-} = .0265$
$\tau_{+} = 20$ ms	$\tau_{-} = 20$ ms
$W_{\mathrm{local}} = .5$	$W_{\mathrm{bridge}} = 0$
$W_{\mathrm{dummy}} = 6.0/g_{\mathrm{max}}$	refract period = 2 ms
100K Neurons per Node Networks	
$g_{\mathrm{max}}^{\mathrm{exc}} = 7$ nS	$g_{\mathrm{max}}^{\mathrm{inh}} = 7$ nS
dummy inputs = 500	isi-mean = 25 ms
20K Neurons per Node Networks	
$g_{\mathrm{max}}^{\mathrm{exc}} = 6.02$ nS	$g_{\mathrm{max}}^{\mathrm{inh}} = 6.02$ nS
dummy inputs = 181	isi-mean = 45.25 ms

a simple network with a tight Gaussian firing rate distribution is easy to simulate and parallelize because each process would be doing a similar amount of work. On the other hand, a network with uncorrelated bursting dynamics is much harder to simulate because each process will be doing a different amount of work. This differentiation in computational effort bounds the simulation time of each iteration to the slowest simulating resource. For example, if two nodes are simulating a network and they take 0.5 and 1 ms, respectively, then the overall runtime (i.e., the time taken to simulate the network) of that iteration would be 1 ms. This property makes it difficult to efficiently scale the simulation of these networks to a large number of nodes because the runtime of each iteration is proportional to the slowest simulation across all the nodes. The bottleneck of any simulation is the slowest (most burdened) node; thus in the following paragraphs, we analyze compute time as a function of the number of spikes a node must process in a given time step. For the following discussion, let $\mathcal{I}$ be the set of all iterations, or simulation timesteps. For $I \in \mathcal{I}$, define the size $||I||$ to be the maximum over the number of spikes generated on each node. Then, define the max node of an iteration I to be the node, which generated $||I||$ spikes during the iteration. If there is a tie, then the slowest of the tied nodes is the max node.

To create these bursting networks, each node simulates an excitatory/inhibitory network of LIF neurons, in the ratio of 80%/20%. Each node simulates either 20 or 100-k neurons with a local fan-out of 100 randomly chosen targets. The local



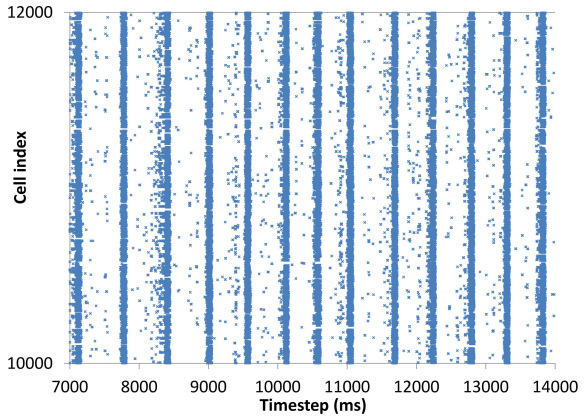


Fig. 7. Top: two 80% excitatory (green) / 20% inhibitory (red) networks connected in a small world fashion. Here, 25% means that 25% of all the neurons have outgoing axons that connect to an external network, with synaptic weights of zero. 1:80, 1:20, and 1:1 show a fan-out of 80, 20, and 1, respectively. Bottom: the raster plot of 2000 neurons from one of these networks showing their bursting firing.

synaptic weights (W_{local}) are all initialized to 0.5, and only the synapses of the connections originating from excitatory neurons are plastic. Furthermore, 25% of the local neurons form an additional single external connection to a randomly chosen neuron on another randomly chosen node (uniform distribution), creating small-world connectivity [47]. Thus, a network with 100-k neurons would have 25 k outgoing connections. Table I shows the parameters used in the model for these benchmarks. To generate network activity within each local network, a set of dummy inputs is created for spike injection. They are connected in a one-to-one fashion to a random subset of the excitatory population. The dummy input synaptic strengths, which are fixed, and number of dummy inputs are shown in Table I. We define an isi-mean for the dummy spikes (the values are shown in Table I). After each interspike time, selected from an exponential distribution with mean given by isi-mean, a single randomly chosen dummy neuron is seeded with a spike. This produces the desired network activity.

Fig. 7 shows how two of these networks would be connected and it also shows burst firing in one of the two networks. To keep the firing rates similar, the connections between the networks were set to a fixed weight (W_{bridge}) of zero, which still resulted in communication but kept the firing rates of the networks independent. When examining the distribution of $\{||I|| : I \in \mathcal{I}\}$, shown in Figs. 8(e), 9(e), and 10(e), there

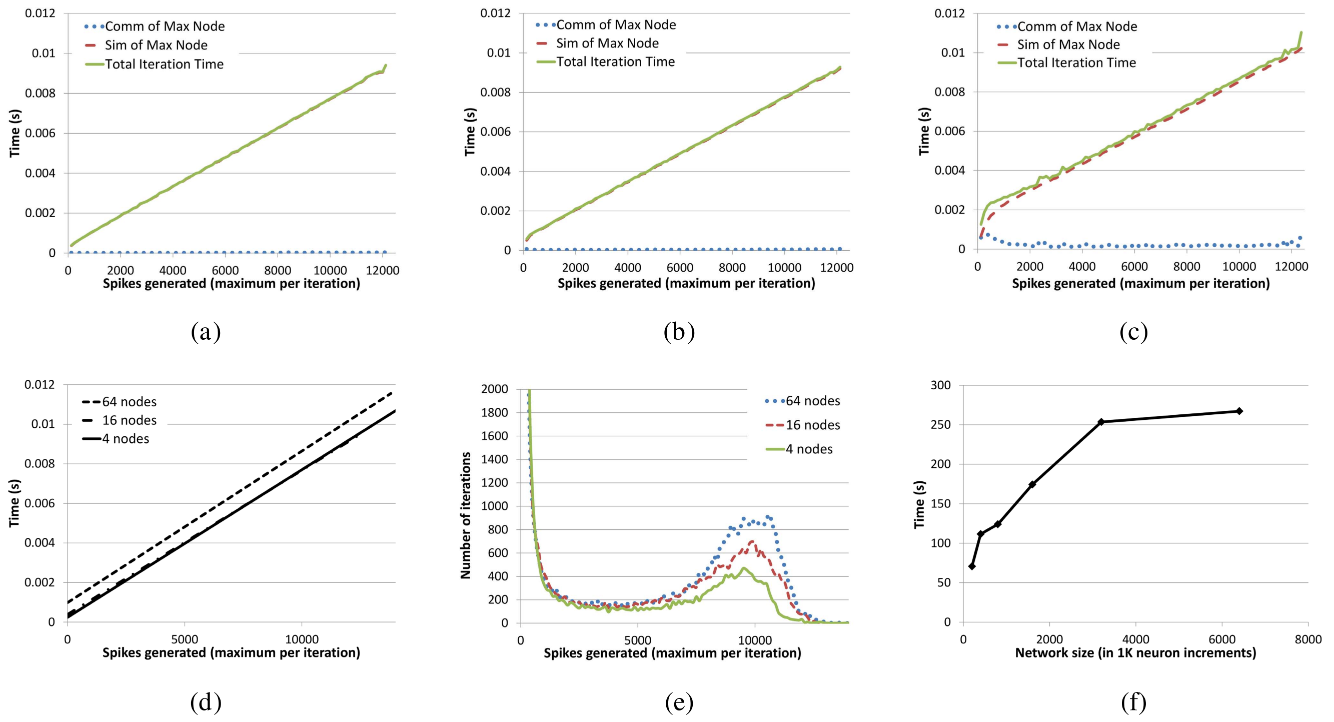


Fig. 8. GPU results, 100 000 neurons/node. (a)–(c) Runtime distribution, with respect to $||I||$, on 4, 16, and 64 nodes (400, 1600, and 6400 K total neurons), respectively. (d) Total-time linear regression for plots (a)–(c). (e) Histogram of $||I||$ over $I \in \mathcal{I}$. (f) Scaling of the runtime to network size.

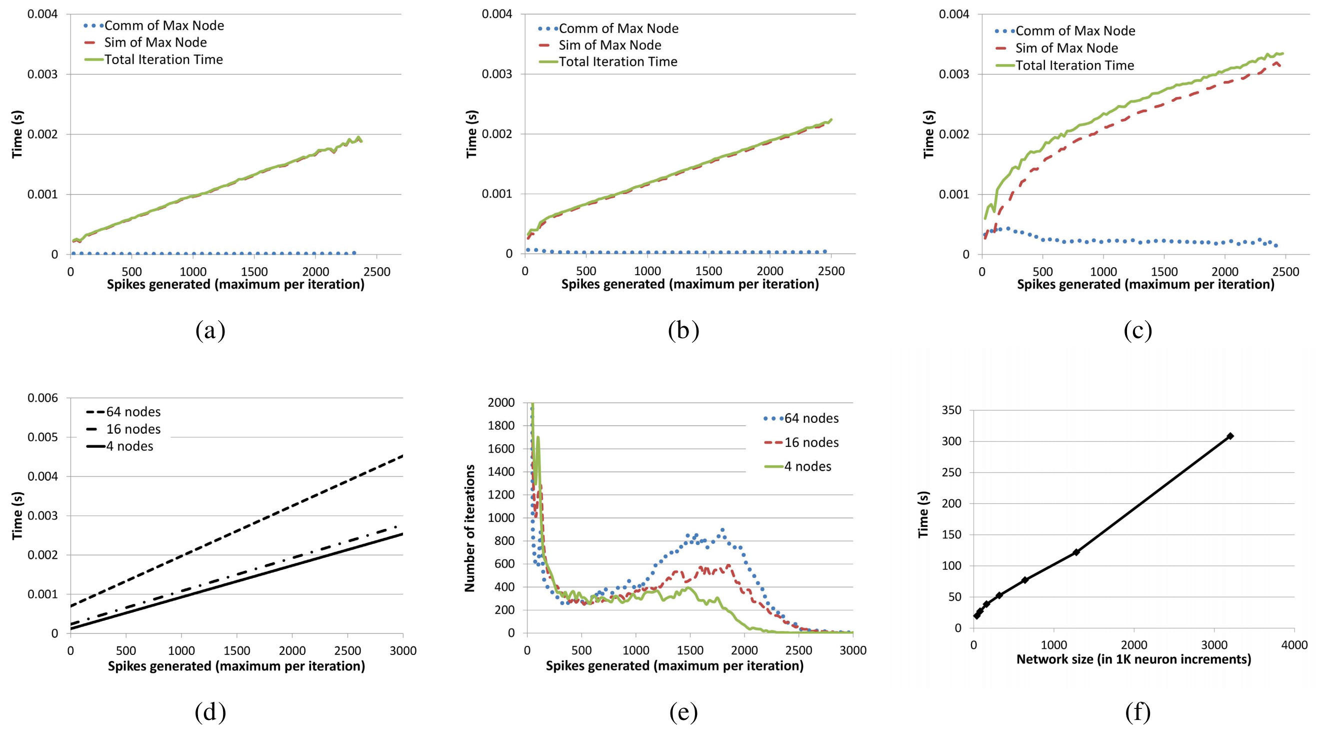
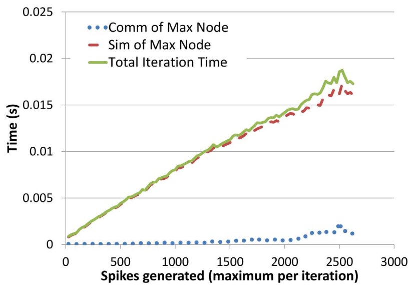


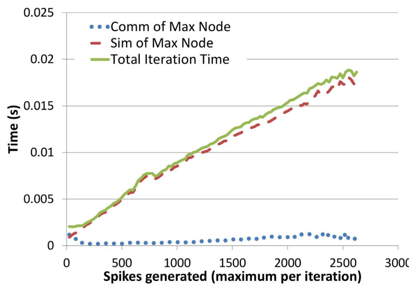
Fig. 9. GPU results, 20 000 neurons/node. (a)–(c) Runtime distribution, with respect to $||I||$, on 4, 16, and 64 nodes (80, 320, and 1280-k total neurons), respectively. (d) Total-time linear regression for plots (a)–(c). (e) Histogram of $||I||$ over $I \in \mathcal{I}$. (f) Scaling of the runtime to network size.

is a clear trend that when more nodes are being used, then Prob $(||I|| > k)$ increases for any positive k. For this reason, this specific type of network is the slowest to simulate and thus was chosen as a worst case scenario for evaluating HRLSim.

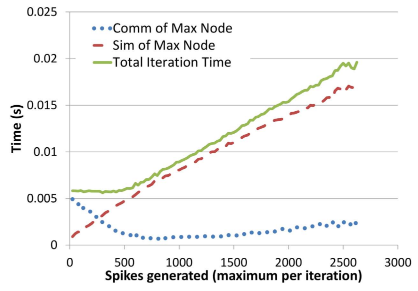
This benchmark was performed on a cluster of 92 compute nodes, each with two Intel Xeon E5520 2.27-GHz CPUs (four-dual thread cores/card) and two NVIDIA Tesla C1060 cards, with an Infiniband communication backend. Each slave node has 12 GB of memory, where the head node used in these



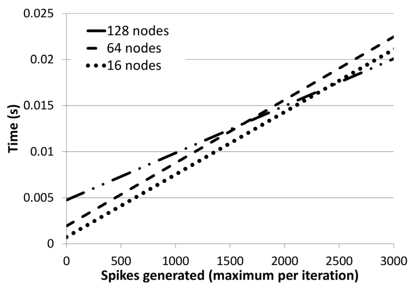
(a)



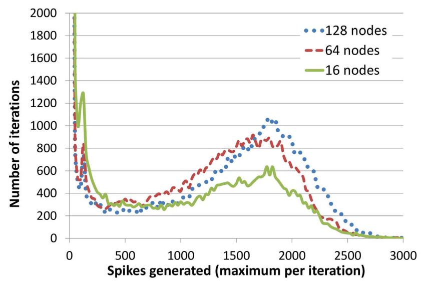
(b)



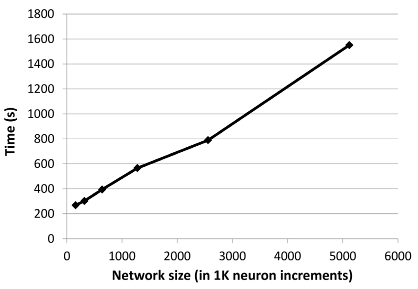
(c)



(d)



(e)



(f)

Fig. 10. CPU results, 20 000 neurons/node. (a)–(c) Runtime distribution, with respect to $||I||$, on 16, 64, and 128 nodes (320, 1280, and 2560-k total neurons), respectively. (d) Total-time linear regression for plots (a)–(c). (e) Histogram of $||I||$ over $I \in \mathcal{I}$. (f) Scaling of the runtime scales to network size.

benchmarks has 48 GB of memory. Note that in the following simulations, it was necessary to use a big-memory node for the master node because of memory constraints (Section III-D2). Despite the limitations imposed by the head node, the entire simulation on each slave node required less than 300 MB of memory (for 100-K neurons with a local fan-out of 100 random neurons).

B. GPU Performance

Using the C1060 card, HRLSim can simulate a 110-k neuron and 11-M synapse network for 100 virtual seconds in 99 s. The following benchmarks are all simulated for 100 virtual seconds, and the results are measured in wall time, or real-time elapsed. For the benchmarks, Master is used for measuring the wall time of a system wide iteration. Each node measures the time it takes to complete a single iteration and sync with Master, as well as the time required for the computation part of the iteration. Each node then records its communication time as its iteration time minus its computation time. Because of the communication with Master, all nodes sync to Master at every iteration.

To test how HRLSim with CUDA computation scales, with respect to network size, it was tested with 2, 4, 8, 16, 32, and 64 GPU cards where each GPU card either simulated a 100 k (Fig. 8) or a 20 k (Fig. 9) spiking neuron network. For brevity, only the results for 4, 16, and 64 GPU cards are presented, though the others are similar. Plots (a)–(c) from Figs. 8 and 9 show: the total wall time (green) to execute a system-wide iteration, I; the wall time required for communication (blue) on the max node; and the wall time required to execute iteration I on the max node as a function of $||I||$. Notice that the total iteration time (green) trends as the sum of the almost constant max node communication time (blue) and the wall time of the max node (red). Communication roughly remains constant because of two factors: 1) HRLSim's message packing creates a hard limit on the amount of data to be transmitted and 2) HRLSim is able to efficiently parallelize the communication and the simulation. The communication of the max node is hidden by the threading, as well as the fact that the max node does not have to wait for the faster nodes (however, the faster nodes have a higher communication cost because they must wait for the slower nodes). All these results were tightly grouped except for Fig. 9(c) where there seems to be two trend lines. Upon further investigation, it was observed that the simulation runtime had two main contributors: 1) integration/synaptic updates and 2) message packing. The problem is that for the 64 node system, during low firing activity, it had two distinct operating states depending on how many outgoing messages needed to be packed. For low firing rate, packing the outgoing messages using the AER method (Section III-C2) took almost as much time as everything else. As the firing rate increases, the message packing penalty becomes constant due to the bit representation, and the integration and synaptic updating dominated the simulation time. This phenomenon was only seen here because of the unusual combination of small network (20 k) and high external connectivity.

Figs. 8(d) and 9(d) are side-by-side comparisons of the linear regressions of the total simulation time for each network size. Figs. 8(e) and 9(e) are the histograms depicting Prob $(||I|| = k)$ for all observed iteration sizes k. Increasing the number of slave nodes used produces a greater probability that one of them will run slowly relative to the other nodes. Notice the vertical translation of the iteration trend lines in

Fig. 8(d) as the number of nodes increases. This is due to the additional number of spike packages that are required to be transferred between the GPU and the CPU. This is a consistent trend throughout all the GPU simulations according to the data, though imperceivable within most of the figures. This effect would be reduced through the use of newer GPU cards because they are designed to handle multiple memory accesses efficiently.

An interesting result was obtained when total simulation runtime was monitored as a function of network scale [as shown in Figs. 8(f) and 9(f)]. The 20 k network shows almost linear scaling but the 100-K network flattens out. This flattening out suggests that as long as there is a slave node taking a significant amount of computation time the communication latency will be hidden due to threading, taking no additional time. As more nodes are added to the network, it becomes more likely that a node is producing a large number of spikes (bursting), as shown in Figs. 8(e) and 9(e). The saturation of slow nodes across the iterations dominates the simulation time, giving the flattening in the 100-k networks. In the 20-k networks, we fully expect this sort of flattening as more data points are included. The 20-k networks do not burst as readily as the 100-k networks, thus the flattening is delayed. These results demonstrate that HRLSim's computation time degrades gracefully (linearly or better) with this type of scaling.

In the above analysis, it was determined that HRLSim scales well with network size. To determine how HRLSim scales with respect to a network of fixed size, a 1.6 million neuron network was divided using 16 and 80 nodes, corresponding to 100 and 20-k neurons/node respectively. The simulation took 174 s using 16 nodes and 148 s using 80 nodes. Although this suggests that using more nodes is more efficient, this result does not extend to 160 nodes (10 k/node) where even the 16 node simulation performs much better due to a reduction in the number of nodes communicating.

C. CPU Performance

To evaluate the CPU simulation core, it is necessary to determine the largest network that it can simulate in real time. Thus, it was determined that HRLSim's CPU core can simulate a network with 20-k neurons and 2-M synapses firing at 10 Hz for 100 virtual seconds in 98 real seconds. This limit of real-time simulation is why the 20-k network was selected for the scalability test and each simulation is simulated for 100 virtual seconds as in the GPU case. To test the scalability of HRLSim using CPU computation, it was tested on 8, 16, 32, 64, 128, and 256 CPU cores where each core simulated a 20-k neuron network (Fig. 10). Plots (a)–(c) from Fig. 10 summarize the results for 16, 64, and 128 cores, respectively. The figures show that starting with 128 cores, just the empty MPI synchronization calls account for a significant amount of runtime, and with 256 cores communication time dominates until when about 5% of the network starts firing (not shown). Even with these high communication costs, simulation time eventually dominates and allows the communication latency to be mostly hidden, adding almost zero time to the iteration. Examining the trend lines corresponding to the total iteration time, Fig. 10(d) shows that as the number of nodes increases so does the initial cost. Though the slope appears to decrease for 128 nodes, this is an artifact of the communication time penalty in the regimes where the max node's iteration time cannot hide it, as observed in Fig. 10(c). In addition, as with the CUDA simulation, there is some spiking imbalance when the number of nodes is increased as Prob $(||I|| > k)$ increases with the number of nodes [Fig. 10(e)]. Overall, the CPU scaling simulation data, as shown in Fig. 10(f), has a favorable linear relationship between the network size and runtime.

D. Network Splitting

In the benchmarks above, the networks were split based on the small-world network topologies, where tightly connected components were assigned to the same computation resource. This corresponds to the optimal way a network can be split but for most other networks such a simple partition is not feasible. To test the worst split possible, each neuron was assigned to a random computation resource. When looking at the 128 node network with 20-k neurons/CPU core, the number of outgoing axons on each node increased over 400× (from 3 to 1385 k) but the simulation time only increased 23% (from 790 to 970 s). This implies that the communication and simulation threading were working correctly and the message packing scheme prevented the network from getting saturated.

E. Memory Consumption

Memory requirements are often a bottleneck when parallelizing large-scale neural simulations across many nodes. Kunkel *et al.* [48] thoroughly examine the memory requirements of NEST for simulating large-scale networks across many processes. Before a thorough investigation into HRLSim's memory requirements, HRLSim's build process needs to be parallelized. Because the build process is all done on Master, the memory of Master (48 GB) limits the size of networks that can be built. Furthermore, HRLSim currently stores a redundant copy of the network on the Master node, even after the split and distribution of the network to the slaves. This allows Master to print and compute statistics, and to record spike and weight histories and the network state. This redundant memory counts against HRLSim in a full memory analysis. After HRLSim's build and data recording processes are parallelized, a full memory analysis, in the spirit of [48], will be done for both CPU and GPU in future versions of the simulator. Here, we report the memory consumption of the 20-k CPU experiment from Section IV-A. We also considered a more realistic network where the fan-out is 10 k, rather than 100, but everything else is kept the same. Table II shows the memory used by both Master and the slave nodes, where four slaves are used. Note that the memory is reported for the actual simulation, after the network representation has been flattened. We used only four slaves nodes (80-k neurons) because building the 10-k connectivity experiment required 47.9 GB of memory on Master. Thus, more than four nodes were not considered.

TABLE II
MEMORY CONSUMED BY HRLSim DURING SIMULATION FOR THE 20-K NEURON NETWORK TOPOLOGY DESCRIBED IN SECTION IV-A

100 Fanout		10K Fanout	
80K Neurons	8M Synapses	80K Neurons	800M Synapses
Master	.413 GB	Master	41.2 GB
Slave	.0892 GB	Slave	3.96 GB

Left column: memory for the 100 fan-out networks used in the benchmarks throughout and right column: memory for a more realistic situation of a 10-k fan-out.

V. DISCUSSION

While the neural models simulated in Section IV are the simplest that would convey the scalability of HRLSim, many other networks have been simulated by the SyNAPSE team members including HRL Laboratories, Boston University, George Mason University, University of California at Irvine, and the Neurosciences Institute. In addition to the testing performed by the team members, the functionality of the simulator was verified against [16] by checking that an 80/20 Izhikevich network had the same firing dynamics on both simulators. This implies that the simulator is accurate.

The original motivation for the HRLSim simulator was to perform simulation of neural networks that would be implemented on SyNAPSE [3] hardware. During the development process, it has evolved into much more than a simulator of a specific neural hardware. It can now be used for evaluating novel neural dynamics, additional biological features, and specialized computational techniques. However, the trend in neural simulation, and software in general, is that the more extensible the design, the more the performance suffers.

GPU simulators, including this one, currently lack a level of extensibility that most developers would desire. In addition, almost all of them are tied to a specific hardware platform. Adding new features can be tedious and difficult to implement, while porting these implementations to new hardware is almost completely unreasonable. The tradeoff between performance and extensibility is a choice that either complicates the design or restricts the environment to specific hardware and features. Projects like OpenCL [49] can abstract away some of the hardware dependence but that comes at a cost in performance. In addition, template-based designs can improve extensibility but that brings an associated increase in code complexity and again, can introduce decreased performance. The need for simulator performance is in direct conflict with the need for extensibility.

HRLSim requires high performance to support the SyNAPSE project and its user base. However, the compiled nature of HRLSim does allow for future extensions without a negative impact on the current performance. This ensures that supporting the needs of future users will not affect the current users but the complexity of implementing those extensions varies.

In addition to the features the simulator currently support, we plan on adding additional features such as: a PyNN interface to simplify importing existing models, additional visualization tools, parallel network building capabilities, parallel statistics calculations, parallel weight/spike history file generation, and a simple API to specify hardware restrictions needed to simulate custom architectures such as Neurogrid [50] and FACETS [51]. These additional features and a complete analysis of the communication [42] will be discussed in future publications.

VI. CONCLUSION

A novel simulator environment, HRLSim, was described for modeling very large-scale spiking neural models. The design of the simulator code as well as the communication was described. Using small-world network topologies, various performance measures were measured and analyzed using a GPGPU cluster. The performance shows that HRLSim offers a real-time simulation tool to study very large-scale spiking neural models. Furthermore, with GPGPU cards becoming more affordable combined with higher performance for each card, this enables a scalable solution to support the design and analysis of very large-spiking neural models based on HRLSim.

REFERENCES

[1] G. Buzsáki, *Rhythms of the Brain*, 1st ed. Oxford, U.K.: Oxford Univ. Press, Aug. 2006.

[2] B. A. Wilt, L. D. Burns, E. T. Wei Ho, K. K. Ghosh, E. A. Mukamel, and M. J. Schnitzer, "Advances in light microscopy for neuroscience," *Annu. Rev. Neurosci.*, vol. 32, no. 1, pp. 435–506, 2009.

[3] N. Srinivasa and J. Cruz-Albrecht, "Neuromorphic adaptive plastic scalable electronics: Analog learning systems," *IEEE Pulse*, vol. 3, no. 1, pp. 51–56, Jan. 2012.

[4] K. Minkovich, N. Srinivasa, J. M. Cruz-Albrecht, Y. K. Cho, and A. Nogin, "Programming time-multiplexed reconfigurable hardware using a scalable neuromorphic compiler," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 23, no. 6, pp. 889–901, Jun. 2012.

[5] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2004, p. 47.

[6] M. Hines and N. Carnevale, "Translating network models to parallel hardware in NEURON," *J. Neurosci. Methods*, vol. 169, no. 2, pp. 425–455, 2007.

[7] M. L. Hines and N. T. Carnavale, "The neuron simulation environment," *Neural Comput.*, vol. 9, no. 6, pp. 1179–1209, 1997.

[8] J. Bower, D. Beeman, and M. Hucka, "GENESIS simulation system," *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA, USA: MIT Press, 2002, pp. 475–478.

[9] J. Bower and D. Beeman, *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*, 2nd ed. New York, NY, USA: Springer-Verlag, 1998.

[10] M.-O. Gewaltig and M. Diesmann, "NEST (NEural Simulation Tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007.

[11] E. C. Wilson, "Parallel implementation of a large scale biologically realistic neocortical neural network simulator," M.S. thesis, Univ. Nevada, Reno, NV, USA, Aug. 2001.

[12] C. Wilson, P. Goodman, and C. Harris, Jr., "Implementation of a biologically realistic parallel neocortical-neural network simulator," in *Proc. SIAM Conf. Parallel Process. Sci. Comput.*, 2001, pp. 1–11.

[13] D. Pecevski, T. Natschläger, and K. Schuch, "PCSIM: A parallel simulation environment for neural circuits fully integrated with python," *Frontiers Neuroinf.*, vol. 3, no. 11, p. 15, May 2009.

[14] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. Bower, M. Diesmann, A. Morrison, P. Goodman, F. Harris, M. Zirpe, T. Natschlager, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. Davison, S. El Boustani, and A. Destexhe, "Simulation of networks of spiking neurons: A review of tools and strategies," *J. Comput. Neurosci.*, vol. 23, no. 3, pp. 349–398, 2007.

[15] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," *Neural Netw.*, vol. 22, nos. 5–6, pp. 791–800, 2009.

[16] M. Richert, J. M. Nageswaran, N. Dutt, and J. L. Krichmar, "An efficient simulation environment for modeling large-scale cortical processing," *Frontiers Neuroinf.*, vol. 5, p. 19, Sep. 2011.

[17] C. M. Thibeault, R. Hoang, and F. C. Harris, Jr., "A novel multi-GPU neural simulator," in *Proc. 3rd BICoB*, Mar. 2011.

[18] E. Izhikevich, "Simple model of spiking neurons," *IEEE Trans. Neural Netw.*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003.

[19] R. Scorcioni, "GPGPU implementation of a synaptically optimized, anatomically accurate spiking network simulator," in *Proc. BSEC*, May 2010, pp. 1–3.

[20] J.-P. Tiesel and A. S. Maida, "Using parallel GPU architecture for simulation of planar I/F networks," in *Proc. INNS*, 2009, pp. 3118–3123.

[21] B. Han and T. M. Taha, "Acceleration of spiking neural network based pattern recognition on NVIDIA graphics processors," *Appl. Opt.*, vol. 49, no. 10, pp. B83–B91, 2010.

[22] J. Igarashi, O. Shouno, T. Fukai, and H. Tsujino, "Real-time simulation of a spiking neural network model of the basal ganglia circuitry using general purpose computing on graphics processing units," *Neural Netw.*, vol. 24, no. 9, pp. 950–960, 2011.

[23] P. Richmond, B. Lars, G. Michele, and V. E. Richmond, "Democratic population decisions result in robust policy-gradient learning: A parametric study with GPU simulations," *PLoS ONE*, vol. 6, no. 5, p. e18539, May 2011.

[24] A. Fidjeland and M. Shanahan, "Accelerated simulation of spiking neural networks using GPUs," in *Proc. IJCNN*, Jul. 2010, pp. 1–8.

[25] D. Yudanov, M. Shaaban, R. Melton, and L. Reznik, "GPU-based simulation of spiking neural networks with real-time performance amp; high accuracy," in *Proc. IJCNN*, Jul. 2010, pp. 1–8.

[26] A. Nere, A. Hashmi, and M. Lipasti, "Profiling heterogeneous multi-GPU systems to accelerate cortically inspired learning algorithms," in *Proc. IEEE IPDPS*, May 2011, pp. 906–920.

[27] R. Y. de Camargo, L. Rozante, and S. W. Song, "A multi-GPU algorithm for large-scale neuronal networks," *Concurrency Comput., Pract. Exper.*, vol. 23, no. 6, pp. 556–572, 2011.

[28] N. Burkitt, "A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input," *Biol. Cybern.*, vol. 95, pp. 1–19, Jun. 2006.

[29] E. M. Izhikevich, *Dynamical Systems in Neuroscience*. Cambridge, MA, USA: MIT Press, 2007.

[30] E. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE Trans. Neural Netw.*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004.

[31] S. Song, K. D. Miller, and L. F. Abbott, "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity," *Nature Neurosci.*, vol. 3, no. 9, pp. 919–926, 2000.

[32] A. P. Davison, D. Brüderle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "PyNN: A common interface for neuronal network simulators," *Frontiers Neuroinf.*, vol. 2, pp. 1–11, Jan. 2009.

[33] J. Hickey and A. Nogin, "Omake: Designing a scalable build process," in *Fundamental Approaches to Software Engineering* (Lecture Notes in Computer Science), vol. 3922, L. Baresi and R. Heckel, Eds. Berlin, Germany: Springer-Verlag, 2006, pp. 63–78.

[34] A. Pope and P. Langley, "CASTLE: A framework for integrating cognitive models into virtual environments," AAAI, Palo Alto, CA, USA, Tech. Rep. FS-08-04 (152), 2008.

[35] O. Michel, "Webots: Symbiosis between virtual and real mobile robots," in *Virtual Worlds* (Lecture Notes in Computer Science), vol. 1434, J.-C. Heudin, Ed. Berlin, Germany: Springer-Verlag, 1998, pp. 254–263.

[36] (2003). *Python: Boost Documentation* [Online]. Available: http://www.boost.org/libs/python/doc/

[37] A. Morrison, C. Mehring, T. Geisel, A. Aertsen, and M. Diesmann, "Advancing the boundaries of high-connectivity network simulation with distributed computing," *Neural Comput.*, vol. 17, no. 8, pp. 1776–1801, Aug. 2005.

[38] H. Plesser, J. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig, "Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers," in *Euro-Par 2007 Parallel Processing* (Lecture Notes in Computer Science), vol. 4641, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds. Berlin, Germany: Springer-Verlag, 2007, pp. 672–681.

[39] M. Migliore, C. Cannia, W. Lytton, H. Markram, and M. Hines, "Parallel network simulations with NEURON," *J. Comput. Neurosci.*, vol. 21, no. 2, pp. 119–129, 2006.

[40] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address-events," *IEEE Trans. Circuits Syst. II*, vol. 47, no. 5, pp. 416–34, May 2000.

[41] (2010). *MVAPICH: MPI Over InfiniBand, 10GigE/iWARP and RoCE*. Ohio State Univ., Columbus, OH, USA [Online]. Available: http://mvapich.cse.ohio-state.edu/overview/mvapich2/

[42] C. M. Thibeault, K. Minkovich, M. J. O'Brien, F. C. Harris, and N. Srinivasa, "Efficiently passing messages in distributed spiking neural network simulation," *Frontiers Comput. Neurosci.*, vol. 7, p. 77, Jun. 2013.

[43] A. Morrison, S. Straube, H. E. Plesser, and M. Diesmann, "Exact subthreshold integration with continuous spike times in discrete-time neural network simulations." *Neural Comput.*, vol. 19, no. 1, pp. 47–79, 2007.

[44] N. Srinivasa and Y. K. Cho, "A self-organizing spiking neural model for learning fault-tolerant spatio-motor transformations," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 23, no. 10, pp. 1526–1538, Oct. 2012.

[45] M. J. O'Brien and N. Srinivasa, "A spiking neural model for stable reinforcement of synapses based on multiple distal rewards," *Neural Comput.*, vol. 25, no. 1, pp. 123–156, 2013.

[46] A. Morrison, A. Aertsen, and M. Diesmann, "Spike-timing-dependent plasticity in balanced random networks," *Neural Comput.*, vol. 19, no. 6, pp. 1437–1467, 2007.

[47] S. Achard, R. Salvador, B. Whitcher, J. Suckling, and E. Bullmore, "A resilient, low-frequency, small-world human brain functional network with highly connected association cortical hubs." *J. Neurosci.*, vol. 26, no. 1, pp. 63–72, 2006.

[48] S. Kunkel, T. C. Potjans, J. M. Eppler, H. E. Plesser, A. Morrison, and M. Diesmann, "Meeting the memory challenges of brain-scale network simulation," *Frontiers Neuroinf.*, vol. 5, no. 35, pp. 1–15, 2012.

[49] Khronos. (2009). *OpenCL—The Open Standard for Parallel Programming of Heterogeneous Systems*, Beaverton, OH, USA [Online]. Available: http://www.khronous.org/opencl/

[50] P. A. Merolla, J. V. Arthur, B. E. Shi, and K. A. Boahen, "Expandable networks for neuromorphic chips," *IEEE Trans. Circuits Syst.*, vol. 54, no. 2, pp. 301–311, Feb. 2007.

[51] J. Schemmel, A. Grübl, K. Meier, and E. Mueller, "Implementing synaptic plasticity in a VLSI spiking neural network model," in *Proc. Int. Joint Conf. Neural Netw.*, 2006, pp. 1–6.

Kirill Minkovich received the B.A. degree from the University of California, Berkeley, Berkeley, CA, USA, in 2003, and the M.S. and Ph.D. degrees from the University of California, Los Angeles, Los Angeles, CA, USA, in 2006 and 2010, respectively, all in computer science.

He was a Technical Staff Member with HRL Laboratories, Malibu, CA, in 2009 and 2012. He is currently a Member of Consulting Staff at Mentor Graphics. His current research interests include synthesis for nanoscale architectures, distributed algorithms, large-scale simulations, and big data analytics for physical synthesis.

Corey M. Thibeault (S'06–M'13) received the B.S. degree in computer engineering from the Rochester Institute of Technology, Rochester, NY, USA, in 2004, the B.S. and M.S. degrees in mechanical engineering from Minnesota State University, Mankato, MN, USA, in 2006 and 2009, respectively, and the M.S. degree in computer engineering and the Ph.D. degree in biomedical engineering from the University of Nevada, Reno, NV, USA, in 2012.

He is currently a Post-Doctoral Researcher with the Center for Neural and Emergent Systems, HRL Laboratories, Malibu, CA, USA. His current research interests include computational neuroscience and biology, high-performance computing, and intelligent systems.

Michael John O'Brien received the B.A. degree in mathematics, physics, and computer science from Claremont McKenna College, Claremont, CA, USA, in 2005, the M.S. degree in mathematics and the Ph.D. degree in mathematics with an emphasis on modeling self-organized and emergent neural systems from the University of California, Los Angeles, Los Angeles, CA, USA, in 2009 and 2013, respectively.

He has been with the Center for Neural and Emergent Systems, Information and System Sciences Department, HRL Laboratories LLC, Malibu, CA, USA, since 2008.

Aleksey Nogin joined the HRL Laboratories, LLC, Malibu, CA, USA, in 2006, where he is currently a Research Computer Scientist, PM, and PI for a DARPA HACMS Project as well as several internal projects in cyber security. His current research interests include cyber security, formal proof systems, formal methods, software reliability, foundations of programming languages, and complexity theory.

Youngkwan Cho received the B.S. and M.S. degrees in computer science from Seoul National University, Seoul, Korea, and the Ph.D. degree in computer science from the University of Southern California, Los Angeles, CA, USA.

He is a Research Staff with the Information and System Sciences Laboratory, HRL Laboratories, Malibu, CA, USA. He has served on the industry advisory boards of universities. His current research interests include neuroscience, virtual and augmented reality, advanced 3-D object rendering, object modeling, visualization, computer vision, and image processing.

Narayan Srinivasa (M'00–SM'12) received the Ph.D. degree in mechanical engineering from the University of Florida, Gainesville, FL, USA, in 1994.

He is a Principal Research Scientist and Manager for the Center for Neural and Emergent Systems, Information and System Sciences Department, HRL Laboratories LLC, Malibu, CA, USA. He is currently the Program Manager and Principal Investigator for three DARPA projects. The DARPA SyNAPSE and Physical Intelligence Programs attempt to develop a theoretical foundation inspired by brain science and physics to engineer electronic systems that exhibit intelligence. The DARPA UPSIDE Program seeks to develop emerging device technologies for highly energy efficient implementations for real-world applications, including image processing. He has managed several projects for GM and Boeing solving real-world problems in the areas of sensing and control. He was a Beckman Fellow with the Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, IL, USA, from 1994 to 1997. He has published 83 technical papers and holds 32 U.S. patents.

Dr. Srinivasa received the HRL Distinguished Inventor Award, the GM Most Valuable Colleague Award, the HRL Outstanding Team Award, and the HRL Chairman Award. He is a member of INNS and AAAS.