CARLsim 3: A User-Friendly and Highly Optimized Library for the Creation of Neurobiologically Detailed Spiking Neural Networks

Michael Beyeler*, Kristofor D. Carlson*, Ting-Shuo Chou*, Nikil Dutt, Jeffrey L. Krichmar
Department of Cognitive Sciences and Department of Computer Science
University of California, Irvine
Irvine, CA, 92697 USA
{mbeyeler, tingshuc, kdcarlso, dutt, jkrichma}@uci.edu
*Authors contributed equally to this work

Abstract—**Spiking neural network (SNN) models describe key aspects of neural function in a computationally efficient manner and have been used to construct large-scale brain models. Large-scale SNNs are challenging to implement, as they demand high-bandwidth communication, a large amount of memory, and are computationally intensive. Additionally, tuning parameters of these models becomes more difficult and time-consuming with the addition of biologically accurate descriptions. To meet these challenges, we have developed CARLsim 3, a user-friendly, GPU-accelerated SNN library written in C/C++ that is capable of simulating biologically detailed neural models. The present release of CARLsim provides a number of improvements over our prior SNN library to allow the user to easily analyze simulation data, explore synaptic plasticity rules, and automate parameter tuning. In the present paper, we provide examples and performance benchmarks highlighting the library's features.**

Keywords—spiking neural networks; simulation tools; GPU computing; large-scale brain models; neuromorphic engineering

I. INTRODUCTION

Spiking neural network (SNN) models play an important role in understanding brain function [1] and in designing neuromorphic devices with the energy efficiency, computational power, and fault-tolerance of the brain [2]. These models represent a compromise between execution time and biological fidelity by capturing essential aspects of neural communication, such as spike dynamics, synaptic conductance, and plasticity, while foregoing computationally expensive descriptions of spatial voltage propagation found in compartmental models [3]. However, developing efficient simulation environments for SNN models is challenging due to the required memory to store the neuronal/synaptic state variables and the time needed to solve the dynamical equations describing these models. A number of simulators are already available to the public [4], each providing their own unique qualities and trade-offs based on the employed abstraction level and supported computer hardware (see Table I). Given the considerable potential for parallelization of artificial neural networks [5], it is not surprising that most simulators today offer implementations on parallel architectures, such as computer clusters or graphics processing units (GPUs). However, in order to be of practical use to the computational modeling community, an SNN simulator should not only be fast, but also freely available, capable of running on affordable hardware, have a user-friendly interface, include complete documentation, and provide tools to easily design, construct, and analyze SNN models.

In this paper we present CARLsim 3, an efficient, easy-to-use, GPU-accelerated library for simulating large-scale SNN models with a high degree of biological detail. Due to its efficient GPU implementation, CARLsim 3 is useful for designing large-scale SNN models that have real-time constraints (e.g., interacting with neuromorphic sensors or controlling neurorobotics platforms).

Building on the demonstrated efficiency and scalability of earlier releases [6], [7], the present release improves the usability of our software by means of platform compatibility (Linux, Mac OS X, and Windows), rigorous code documentation (including an extensive user guide and tutorials), a test suite for functional code verification, and a MATLAB toolbox for the visualization and analysis of neuronal, synaptic, and network information. In addition, CARLsim 3 provides native support for a range of spike-based synaptic plasticity mechanisms and topographic synaptic projections, as well as being among the first to provide support for a network-level parameter tuning interface.

To promote its use among the computational neuroscience and neuromorphic engineering communities, CARLsim is provided as an open-source C++ package, which can be freely obtained from: www.socsci.uci.edu/~jkrichma/CARLsim.

II. CARLSIM

CARLsim is a C/C++ based SNN simulator that allows execution of networks of Izhikevich spiking neurons [8] with realistic synaptic dynamics on both generic x86 CPUs and standard off-the-shelf GPUs. The simulator provides a PyNN-like programming interface [9], which allows for details to be specified at the synapse, neuron, and network level.

This work was supported by National Science Foundation (NSF) Award No. IIS-1302125 and an award from Qualcomm Technologies Incorporated.

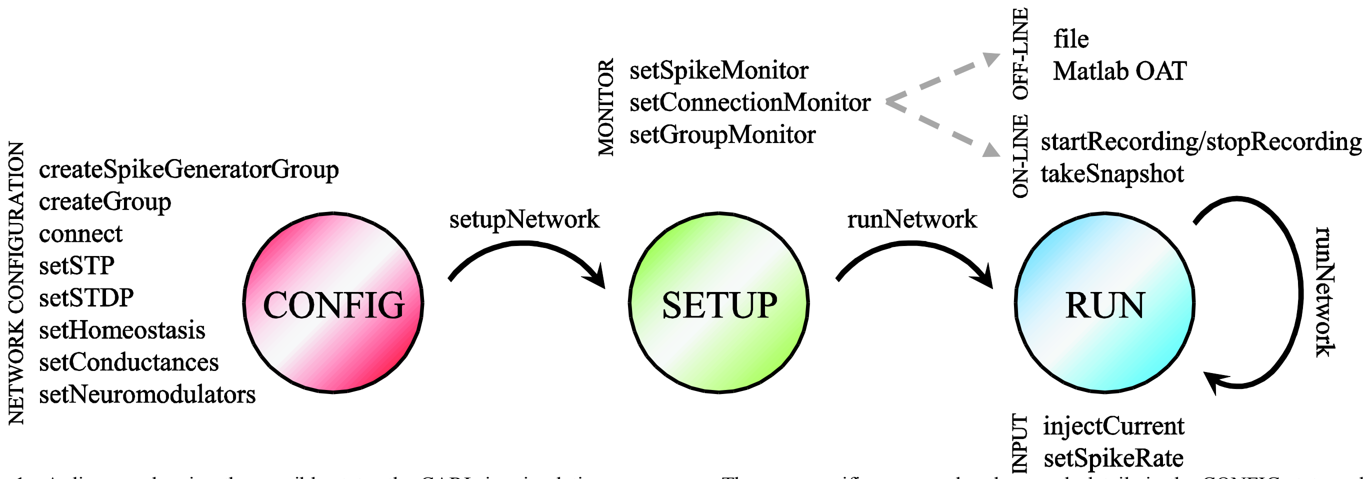

Fig. 1. A diagram showing the possible states the CARLsim simulation can occupy. The user specifies neuronal and network details in the CONFIG state and moves to the SETUP state, where data monitors can be set, by running `setupNetwork`. Next, the user calls `runNetwork` to move to the RUN state where the input stimuli are set and the simulation is executed. The user can optionally start and stop data monitors during the simulation or use the offline analysis toolbox (OAT) with MATLAB to analyze data written to files.

The simulator was first introduced in 2009 (now referred to as CARLsim 1), where it demonstrated near real-time performance for 100,000 spiking neurons on a single NVIDIA GTX 280 GPU [6]. CARLsim 2 added basic support for synaptic conductances, spike-timing dependent plasticity (STDP) and short-term plasticity (STP) [7]. The present release, CARLsim 3, greatly expands the functionality of the simulator by adding a number of features that enable and simplify the creation, tuning, and simulation of complex networks with spatial structure. These features include: 1) real-time and offline data analysis tools, 2) a more complete STDP implementation that includes dopaminergic neuromodulation, and 3) an automated parameter tuning interface. In addition, several software engineering techniques and more complete documentation are introduced in the present release to ensure the integrity of the current code base and to lay the groundwork for the success of future releases. The following subsections will explain these achievements in detail.

A. *CARLsim API: General Workflow*

The workflow of a typical CARLsim 3 simulation is organized into three distinct, consecutive states (see Fig. 1): the configuration state (`CONFIG`), the set up state (`SETUP`), and the run state (`RUN`). User functions in the C++ API are grouped according to these stages, which streamline the process and prevent race conditions. State transitions are handled by special user functions such as `setupNetwork` and `runNetwork`.

The first step in using CARLsim 3 (`libCARLsim`) imports the library and instantiates the main simulation object:

```
#include <carlsim.h>
CARLsim sim("example", GPU_MODE, SILENT);
```

This prepares the simulation for execution in either `CPU_MODE` or `GPU_MODE`, and specifies the verbosity of the status reporting mechanism (`SILENT` indicating that no console output will be produced). From then on, the simulation is in `CONFIG` state, allowing the properties of the neural network to be specified.

Similar to PyNN [9] and many other simulation environments, CARLsim uses groups of neurons and connections as an abstraction to aid defining synaptic connectivity. Different groups of neurons can be created from a one-dimensional array to a three-dimensional grid via `createSpikeGeneratorGroup` or `createGroup`, and connections can be specified depending on the relative placement of neurons via `connect`. This allows for the creation of networks with complex spatial structure. For a selective list of available function calls in `CONFIG` state please refer to the left-hand side of Fig. 1.

Izhikevich spiking neurons [8] with either current-based or conductance-based synapses are currently supported, but more neuron types are planned for the future. The following code snippet creates a group of 500 excitatory neurons arranged on a 10x10x5 three-dimensional grid:

```
int gOut = sim.createGroup("output", Grid3D(10,10,5),
    EXCITATORY_NEURON);
```

Here, `"output"` is an arbitrary name for the group while `EXCITATORY_NEURON` denotes that all neurons in the group have glutamatergic synapses. Neurons with GABAergic synapses are supported with the `INHIBITORY_NEURON` keyword. Neurons with dopaminergic synapses are supported with the `DOPAMINGERIC_NEURON` keyword and can be used to modulate STDP learning curves (see below). To refer to this group in later method calls, the `createGroup` method returns a group ID, `gOut`. Next, the Izhikevich parameters are specified, in this case for class 1 excitability regular spiking neurons:

```
sim.setNeuronParameters(gOut,0.02f,0.2f,-65.0f,8.0f);
```

where 0.02f, 0.2f, -65.0f, and 8.0f correspond respectively to the *a*, *b*, *c*, and *d* parameters of the Izhikevich neuron.

To create a group of neurons that generate Poissonian spike trains, the user specifies a name, size, and type. These neurons are used to input activity into the network and can be driven by sensory stimuli.

```
int gIn = sim.createSpikeGeneratorGroup("input",
    Grid3D(10,10,5), EXCITATORY_NEURON);
```

The following statement then creates a random connection pattern from group `gIn` to group `gOut` with weight 0.1f, 10% connection probability, a synaptic delay uniformly distributed between 1 ms and 20 ms, a specific receptive field size, and fixed synapses (SYN_FIXED):

```
sim.connect(gIn, gOut, ”random”, RangeWeight(0.1f),
    0.1f, RangeDelay(1,20), RadiusRF(3,3,3), SYN_FIXED);
```

Here, the RangeWeight struct usually takes three parameters specifying the minimum, initial, and maximum weights, or just one parameter if all values are identical (as is the case for fixed synapses). The (optional) RadiusRF struct specifies the x, y, and z dimensions of a radial receptive field, following the topographic organization of the Grid3D struct. For example, the above snippet creates a spherical receptive field that only connects neurons in gIn to neurons in gOut if the distance between their locations on the three-dimensional grid is less than 3 (arbitrary units). Plastic connections are created simply by replacing the keyword SYN_FIXED with SYN_PLASTIC. If one is not satisfied with the built-in connection types ("one-to-one", "full", "random", and "gaussian"), a callback mechanism is available for user-specified connectivity.

The present release allows users to choose from a number of synaptic plasticity mechanisms. These include standard equations for STP [10], [11], various forms of additive nearest-neighbor STDP [12], and homeostatic plasticity in the form of synaptic scaling [13]. STDP is a paradigm that modulates the weight of synapses according to their degree of causality. Many different variants of STDP seem to exist in the brain, the functional roles of which are still largely unknown. CARLsim now provides an efficient implementation of a number of experimentally validated STDP curves (see Fig. 2), which may greatly enhance the study of STDP's functional role in computational models of brain function. STDP can either apply to glutamatergic synapses (E-STDP; see Fig. 2(a), (b)) or GABAergic synapses (I-STDP; see Fig. 2(a), (c)), and is specified post-synaptically.

In addition, CARLsim 3 supports dopamine-modulated STDP (DA-STDP) [14]. In the following code example, 100 dopaminergic neurons (gDA) project to a group of ten regular spiking neurons (gRS) with exponential DA-STDP:

```
// create groups
int gDA = sim.createSpikeGeneratorGroup("DA input", 100,
    DOPAMINERGIC_NEURON);
int gRS = sim.createGroup(“RS”, 10, EXCITATORY_NEURON);
sim.setNeuronParameters(gRS, 0.02f, 0.2f, -65.0f, 8.0f);

// “all-to-all” connectivity with plastic synapses
// weights initialized to 0.01f, range is [0.0f, 0.1f]
// receptive field struct is ignored
sim.connect(gDA, gRS, “full”,
    RangeWeight(0.0f, 0.01f, 0.1f), 1.0f,
    RangeDelay(1,10), RadiusRF(-1), SYN_PLASTIC);

// set DA-STDP on all plastic synapses to gRS
float alpha_plus  =  0.001f,  tau_plus  = 20.0f;
float alpha_minus = -0.0015f, tau_minus = 20.0f;
sim.setESTDP(gRS, DA_MOD, ExpCurve(alpha_plus, tau_plus,
    alpha_minus, tau_minus));
```

Once the spiking network has been specified, the function setupNetwork optimizes the network state for the chosen back-end (CPU or GPU) and moves the simulation into SETUP state (see Fig. 1). In this state, a number of monitors can be set to record variables of interest (e.g., spikes, weights, state variables) in binary files for off-line analysis. New in CARLsim 3 is a means to make these data available at run-time (without the computational overhead of writing data to disk), which can be queried for data in the RUN state.

The first call to runNetwork will take the simulation into RUN state. The simulation can be repeatedly run (or "stepped") for an arbitrary number of sec*1000+msec milliseconds:

```
sim.runNetwork(sec, msec);
```

Input can be generated via current injection or spike injection (the latter using Poisson spike generators, or a callback mechanism to specify arbitrary spike trains). We also provide plug-in code to generate Poisson spike trains from animated visual stimuli such as sinusoidal gratings, plaids, and random dot fields created via the VisualStimulus toolbox.

Monitors allow users to selectively record variables of interest during a subset of the simulation, and provide a number of metrics useful for real-time analysis:

```
// create network
int g0=sim.createGroup(“group0”, ... // etc.
sim.setupNetwork();

// set spike monitor for g0, record to “default” file
// “./results/spk_group0.dat”
SpikeMonitor* SM = sim.setSpikeMonitor(g0, ”default”);

// in addition, record data to object for 1 sec
SM.startRecording();
sim.runNetwork(1,0);
SM.stopRecording();

// after calling stopRecording, SpikeMonitor object can
// be queried for spike stats
// e.g.: get the mean firing rate of neurons in group g0
float rate = SM.getPopMeanFiringRate();

// run some more w/o recording to SpikeMonitor object
sim.runNetwork(10,0); // etc.
```

Here, spike events (spike times and neuron IDs) from “group0” with ID g0 are recorded exclusively during the first call to runNetwork. Hence metrics (e.g. the average firing rate for “group0”) returned by the SpikeMonitor object will typically refer to the last recording period, although it is possible to concatenate independent recording periods using PersistentMode. Analogously, one could set up a ConnectionMonitor to analyze the average weight change during a training session to infer when learning has saturated.

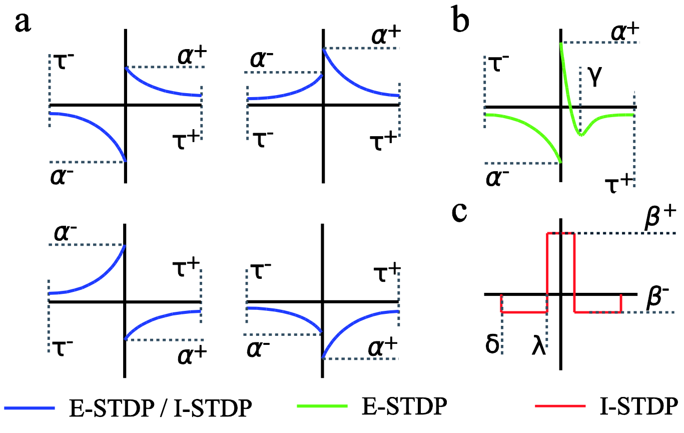

Fig. 2: Possible STDP curves in CARLsim (weight change for a pair of pre-post spikes; pre-before-post: +, pre-after-post: –). Green curves can be applied to glutamatergic synapses, red curves can be applied to GABAergic synapses, and blue curves can be applied to both. **(a)** Exponential curve. **(b)** Timing-based curve. **(c)** Pulsed curve.

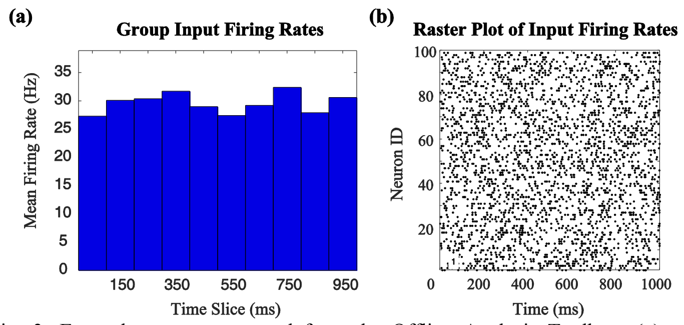

Fig. 3: Example output generated from the Offline Analysis Toolbox. **(a)** Visualization of the mean firing rate of a single input group of 100 neurons with an average firing rate of 30 Hz. **(b)** A raster plot of the same spiking data shown in **(a)**.

In addition to the real-time monitors, CARLsim 3 provides a versatile Offline Analysis Toolbox (OAT) written in MATLAB. Assuming the corresponding binary files exist, visualizing network activity is achieved by simply passing the name of the neuron group to a `GroupMonitor` in MATLAB:

```
GM = GroupMonitor('input'); % group is named 'input'
GM.plot('histogram'); % plots the data as a histogram
GM.plot('rasterplot'); % plots the data as a raster plot
```

The code snippet above generated the plots shown in Fig. 3. We first select the group named `input`, change the plot type to `histogram`, and plot the firing rate data shown in Fig. 3(a). We then change the plot type to `rasterplot` to generate Fig. 3(b). This allows users to quickly visualize and easily analyze their network simulations.

B. *CARLsim Kernel: Simulation Engine*

An important feature of CARLsim is the ability to run spiking networks not only on CPUs, but also on off-the-shelf NVIDIA GPUs. The GPU implementation of CARLsim was written to optimize four main performance metrics: parallelism, memory bandwidth, memory usage, and thread divergence [6]. The simulation is broken into steps that update neuronal state variables in parallel (N-parallelism) and steps that update synaptic state variables in parallel (S-parallelism). Sparse representation techniques for spiking events and neuronal firing decrease both memory and memory bandwidth usage. Buffering data helps decrease thread divergence and enables efficient run-time access of recorded variables (e.g., via `SpikeMonitor`) that are stored on the GPU.

The CARLsim 3 kernel introduces a number of software engineering techniques to assure the integrity of the current code base and to enable successful growth of the project in the future. For example, the pointer to implementation (pImpl) idiom is used to programmatically separate user interface from implementation, which will simplify the addition of different front-ends and back-ends in the future. Regression testing, via Google Test, is used to ensure that the development of new functionality did not compromise the existing code base, and new features are validated using functional and unit testing. The full test suite is visible to the user for automatic quality assurance after the installation.

C. *Parameter Tuning with Evolutionary Algorithms*

CARLsim 3 introduces a software interface to an evolutionary computation system written in Java (ECJ) [15] to provide an automated parameter tuning framework (Linux). As more complex biological features are integrated into SNN models, it becomes increasingly important to provide users with a method of tuning the large number of open parameters. Evolutionary Algorithms (EAs) enable flexible parameter tuning by means of optimizing a generic fitness function. The effectiveness of this approach was previously demonstrated by tuning an SNN to produce V1-like tuning curve responses with self-organizing receptive fields using an EA library called Evolving Objects (EO) [16]. ECJ was chosen to replace EO because it is under active development, supports multi-threading, has excellent documentation, and implements a variety of EAs [15].

Fig. 4 shows the general approach of the automated parameter tuning framework. ECJ implements an EA with a parameter file that includes: EA parameters, the number of individuals per generation, and parameter ranges. Every step of the EA is executed by ECJ except for the evaluation of the fitness function, which is completed by CARLsim. CARLsim evaluates the fitness function in parallel by running multiple SNN individuals concurrently on the GPU, where the bulk of the computations occur. ECJ is written in Java, which is slower than C, but the majority of the execution time is spent running CARLsim's optimized C++/CUDA code. At the beginning of every generation, the parameters to be tuned are passed from ECJ to CARLsim using standard input/output streams. CARLsim evaluates individuals in parallel and returns the resulting fitness values to ECJ via standard streams. The tuning framework allows users to tune virtually any SNN parameter, while the fitness functions can be written to depend on the neuronal activity or synaptic weights of the SNN. This allows users to explore a variety of plasticity rules, firing activities, and functionally relevant SNN behaviors.

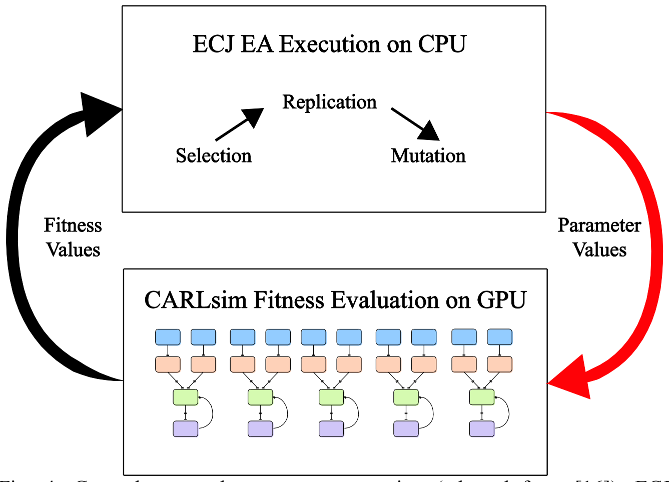

Fig. 4: General approach to parameter tuning (adapted from [16]). ECJ performs the Evolutionary Algorithm (EA) and passes the current generation of parameters (red arrow) to CARLsim for evaluation using the parameter tuning interface (PTI) code. CARLsim assigns each parameter set to an SNN and evaluates all the individuals in parallel, passing the fitness values back to ECJ for selection of individuals in the next generation (black arrow).

III. Results

A. *Existing Models Using CARLsim*

Previous CARLsim versions have been used to implement a broad range of computational models on the order 10k–100k neurons and millions of synapses, with examples that include models of visual processing [7], [17], neuromodulation [18], synaptic plasticity [13], and attention [19], [20]. In these SNN implementations, CARLsim allowed modelers to quickly and efficiently simulate large-scale networks to examine the role biophysical mechanisms play in behavioral tasks. In one such study, a digit categorization SNN used low-level memory encoding mechanisms to classify handwritten digits with 92% accuracy in real-time while quantitatively and qualitatively reproducing psychophysical experimental data [21]. In another study, a novel homeostatic synaptic plasticity model was shown to stabilize STDP and play an important role in the production of self-organizing receptive fields [13].

B. *Example of New CARLsim 3 Functionality*

CARLsim 3 has increased functionality to support the added complexity of future SNN models. For example, Connection Monitors can be used to easily determine when learning has saturated and a training session should be ended, whereas real-time Spike Monitors allow for the integration of CARLsim with robotics and neuromorphic platforms. To showcase new CARLsim 3 functionality, we next reproduce results from a study on instrumental conditioning using DA-STDP [14].

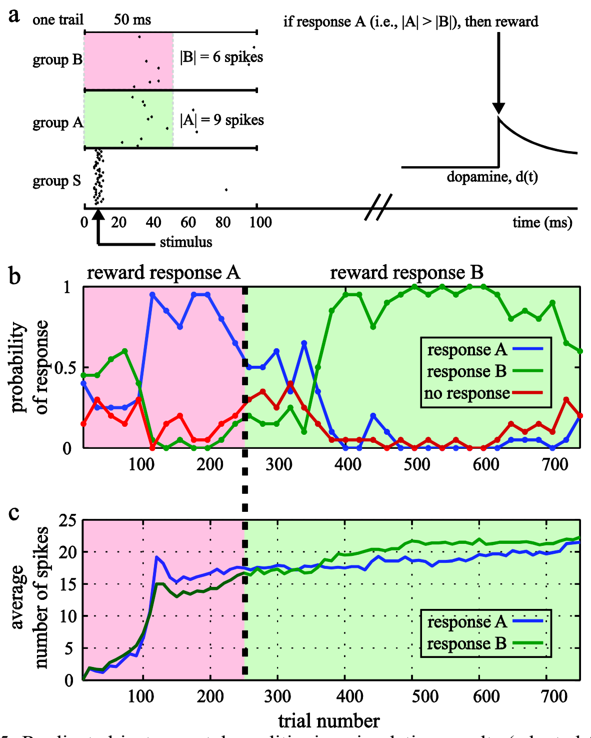

Fig. 5: Replicated instrumental conditioning simulation results (adapted from [14]). **(a)** Experimental setup. **(b)** Plot of probability of network response versus trial number. Pink shaded areas denote reward applied to response *A* while green shaded areas denote reward applied to response *B*. **(c)** Plot of average number of spikes from groups *A* and *B* versus trial number.

We reproduced findings from a computational study that used spike firing patterns and a global diffusive reinforcement signal (dopamine) to learn associations between cues and rewards [14]. Our simulations used DA-STDP with the experimental setup shown in Fig. 5(a). For each trial, a stimulus was delivered to group *S* resulting in the firing of a majority of the neurons in the group. The firing activity from group *S* evoked a small number of spikes in groups *A* and *B*, which represented the response of the network to the stimulus. As in [14], the network response state consisted of 3 possible states: *A*, *B*, or no response. If group *A* had more spikes than group *B*, the network response was said to be in state *A*, with the opposite being true for state *B*, and the no response state occurring when both groups *A* and *B* had the same number of spikes.

An increase in the dopamine concentration in the network represented reward in the simulation. In the first 250 trials, the network was rewarded only if it produced response *A*. After 250 trials, the reward condition changed and the network was rewarded only if it produced response *B*. Fig. 5(b) and (c) show simulation results that replicate previous work [14]. During the initial stage (i.e., trials 1 ~ 100), the firing rates in both groups *A* and *B* increased but the network response was random. After 100 trials, the network had response *A* with significant higher probability. At trial 250 the reward condition was changed to response *B*, triggering the network to change its response to *B* after trial 360. An interesting phenomenon is that the network spent time (trial 250 ~ 360) re-associating reward with a different response, which was not obvious in previous work [14]. With the DA-STDP feature in CARLsim 3, users can easily implement reinforcement learning applications. The reproduced instrumental conditioning example consumed less than 300 lines of C++ code. For code examples using DA-STDP please see Subsection II-A.

C. *Computational Peformance*

In order to demonstrate the efficiency and scalability of CARLsim 3, we ran simulations consisting of various sized SNNs and measured their execution time. GPU simulations were run on a single NVIDIA GTX 780 (3 GB of memory) using CUDA, and CPU simulations were run on an Intel Core i7 CPU 920 operating at 2.67 GHz.

The results of these benchmarks are summarized in Fig. 6. Networks consisted of 80% excitatory and 20% inhibitory neurons, with an additional 10% of neurons being Poisson spike generators that drove network activity. The two neuronal populations were randomly connected with a fixed connection probability, and E-STDP was used to generate sustained irregular activity as described by Vogels and Abbott [22]. The number of synapses per neuron ranged from 100 to 300 connections, and the overall network activity was ~10 Hz.

GPU simulation speed, given as the ratio of GPU execution time over real-time, is plotted in Fig. 6(a). Execution time scaled linear with workload, both in terms of number of neurons as well as number of synapses. In an earlier paper [7] we reported that networks with 110,000 neurons and 100 synaptic connections per neuron took roughly two seconds of clock time for every second of simulation time (CARLsim 2). The current release (CARLsim 3) was slightly slower in this scenario, taking roughly 2.5 seconds of clock time for every second of simulation time, which is not surprising given the large number of additional features in the present release. On the other hand, GPU mode still significantly outperformed CPU mode (see Fig. 6(b)). GPU execution time was up to 60 times faster than the CPU. This result is due to a combination of newer hardware (GTX 780) as well as code-level optimization that allowed the effective use of S-parallelism [6] for new features such as synaptic receptor ratios and different shaped STDP curves, which are not possible in a single-threaded CPU simulation. In summary, despite the additional features and new programming interface, CARLsim 3 is similar in performance to our previous releases and highly optimized for SNNs on the order of 10^5 neurons and 10^7 synapses.

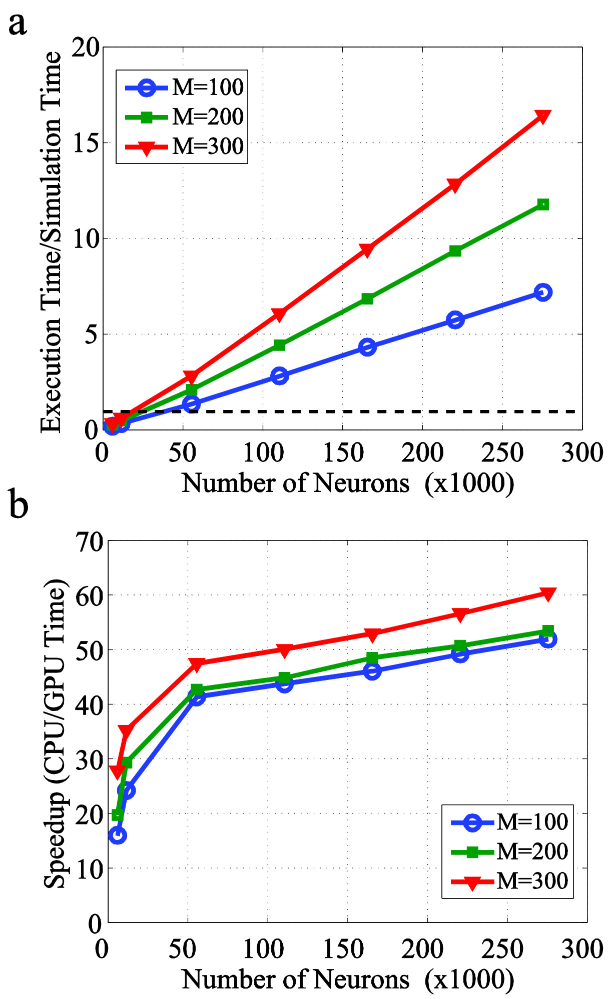

Fig. 6 **(a)** Ratio of execution time to simulation time versus number of neurons for simulations with 100, 200, and 300 synapses/neuron. The dotted line represents simulations running in real-time. **(b)** Simulation speedup versus number of neurons. Speedups increase with neuron number. Models with 50k neurons or more will have the most dramatic speedup.

IV. Related Work

Today, a wide variety of simulation environments are available to the computational neuroscience community that allow for the simulation of SNNs. Although many simulators share similar approaches and features, most have unique qualities that distinguish them from the others. In an effort to identify and highlight these qualities, we compared a list of features provided by a number of other SNN simulators with CARLsim 3. We limited this comparison to large-scale SNN simulators that are most common to CARLsim in that they: (i) are open source, (ii) support the clock-driven and parallelized simulation of point neurons, such as LIF, Izhikevich or aIF neurons, (iii) have conductance-based synapses, and (iv) provide some form of synaptic plasticity. Specifically, we chose (in alphabetical order) Brian 2 [23], GeNN 2 [24], NCS 6 [25], NeMo 0.7 [26], Nengo 2 [27], NEST 2.6 [28] and PCSIM 0.5 [29].

Table I compares different simulation environments with respect to features of both biological realism and technical capability. A specific feature is indicated as either fully implemented (‘ X ’) or absent (blank, ‘ ’). A ‘ / ’ denotes a feature that is only partially implemented (e.g., neuromodulation in NEST), requires substantial user efforts to implement (e.g., DA-STDP in Brian), or is reportedly untested (e.g., Windows support for PCSIM). Although we tried to be as objective as possible in assigning labels, categorization remains subjective in nature. In addition, some of these features were not well-documented; but we were still able to verify their existence by reading through the actual source code and reaching out to the authors for clarification. Overall, we believe that the table accurately and fairly reflects the development status for each of the listed SNN simulators at the time of this publication.

All of the simulators listed in Table I offer many features that allow the simulation of complex neural networks on several back-ends. In addition, if users are interested in modeling certain details of the biological model that are not natively supported, all of the simulators offer ways to extend the code base, either by ways of plug-in code, implementation inheritance, or dynamic code generation. For example, both Brian and GeNN offer ways for the user to formulate any neuronal, synaptic, or learning model they please. This fact makes the simulators invaluable for power-users, but may be difficult for less-experienced programmers and may prohibit code-level optimizations for certain user-defined functionality. Nengo provides flexibility through its scripting interface, and also provides a graphical user interface to construct SNNs at different levels of abstraction. In Nengo, large-scale functional networks can be achieved using the Neural Engineering Framework (NEF), which is a theoretical framework that can use anatomical constraints, functional objectives, and control theory to find the set of weights that approximate some desired functionality [30].

TABLE I. FEATURE COMPARISON FOR SOME COMMON OPEN-SOURCE SNN SIMULATORS (NON-EXHAUSTIVE LIST)

	Neuron model			**Synapse model**				**Synaptic plasticity**					**Input**		**Tools**			**Integration methods**			**Front-ends**			**Back-ends**					**Platforms**		
	Leaky integrate-and-fire (LIF)	Izhikevich 4-param	Hodgkin-Huxley	Current-bsaed (CUBA)	Conductance-based (COBA)	AMPA, NMDA, GABA	Neuromodulation	Short-term plasticity (STP)	E-STDP	I-STDP[a]	DA-STDP	Synaptic scaling / homeostasis	Current injection	Spike injection	Parameter tuning	Analysis and visualization	Regression suite	Forward / exponential Euler	Exact integration[b]	Runge-Kutta	Python / PyNN	C / C++	Java	Single-threaded	Multi-threaded	distributed	Single GPU	Multi-GPU	Linux	Mac OS X	Windows
CARLsim 3		X		X	X	X	/	X	X	X	X	X	X	X	X	X	X	X				X		X			X	/	X	X	X
CARLsim 2		X		X	X	X		X	X					X		/		X				X		X			X		X		/
Brian 2	X	X	X	X	X	X	/	X	X	X	/	X	X	X	/	/	X	X	X	X	X	X		X	X	/			X	X	X
GeNN 2		X	X	X	X	X	/	/	X			/	X	/				X				X		X			X		X	X	X
NCS 6	X	X	X	X	X	/		X	X				X	X		/		X			X	X		X	X	X	X	X	X		
NeMo 0.7		X			X		/	X	X	X			X	X		X	X	X			X	X		X	X		X		X	X	X
Nengo 2	X	X	X	X	X	X	X	X	/			X	X		X	X	X			X	X		X	X	X		X		X	X	X
NEST 2.6	X	X	X	X	X	X	/	X	X		X	X	X	X			X	X	X	X	X			X	X	X			X	X	
PCSIM 0.5	X	X	X	X	X	X	/	X	X		X	X	X	X		/	X	X			X	/	X	X	X	X			X	X	/

a. such as anti-Hebbian or constant symmetric STDP on GABAergic synapses

b. as described in [31]

Given the massive potential for parallelization of artificial neural networks [5], it is not surprising that all of the presented simulators offer implementations on at least one parallel architecture. In order to efficiently run large-scale SNNs, simulators such as NCS, NEST, and PCSIM use distributed computing across standard computer clusters, whereas simulators such as CARLsim, GeNN, NCS, and NeMo leverage the parallel processing capability of NVIDIA GPUs. Currently, NCS 6 seems to be the only open-source SNN simulator to support execution on heterogeneous clusters of CPUs and GPUs. CARLsim 3 currently has partial multi-GPU support as the tuning framework can utilize multiple GPUs. However, full multi-GPU support is reportedly under development for a number of platforms, including a near-future release of CARLsim.

Few simulators have provided a means to automatically tune open parameters of large-scale SNNs. Brian has support for tuning parameters of individual neurons, which has been used to match individual neuron models to electrophysiological data [32]. However, this framework does not easily extend to networks of neurons. Nengo uses the NEF [30] to find synaptic weights between two neuronal populations that approximate a desired non-linear function. The potential of this approach has been demonstrated in Spaun, a 2.5 million neuron simulation that performed eight diverse cognitive tasks [1]. However, in order for the NEF to be effective, the modeler has to know the desired functionality of the neuronal population a priori (i.e., the mathematical function to be approximated). On the other hand, the parameter tuning framework supported by CARLsim 3 does not require this information to be known beforehand, but rather assigns a fitness value to an SNN based on parameters that could relate to anything from synaptic weights over plasticity rules to connection topologies or any other number of biologically or behaviorally relevant parameters. The problem of finding parameter values that maximize fitness is then formulated as an optimization problem, which can be solved with a suitable search method, such as the evolutionary algorithm provided by CARLsim 3's ECJ plug-in. In the future, this framework in combination with CARLsim's GPU implementation could significantly reduce the time researchers spend constructing and tuning large-scale SNNs.

Although all of the simulation environments listed in Table I have their own pros and cons, we believe that CARLsim 3 has advantages when it comes to efficiently simulating large-scale SNNs without having to sacrifice biological realism. In particular, we have made serious efforts to improve the usability of our platform by means of platform compatibility (Linux, Mac OS X, and Windows), rigorous code documentation (including an extensive user guide and tutorials), a regression suite for functional code verification, and a MATLAB toolbox for the visualization and analysis of neuronal, synaptic, and network information. CARLsim 3 provides native support for a range of spike-based synaptic plasticity mechanisms and topographic synaptic projections, as well as being among the first to provide support for a network-level parameter tuning interface. Additionally, the PyNN-like interface, flexible visualization tools, and much improved documentation make CARLsim 3 easy to use.

V. Conclusion

CARLsim 3 is an open-source, C/C++ based SNN simulator that allows the execution of networks of Izhikevich spiking neurons with realistic synaptic dynamics on both generic x86 CPUs and standard off-the-shelf GPUs. The simulation library has minimal external dependencies and provides users with a PyNN-like programming interface. Additionally, CARLsim 3 provides online and offline data analysis tools as well as support for an automated parameter tuning framework. The library, documentation, tutorials and examples can be obtained from: www.socsci.uci.edu/~jkrichma/CARLsim.

CARLsim 3 is well-suited to run SNN models that require a high degree of biological detail without sacrificing performance, which might be potentially useful in real-time systems that combine large-scale SNN models with neuromorphic sensors and neurorobotics platforms. The simulation library can output user-selected neuronal group firing rates on the millisecond time-scale and can thus interact with real-time neuromorphic hardware devices and robotics platforms. In the near future, we plan to add support for a number of different neuron models, integration methods, and back-ends, including the ability to run simulations on multiple GPUs.

References

[1] C. Eliasmith, et al., "A Large-Scale Model of the Functioning Brain," *Science*, vol. 338, no. 6111, pp. 1202–1205, Nov. 2012.

[2] K. Minkovich, C. M. Thibeault, M. J. O'Brien, A. Nogin, Y. Cho, and N. Srinivasa, "HRLSim: A High Performance Spiking Neural Network Simulator for GPGPU Clusters," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. Early Access Online, 2013.

[3] C. Koch, *Biophysics of computation: information processing in single neurons*. Oxford university press, 2004.

[4] R. Brette, et al., "Simulation of networks of spiking neurons: a review of tools and strategies," *J. Comput. Neurosci.*, vol. 23, no. 3, pp. 349–398, Dec. 2007.

[5] T. Nordström and B. Svensson, "Using and designing massively parallel computers for artificial neural networks," *J. Parallel Distrib. Comput.*, vol. 14, no. 3, pp. 260–285, Mar. 1992.

[6] 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. Off. J. Int. Neural Netw. Soc.*, vol. 22, no. 5–6, pp. 791–800, Aug. 2009.

[7] M. Richert, J. M. Nageswaran, N. Dutt, and J. L. Krichmar, "An efficient simulation environment for modeling large-scale cortical processing," *Front. Neuroinformatics*, vol. 5, no. 19, 2011.

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

[9] A. P. Davison, et al., "PyNN: A Common Interface for Neuronal Network Simulators," *Front. Neuroinformatics*, vol. 2, Jan. 2009.

[10] M. Tsodyks, K. Pawelzik, and H. Markram, "Neural Networks with Dynamic Synapses," *Neural Comput.*, vol. 10, no. 4, pp. 821–835, May 1998.

[11] Senn, W., Markram, H., and Tsodyks, M., "An algorithm for modifying neurotransmitter release probability based on pre- and post-synaptic spike timing", *Neural Comput.*, vol. 13, no. 1, pp. 35-67, Jan. 2001.

[12] E. M. Izhikevich and N. S. Desai, "Relating STDP to BCM," *Neural Comput.*, vol. 15, no. 7, pp. 1511–1523, Jul. 2003.

[13] K. D. Carlson, M. Richert, N. Dutt, and J. L. Krichmar, "Biologically Plausible Models of Homeostasis and STDP: Stability and Learning in Spiking Neural Networks," in *Proceedings of the 2013 International Joint Conference on Neural Networks (IJCNN)*, Dallas, Texas, USA, 2013, pp. 1 – 8.

[14] E. M. Izhikevich, "Solving the distal reward problem through linkage of STDP and dopamine signaling," *Cereb. Cortex N. Y. N 1991*, vol. 17, no. 10, pp. 2443–2452, Oct. 2007.

[15] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley, and A. Chircop, "ECJ: A java-based evolutionary computation research system," *http://cs. gmu. edu/eclab/projects/ecj*, 2006.

[16] K. D. Carlson, J. M. Nageswaran, N. Dutt, and J. L. Krichmar, "An efficient automated parameter tuning framework for spiking neural networks," *Front. Neurosci.*, vol. 8, no. 10, 2014.

[17] M. Beyeler, M. Richert, N. D. Dutt, and J. L. Krichmar, "Efficient Spiking Neural Network Model of Pattern Motion Selectivity in Visual Cortex," *Neuroinformatics*, vol. 12, no. 3, pp. 435–454, Jul. 2014.

[18] M. C. Avery, N. Dutt, and J. L. Krichmar, "A large-scale neural network model of the influence of neuromodulatory levels on working memory and behavior," *Front. Comput. Neurosci.*, vol. 7, p. 133, 2013.

[19] M. Avery, J. L. Krichmar, and N. Dutt, "Spiking neuron model of basal forebrain enhancement of visual attention," in *Proccedings of the 2012 International Joint Conference on Neural Networks (IJCNN)*, Brisbane, Australia, 2012, pp. 1–8.

[20] M. C. Avery, N. Dutt, and J. L. Krichmar, "Mechanisms underlying the basal forebrain enhancement of top-down and bottom-up attention," *Eur. J. Neurosci.*, vol. 39, no. 5, pp. 852–865, Mar. 2014.

[21] M. Beyeler, N. D. Dutt, and J. L. Krichmar, "Categorization and decision-making in a neurobiologically plausible spiking network using a STDP-like learning rule," *Neural Netw. Off. J. Int. Neural Netw. Soc.*, vol. 48C, pp. 109–124, Aug. 2013.

[22] T. P. Vogels and L. Abbott, "Signal propagation and logic gating in networks of integrate-and-fire neurons," *J. Neurosci.*, vol. 25, no. 46, pp. 10786–10795, 2005.

[23] D. Goodman and R. Brette, "Brian: a simulator for spiking neural networks in Python," *Front. Neuroinformatics*, vol. 2, p. 5, 2008.

[24] T. Nowotny, "Flexible neuronal network simulation framework using code generation for NVidia(R) CUDATM," *BMC Neurosci.*, vol. 12, no. Suppl 1, p. P239, 2011.

[25] R. V. Hoang, D. Tanna, L. C. Jayet Bray, S. M. Dascalu, and F. C. J. Harris, "A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling," *Front. Neuroinformatics*, vol. 7, p. 19, 2013.

[26] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk, "NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, Boston, Massachusetts, USA, 2009, pp. 137–144.

[27] T. Bekolay, et al., "Nengo: a Python tool for building large-scale functional brain models," *Front. Neuroinformatics*, vol. 7, p. 48, 2014.

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

[29] D. Pecevski, T. Natschläger, and K. Schuch, "PCSIM: A Parallel Simulation Environment for Neural Circuits Fully Integrated with Python," *Front. Neuroinformatics*, vol. 3, p. 11, 2009.

[30] C. Eliasmith and C. H. Anderson, *Neural Engineering (Computational Neuroscience Series): Computational, Representation, and Dynamics in Neurobiological Systems*. Cambridge, MA, USA: MIT Press, 2002.

[31] S. Rotter and M. Diesmann, "Exact digital simulation of time-invariant linear systems with applications to neuronal modeling," *Biol. Cybern.*, vol. 81, no. 5–6, pp. 381–402, Nov. 1999.

[32] C. Rossant, D. F. M. Goodman, B. Fontaine, J. Platkiewicz, A. K. Magnusson, and R. Brette, "Fitting neuron models to spike trains," *Front. Neurosci.*, vol. 5, p. 9, 2011.