Faster and Simpler SNN Simulation with Work Queues

Dennis Bautembach
FORTH - ICS and *CSD - UOC*
denniskb@ics.forth.gr

Iason Oikonomidis
FORTH - ICS
oikonom@ics.forth.gr

Nikolaos Kyriazis
FORTH - ICS
kyriazis@ics.forth.gr

Antonis Argyros
FORTH - ICS and *CSD - UOC*
argyros@ics.forth.gr

Abstract—We present a clock-driven Spiking Neural Network simulator which is up to 3x faster than the state of the art while, at the same time, being more general *and* requiring less programming effort on both the user's and maintainer's side. This is made possible by designing our pipeline around "work queues" which act as interfaces between stages and greatly reduce implementation complexity. We evaluate our work using three well-established SNN models on a series of benchmarks.

Index Terms—spiking neural networks, SNN, simulation, DSL, work queues

I. Introduction

Spiking neural networks (SNNs) are an important class of artificial neural networks (ANNs) [1]. Compared to other types of ANNs, SNNs more closely mimic the function of biological neural networks, including human brains. One key difference between SNNs and other types of ANNs is the network topology. ANNs are typically modeled as multipartite graphs whereas SNNs are modeled as directed graphs. Another key difference is in the encoding of network activity. ANNs produce continuous outputs whereas SNNs emit so called "spikes" at distinct points in time. The output of a SNN is its firing pattern over time.

SNNs are fundamentally more powerful than first (perceptron, unidirectional, hand-crafted features, discrete output) and second generation ANNs (deep, bidirectional, continuous, trained automatically) [2]. They (re-)gained a lot of traction and popularity in recent years, promising to aid in understanding the function of human brains, to be better suited at processing spatio-temporal data (real-world sensory data), and to potentially outperform Deep Learning and "classical" (feedforward) neural networks, which may reach a saturation point.

Nevertheless, their training and simulation remain unsolved problems. Training is made difficult because of the discontinuous nature of spikes, which prevents us from applying Backpropagation natively (adaptions of Backpropagation exist which approximate spikes with (sharp) continuous functions [3]). On the simulation side, the communication between neurons is the main bottleneck due to neurons being connected randomly (even when those connections are localized). This makes SNN simulation equivalent to the signal propagation problem in directed graphs. In contrast, the regular nature of conventional ANNs allows us to reduce neuron communication to highly efficient matrix multiplications.

We are reminded that GPU acceleration was a major contributor to the Deep Learning revolution, as it enabled training of large-scale ANNs "at home". A similar revolution has yet to happen in the SNNs domain. Towards this end, we present a clock-driven simulator for spiking neural networks, which improves upon the state of the art in terms of performance, generality, and ease of use. It scales to larger networks than our competition and its unique API allows the creation of custom models with minimal programming effort. The simulator is publicly available at https://github.com/denniskb/ijcnn2020.

II. Related Work

Research related to the design and simulation of SNNs has a long history [4], using both hardware [5], [6], and software [7], [8]. Each approach has its own advantages and disadvantages, and distinct approach subclasses can be identified.

Older approaches have capitalized on advances in processing power [7] and hardware design [5]. A more recent trend follows the ubiquitous success of Deep Learning, which largely capitalizes on the advances in commodity GPU processing power [9]–[13]. In all software solutions to the problem, an important distinction can be made [14]: event-driven vs clock-driven. Event-driven methods adopt a mostly asynchronous approach, recording events and processing their implications in an on-demand manner. On the other hand, time- or clock-driven methods advance the whole network state in lockstep. Furber et al. [15] pursue a hybrid approach: their fundamental design revolves around an event queue but they also organize spikes into $100\mu s$ long bins. All spikes from a bin are regarded as taking place simultaneously and thus processed in parallel.

A. Hardware

Designing hardware specifically for the task of SNN simulation has a long history [5], [6], [16] and is still an actively researched area [15], [17], [18]. Custom-designed hardware in earlier attempts [5], [6], [16] has been recently replaced by designing FPGA solutions [17], and partial components, which can be integrated into larger chips [18]. Finally, combinations of software and hardware components [19], as well as very large scale, neuromorphic systems [15] have been presented.

B. Simulators

In parallel to dedicated hardware, a significant amount of effort has been devoted to developing general-purpose
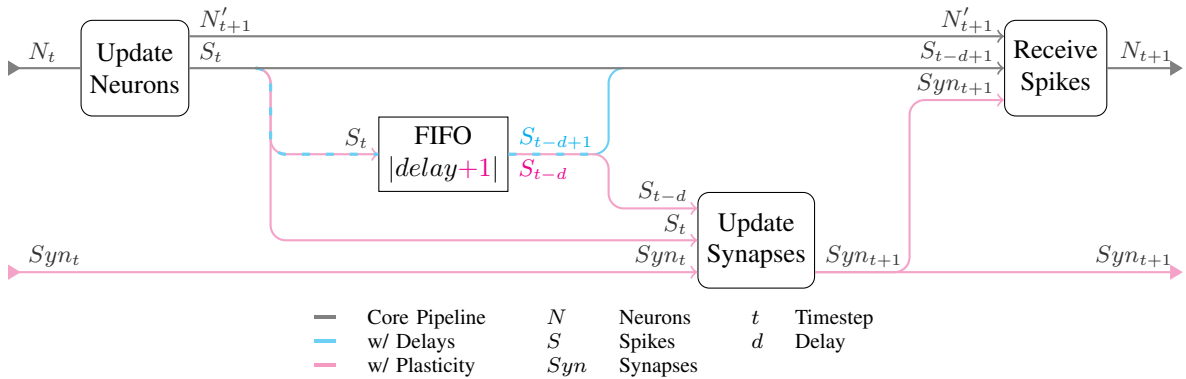
Fig. 1. A single loop iteration of our simulator's pipeline, transitioning the SNN from timestep t to $t+1$. Optional, feature-dependent paths are color-coded. (not depicted) The "Init" stage initializes neurons and synapses to N_0 and Syn_0.

software simulators of SNNs. Such general-purpose simulators include the NEST simulator [20] and the Brian simulator [21] among others [22]. Research on SNNs has recently focused on using software simulation [23]–[26].

C. GPU

Several recent software simulation approaches resort to GPU acceleration. Namely, the projects GeNN [9], Brian2GeNN [27], CARLsim [28], SPIKE [10], Fujita et al. [29], Kasap et al. [30] are all recent (i.e. within the last four years) approaches that employ GPU acceleration. Out of these, GeNN [9] and SPIKE [10] are two actively developed projects and, to the best of our knowledge, are two of the fastest GPU-accelerated SNN simulators, with different trade-offs. SPIKE is heavily optimized for simulation speed, and in doing so, sacrifices some generality and memory efficiency. On the other hand GeNN exposes a versatile programming interface to the user allowing them to implement custom models, at the cost of some speed. We therefore compare the performance of the proposed simulator against these two, as representatives of the state of the art that cover a wide spectrum of trade-offs.

A very recent trend in simulating SNNs is based on the use of the widely popular Deep Learning frameworks. More specifically, BindsNET [31] and SpykeTorch [32] are both based on the PyTorch [33] framework to simulate SNNs. In practice, such approaches do not support general SNN simluation, limiting their applicability.

Overall, the presented SNN simulator leverages the power of consumer GPUs. We adopt a clock-driven approach, and use work-queues for a computationally efficient implementation.

III. METHODOLOGY

A. Architecture

1) Pipeline: We draw inspiration from [34] and organize our simulator into stages which communicate through "work queues": $Stage_i$ produces a queue and $Stage_{i+1}$ consumes it. The queues act as interfaces, decoupling stages, and allow to implement k features with $O(kn)$ effort, n being the number of stages, as opposed to $O(k^n)$ if the stages were tightly coupled. Our core simulation logic is only 70 lines of code long.

On a high level, our simulator consists of three stages: `Init` initializes neurons N and synapses Syn to N_0 and Syn_0. Then the simulation enters a cycle of `Update Neurons`→`Receive Spikes` steps, advancing the SNN from timestep t to $t+1$. `Update Neurons` advances neuron dynamics from N_t to N'_{t+1} and, in doing so, potentially generates a queue of spikes S_t. `Receive Spikes` consumes N'_{t+1} and S_t, delivers the spikes to their recipients, and produces N_{t+1} (Fig. 1, black paths only).

This effectively implements non-instantaneous connections, which are necessary to simulate SNNs that contain cycles. Non-instantaneous connections *imply a synaptic delay of at least 1* timestep. An important feature are arbitrary delays, which allow the simulation of spikes with travel times. Delays are implemented by routing the spikes generated by `Update Neurons` through a first in – first out (FIFO) queue with delay d many entries, *delaying* their arrival at `Receive Spikes` and thus their delivery to their recipients. `Update Neurons` and `Receive Spikes` themselves need not change, they simply write to/read from different queues (Fig. 1, black & blue paths).

Another important feature is spike-timing dependent plasticity (STDP) [35], which can be used to implement Hebbian learning [36], [37]–it is the ability of a synapse to modify its state (typically its weight) depending on local network activity (pre- and postsynaptic spikes). Plasticity is implemented by adding an `Update Synapses` step to the pipeline. `Update Synapses` advances synapse dynamics from Syn_t to Syn_{t+1}, querying S_{t-d} and S_t to determine pre- and postsynaptic spikes respectively (in practice we store bitmasks in addition to queues to speed up those queries). Since we need access to both S_{t-d} and S_t simultaneously, the size of the FIFO queue is increased to $d+1$. `Receive Spikes` now takes into account Syn_{t+1} when delivering spikes to their recipients (Fig. 1, black & pink paths).

In our case we actually implement a lazy variation of plasticity. Rather than eagerly advancing synapse dynamics at every simulation step, we intentionally keep synapses in a stale state and only update them on a need-to-basis, namely if

either of two conditions occur:

- A synapse is about to transmit a spike.
- A synapse is about to "expire", i.e. its age is about to exceed the size of our FIFO queue, after which we would be unable to update the synapse because we would loose access to pre/post-synaptic spike information.

In either case, the synapse is repeatedly updated in a loop until its state is current again. While the amortized number of updates remains identical to the eager version, we still observe a 4x performance gain in practice because the updates are now performed inside registers, avoiding global memory traffic. By abstaining from requiring a closed-form solution for synapse dynamics, which would allow us to update them in a single step (as done, for example, by SPIKE [10]), we stay general and continue to support models that do not have closed-form solutions for their synapse dynamics. Since all out-going synapses of a neuron transmit together, their ages always stay in sync, meaning it is sufficient to store a single age per neuron and synapses can be updated in batches. Lazy plasticity is implemented by letting `Update Neurons` produce one additional queue of "expiring neurons" and `Update Synapses` updating only those synapses belonging to currently spiking or expiring neurons. We also increase the size of our FIFO queue from $d + 1$ to 50 entries (determined empirically) to reduce the frequency of these updates.

2) Data Structures: Subsequently we describe the various data structures employed by our simulator. For brevity, statements regarding neurons also hold for synapses.

We use simple arrays to store most of the SNN state. Users communicate their neurons' fields to us via variadic templates (see section III-B1), which we, using template meta programming, convert into a structure of arrays (SoA)–one array for each field. Except during SNN instantiation and adjacency list construction we have no notion of neuron populations. Instead, users have to create fat neuron layouts and fat callbacks[1]. The advantage is that we can store all neurons in a single SoA, simplifying their traversal, cutting down on kernel invocations, and simplifying indexing into said arrays from the adjacency list.

Queues are simply arrays bundled with an atomic index residing in global memory (for insertions), which is plenty fast for low contention scenarios. They are sized conservatively to avoid re-allocations. Table II shows a detailed breakdown of our simulator's memory consumption.

One of the more interesting data structures is our adjacency list. We use a padded 2D array with $|N|$ rows similar to [9]. Each row stores indices to all its neuron's neighbors. Rows shorter than the maximum degree deg_{max} are padded with sentinels. This introduces a memory overhead of a few percent, but in exchange makes index calculations trivial, cuts down on global memory accesses (for the offset table), and allows us to tune row alignment. In practice, we measure a performance

[1]A structure whose fields are the union of fields of many structures is called fat. A function that incorporates many different code paths, often via a series of if else-switches, is called fat.
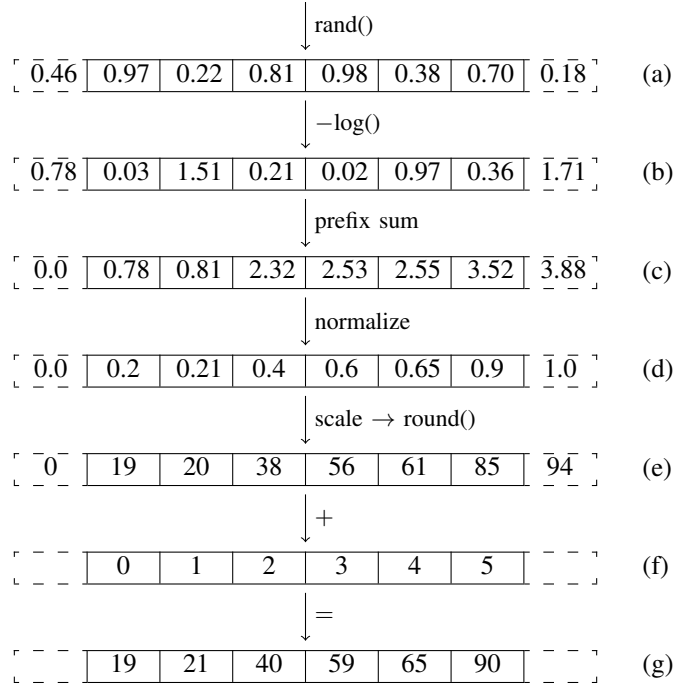

Fig. 2. Efficient generation of sorted, uniformly distributed random integers on the GPU.

gain of a few percent with each row aligned to 128 bytes compared to a compact adjacency list, which requires an additional offset table. In the case of models with synapse state we allocate $|N| * deg_{max}$ synapses with an implicit 1:1 mapping between the adjacency list and the synapse SoA.

As for traversal, we launch one CUDA block [38] per spike, which reads the corresponding row from the adjacency list and delivers the spike to its recipients. We also experimented with launching one warp per spike and with launching one thread per recipient neuron (, which [9] refer to as "postsynaptic parallelism"), both of which performed worse in our benchmarks. Delivering a spike results in poor memory access patterns since all neurons' neighbors are scattered across the whole neuron SoA due to random connectivity. We try to alleviate this somewhat by sorting each row, improving cache locality.

Adjacency list construction is a two step-process. On the CPU, we consume the user-provided SNN description (number and sizes of neuron populations and their connectivity, see listing 1, lines 31 − 32) and generate a queue of jobs $\{(n, a, b, o), ...\}$. Each job can be read as: "Write n sorted, uniformly distributed random integers from the interval $[a, b)$ into the adjacency list starting at offset o". The jobs are uploaded to and processed by the GPU. In order to efficiently generate sequences of sorted random numbers, we take advantage of the fact that the sum of exponentially distributed random numbers is uniformly distributed. Fig. 2 depicts how a job is expanded for $n = 6$ and $[a, b) = [0, 100]$: First, we generate six uniformly distributed random numbers from the interval $(0, 1]$ (2a). We also pad our buffer with two

```cpp
struct ping_pong : model
{
  struct neuron : neuron_desc<bool>
  {
    template <class Iter>
    void init(Iter it)
    {
      if (it.id() < 100)
        it.get<0>() = true;
      else
        it.get<0>() = false;
    }

    template <class Iter>
    bool update(Iter it, float dt)
    {
      bool spike = it.get<0>();
      it.get<0>() = false;
      return spike;
    }

    template <class Iter>
    void receive(Iter from, Iter to)
    {
      to.get<0>() = true;
    }
  };
};

snn<ping_pong> net(
  {100, 100},
  {{0, 1, 0.01}, {1, 0, 0.01}},
  1,
  1
);

while (true)
  net.step();
```

Listing 1. A simple SNN implemented using our framework.

sentinels whose purpose will become apparent shortly. Next, we obtain exponentially distributed numbers by computing the negative (natural) logarithm of these numbers (2b). Afterwards we compute their running sum (2c). At this point we already have obtained a list of sorted, uniformly distributed random numbers. The subsequent steps merely serve to transform this list into the desired range $[a, b)$. We normalize our list (2d), scale it by $b - n = 94$ (2e), and add consecutive integers $\{0, ..., n-1\}$ to it (2f – 2g) in order to ensure that each number is unique (i.e. that the SNN contains only single edges). By using sentinels, the first and last elements of our final list *can* be equal to a and $b - 1$, but need not to. Without them, the final list would always be of the form $\{a, ..., b - 1\}$, which would introduce a non-uniform bias.

In practice, steps (a) – (c) and (d) – (g) can be executed in a single pass, respectively. The first pass is performed inside shared memory while the second pass only writes the final list to global memory. Random numbers are generated directly inside the kernel using the Xorshift RNG [39]. Since the jobs have varying lengths we assign one warp per job. Warps can be scheduled independently from one another and thus do not hold each other hostage like a long-running thread would a block for example. Furthermore, threads of the same warp

can communicate cheaply among themselves via warp-level primitives, making prefix sum calculations very fast. Finally, the CPU part's cost is completely hidden in practice because it is performed in the background of GPU computations. Our adjacency list construction algorithm achieves 90% of the GPU's maximum memory bandwidth.

B. API

Our superior performance (see section IV) does not come at the cost of either generality or usability. On the contrary, we support a wide variety of SNNs by allowing users to implement their own, custom models with minimal programming effort. We present and discuss our API, followed by a comparison with SPIKE and GeNN.

1) API: We draw inspiration from modern graphics pipelines such as DirectX and OpenGL: The GPU facilitates efficient rasterization, texel interpolation, texture filtering, etc, while allowing the user to customize the appearance of the final image through programmable shaders. Similarly, our simulator facilitates efficient spike propagation across the network, handling of delays, plasticity, etc, while allowing the user to customize the behavior of their model by invoking user-defined callbacks.

Let us walk through the implementation of a simple SNN with two randomly inter-connected neuron populations A and B of 100 neurons each that take turns exciting one another (listing 1). We begin by declaring a struct with our model name and inheriting from the "model" interface (line 1). Next we add a child struct "neuron" to our model and communicate our neuron's layout by inheriting from "neuron_desc". In our case neurons have a single field of type bool. Next we have to implement a series of callbacks to define our model's behavior. We start by implementing "init()", which will be called once before the simulation and initialize the first neuron population to true, the second one to false. We get access to our neuron through an iterator (lines 5 – 12). Next we implement "update()", which will be called on every simulation step. If we previously received a spike, we emit one and reset ourselves so we do not spike again until we receive another one (lines 14 – 20). Lastly, we implement "receive()", which will be called whenever we receive a spike from another neuron. If that happens, we simply set our flag to true, which will cause us to spike during the next update step (lines 22 – 26). Now that our model is defined, we can instantiate a SNN with it (line 30), passing neuron populations count and sizes (line 31), neuron populations connectivity (line 32), timestep (line 33), and delay (line 34) — and run our simulation (lines 37 – 38).

The advantages of our approach are:
- **Generality**: Any model that can be expressed using (a) the network state information provided by the framework[2] and (b) the `Update Neurons→Receive Spikes-`

[2] Currently the framework provides the user with information about neuron and synapse states and types, local connectivity, neuron/synapse populations count and sizes, etc., allowing the implementation of a variety of models. In theory, the network's total state could be exposed to the user, but this would defeat the purpose of a framework in the first place, whose role is to strike a balance between enabling users while relieving them at the same time.

simulation loop, can be implemented. The implementation can use any CUDA C features or third party libraries. With this flexibility also comes responsibility: For example, the user has to (remember to) use atomic operations for updating neurons. Such intricacies can mostly be avoided by using existing building blocks, but become necessary when implementing esoteric models ex nihilo.

- **Composability and Reusability**: Class inheritance with method specialization leads itself to composability. Models need not be authored ex nihilo every time, but common components (such as leaky integrate and fire (LIF) neurons) can be extracted into their own classes and reused. Composability in turn increases reusability of such building blocks: If an existing building block does not meet the user's requirements it can be extended and *some* of its functionality specialized, most of it reused. A great example of this approach can be seen in Fig. 3d: We implement the Brunel model (, which consists of Poisson and LIF neurons) by inheriting from the provided LIF neuron type and specializing its update method. Inside we spike randomly if we are a Poisson neuron, and simply delegate the call to the parent method otherwise.

- **Transparency**: The code written ends up being compiled by the native toolchain, making it easy to build a mental model. The compiler is able to provide detailed warnings and error messages. The code can be statically analyzed, debugged, and profiled (e.g. using NVIDIA Nsight).

2) Comparison with SPIKE and GeNN: SPIKE and GeNN employ different means to enable the composition and simulation of SNNs. SPIKE is a runtime library. It ships with a vast collection of popular, highly parameterizable neuron and synapse models that can be used as building blocks. Fixing the building blocks in place lends itself to relentless optimization. However, this comes at the expense of generality: Models that cannot be expressed as a combination of said building blocks, cannot be simulated using SPIKE, unless the authors release new building blocks on-demand. Implicit initialization is performed on the CPU, serially, and with quadratic complexity in the number of synapses. This made it impractical for our experimentation on large networks. To circumvent this and make experiments tractable, we employed user-level multi-threaded (CPU) explicit initialization, with linear complexity. Explicit initialization on the GPU would have been significantly faster, but SPIKE does not provide this option and its API cannot readily accommodate user-level provision. Simulation is performed entirely on the GPU, with the option to "download" timestamped neuron spikes.

GeNN is highly modular and ships with a collection of basic models and building blocks. Implementing a simulation involves a few stages and languages. It amounts to providing the synapse, neuron and plasticity models (using GeNN's domain-specific language (DSL)), running a proprietary compilation step, which brings everything together in a new translation unit of CUDA C++, and linking against said unit from user

(a)   (b)   (c)   (d)

Fig. 3. Comparison of programming effort required to implement the Brunel model in various simulators. Code taken from either official or peer-reviewed samples, excessive comments removed, formatted identically with clang-format. (a) SPIKE (b) GeNN (c) Ours, implemented ex nihilo (d) Ours, implemented using the supplied LIF neuron-building block.

code. Model, network size and connectivity parameters have to be provided at *compile* time, requiring re-compilation for any change. During experimentation we had to maintain several GeNN simulators, for various models and various network sizes, with each one requiring distinct compilation. The user can choose to extend existing models or provide them ex nihilo. Implicit initialization is performed efficiently on the GPU. It is straight-forward to download timestamped spikes.

While designing our simulator, as users of SNN simulation software ourselves, we tended to features that we found to be the most positively impactful in our SNN research, and that will most likely accelerate research for the community, too. It was highly important to us that the entire workflow can be supported by a widespread, well-maintained single tool, like the C++ compiler. Everything, from building to using, was far simpler for SPIKE because of this. We also appreciate GeNN's design choice to provide scaffolding for custom models, rather than fixing building blocks in place, while, at the same time, making stable SNN parts like graph construction, spike propagation, delays, etc. opaque to the user.

We push the envelope on that front and accommodate this functionality by replacing GeNN's DSL paradigm with actual C++ code, compiled by and adhering to the same rules as the tool-chain used for the rest of the framework. Both SPIKE

	SPIKE	GeNN	Ours
Generality	★	★★	★★★
Code Safety	★★★	★★	★
Composability	★★	★★	★★★
Reusability	★	★★	★★★
Transparency	★	★★	★★★
Native Tooling Support	✓	×	✓
Dynamic Parameters	★★★	★	★★★
Ease of Use	★★★	★	★★★

and GeNN, to different degrees, offer the means to reuse components. We too provide this option, to an elevated degree. On the one hand, the user can write arbitrary C++ code to finely control simulation stages. On the other hand, a library of standardized functionality is provided too and is made available for the user to employ when necessary. Through our design choice it naturally derives that established reuse patterns within C++ can be used without restriction, too.

We also wanted to make sure that the required API calls were minimal in count and verbosity. By example (see Fig. 3): it can be seen that our simulator requires the least amount of code to bootstrap. Table I summarizes the qualitative comparison made in this section.

IV. RESULTS

We compare our performance with SPIKE and GeNN in a series of benchmarks using adaptions by [10] of three well-established models: Vogels-Abbott (henceforth referred to as "Vogels") [40], Brunel, and Brunel with plasticity (referred to as "Brunel+") [41]. All models

- use leaky integrate and fire (LIF) neurons.
- subdivide neurons into two groups (aka populations): inhibitory and excitatory. Inhibitory neurons have a high potential leak rate, inhibiting overall network activity. Excitatory neurons have a low to zero leak rate, exciting overall network activity.

They differ in their stimulation, dynamics, and parameterization. In Vogels, a constant background voltage excites all neurons. In Brunel, a population of Poisson firing-neurons excites the remainder of the network. Finally, Brunel can be run with and without STDP. A detailed overview over both models can be found in [10], appendices A & B.

Vogels and Brunel have been conceived nearly 20 years ago and the network sizes they were originally tuned for (4000 neurons and 20,000 neurons respectively) are not remotely large enough to stress modern GPUs and accurately compare SNN simulator performance. Simply increasing the network size alters both models' firing patterns beyond recognition. Therefore we propose a minor change to both models, which

- does not alter the models in any way for the original network sizes.
- retains the models' characteristic firing patterns for all other network sizes (up to billions of synapses).

We do so by scaling the synaptic weights before they are added to the neuron potentials. Normally, when a neuron receives a spike, the weight W of the synaptic connection over which the spike was received is added to the neuron potential V: $V = V + W$, which we change into $V = V + c * W$. c depends on the network size and differs for both models:

$$c = \begin{cases} \dfrac{16{,}000{,}000}{|N|^2}, & \text{Vogels} \\[2ex] \dfrac{20{,}000}{|N|}, & \text{Brunel(+)} \end{cases} \qquad (1)$$

As can be seen, when substituting the original network sizes c becomes 1 and thus has no effect. For larger network sizes, both models retain their characteristic firing patterns. For Vogels, the network's average firing rate (ratio of neurons spiking) remains between half and twice the original rate. For Brunel, the average firing rate remains virtually constant.

The benchmarks were performed on a PC with an Intel Core i7-8700 CPU, 32GB of DDR4 2400 RAM, and an NVIDIA GeForce RTX 2080 Ti GPU.

A. Benchmarks

1) Simulation time as a function of network size: We vary the network size (synapse count) and report the average absolute time in seconds it takes to simulate 10 seconds worth of activity for various models (Fig. 4).

For the tiniest of network sizes, SPIKE is the fastest simulator. However, it is quickly overtaken by us due to its poor scaling. It is also the first simulator to run out of memory, limiting the problem sizes it can be applied to. Compared to GeNN, we are ~3x faster for Vogels, ~1.5x – ~2x faster for Brunel, and just as fast for Brunel+. We also show near-perfect linear scaling for all models and are the last to run out of memory, allowing us to simulate models with up to double the synapse count compared to our closest competitor.

2) Setup time as a function of network size: One important scenario in SNN simulations is running many experiments back-to-back with different parameters/network sizes. Thus, a fast setup time is desirable to maximize time spent inside simulation. Setup consists of three main steps: (1) Constructing the network, (2) initializing neurons and (3) initializing synapses. Steps 2 and 3 are trivial and are really a measure of memory bandwidth rather than algorithmic efficiency. Synapse state dominates setup time for models where it is present. The interesting part is step 1 because it is a costly and complex operation. Therefore, we base this benchmark on Brunel which has no synapse state and simple neuron initialization, making its setup time mostly dependent on graph construction, while still being a real-world model. We vary the network size and report the absolute setup time in seconds. We also benchmark and report our simulator's memory bandwidth (Fig. 5).

SPIKE initializes on the CPU which is why it is 2 orders of magnitude slower than GeNN which initializes on the GPU. Compared to GeNN, which allocates memory for every initialization, we only allocate memory when necessary (using hysteresis). When running many experiments back to
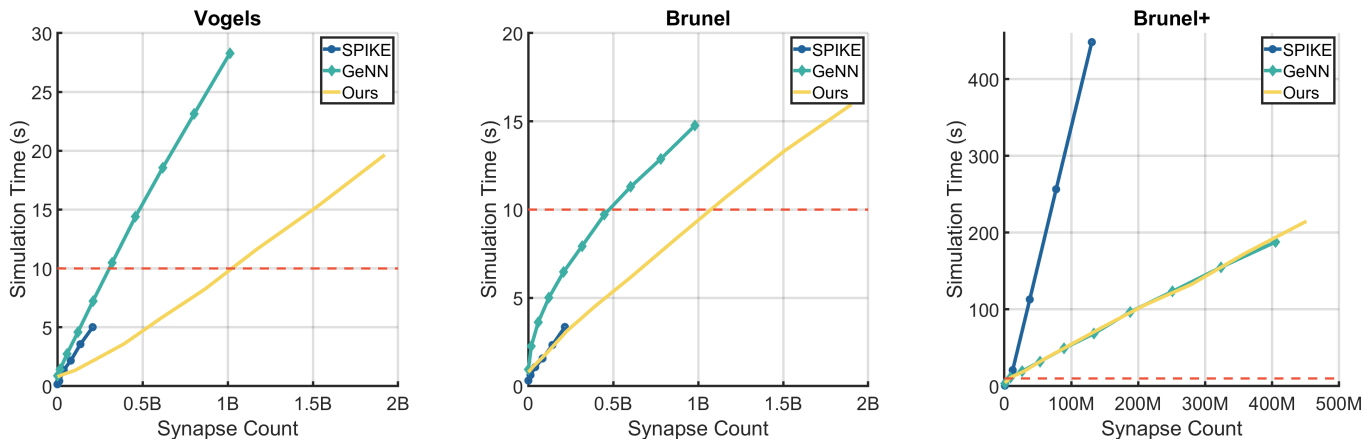
Fig. 4. Simulation time as a function of network size. We vary the synapse count and measure the time in seconds it takes to simulate 10s worth of activity (red dashed line). Left to right: Vogels, Brunel, Brunel+. Graphs are aborted once simulators run out of memory.
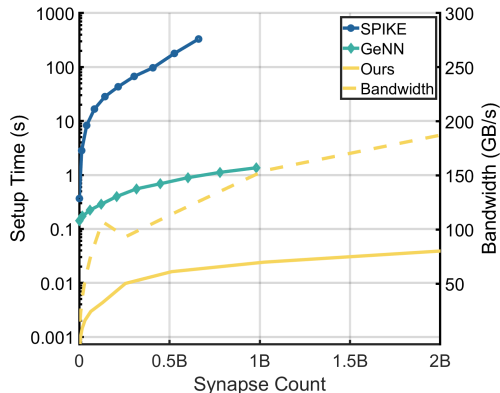

Fig. 5. Setup time as a function of network size for *Brunel* model. Y-axis is logarithmic.
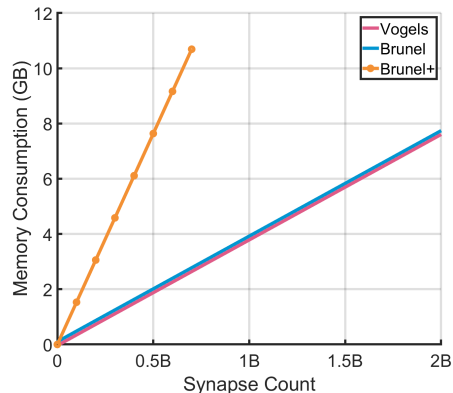

Fig. 6. *Our* simulator's memory consumption for various models as a function of network size.

back we thus often pay the price of allocation only once at the beginning. This, bundled with our highly efficient graph construction algorithm, gives us another 2 orders of magnitude improvement over GeNN. Fig. 5 clearly illustrates that the notion of "setup time not mattering because it only happens once" is a fallacy. In the time it takes SPIKE to initialize a network with 500 million synapses, we can already simulate 6 minutes worth of activity.

3) Memory consumption as a function of network size: We report *our* simulator's memory consumption in gigabytes. It becomes apparent that it is entirely dominated by synapse count. For a detailed breakdown of the total memory consumption see Table II.

V. CONCLUSIONS & FUTURE WORK

We presented a SNN simulator which is faster and consumes less memory than the state of the art, allows the specification of more general models, offers a simpler and less verbose API and build process. Spiking Neural Networks regained a lot of traction and popularity recently. In spite of simulation improving by two orders of magnitude in the last year alone, SNNs still have a long way to go in order to compete with conventional ANNs. There is one virtually unexplored optimization in SNN simulation: Mutli-GPU parallelization. A promising approach might be to parallelize simulation across neuron populations, similarly to how PipeDream [42] parallelizes backpropagation across layers.

TABLE II
DETAILED BREAKDOWN OF OUR SIMULATOR'S MEMORY CONSUMPTION

Model		Vogels	Brunel	Brunel+
per neuron	fields	16B	8B	8B
	spikes	32B	60B	60B
	bitmasks	-	-	6.25B
	ages	-	-	4B
	expirations	-	-	4B
	total	48B	68B	82.25B
per synapse	adjacency list	4B	4B	4B
	fields	-	-	12B
	total	4B	4B	16B

REFERENCES

[1] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[2] ——, "Lower bounds for the computational power of networks of spiking neurons," *Neural computation*, vol. 8, no. 1, pp. 1–40, 1996.

[3] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, 2018.

[4] W. S. McCulloch and W. H. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[5] M. S. Tomlinson, "Spike Transimission for Neural Networks," 1990.

[6] F. J. Pelayo, E. Ros, X. Arreguit, and A. Prieto, "VLSI Implementation of a Neural Model Using Spikes," *Neuron*, vol. 121, pp. 111–121, 1997.

[7] M. Mattia and P. Del Giudice, "Efficient event-driven simulation of large networks of spiking neurons and dynamical synapses," *Neural Computation*, vol. 12, no. 10, pp. 2305–2329, 2000.

[8] J. Reutimann, "Event-driven simulation of spiking neurons with stochastic dynamics," *Neural Computation*, no. 2593, 2002.

[9] E. Yavuz, J. Turner, and T. Nowotny, "GeNN: A code generation framework for accelerated brain simulations," *Scientific Reports*, vol. 6, no. June 2015, pp. 1–14, 2016.

[10] N. Ahmad, J. B. Isbister, T. S. C. Smithe, and S. M. Stringer, "Spike: A GPU Optimised Spiking Neural Network Simulator," *bioRxiv*, p. 461160, 2018.

[11] C. M. Thibeault, R. Hoang, and F. C. Harris, "A novel multi-GPU neural simulator," *3rd International Conference on Bioinformatics and Computational Biology 2011, BICoB 2011*, no. 1, pp. 146–151, 2011.

[12] A. K. Fidjeland and M. P. Shanahan, "Accelerated simulation of spiking neural networks using GPUs," *Proceedings of the International Joint Conference on Neural Networks*, pp. 1–8, 2010.

[13] A. Fernandez, R. San Martin, E. Farguell, and G. E. Pazienza, "Cellular neural networks simulation on a parallel graphics processing unit," *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications*, no. July, pp. 208–212, 2008.

[14] J. A. Garrido, R. R. Carrillo, N. R. Luque, and E. Ros, "Event and Time Driven Hybrid Simulation of Spiking Neural Networks," no. June 2011, 2014.

[15] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the SpiNNaker system architecture," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2454–2467, 2013.

[16] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, "A wafer-scale neuromorphic hardware system for large-scale neural modeling," *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems*, pp. 1947–1950, 2010.

[17] A. Sripad, G. Sanchez, M. Zapata, V. Pirrone, T. Dorta, S. Cambria, A. Marti, K. Krishnamurthy, and J. Madrenas, "SNAVAA real-time multi-FPGA multi-model spiking neural network simulation architecture," *Neural Networks*, vol. 97, pp. 28–45, 2018.

[18] D. Lee, G. Lee, D. Kwon, S. Lee, Y. Kim, and J. Kim, "Flexon: A flexible digital neuron for efficient spiking neural network simulations," *Proceedings - International Symposium on Computer Architecture*, pp. 275–288, 2018.

[19] E. Ros, E. M. Ortigosa, R. Agís, R. Carrillo, and M. Arnold, "Real-time computing platform for spiking neurons (RT-spike)," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 1050–1063, 2006.

[20] J. M. Eppler, "PyNEST: A convenient interface to the NEST simulator," *Frontiers in Neuroinformatics*, vol. 2, no. January, pp. 1–12, 2009.

[21] D. F. M. Goodman, "The Brian simulator," *Frontiers in Neuroscience*, vol. 3, no. 2, pp. 192–197, 2010.

[22] D. Pecevski, D. Kappel, and Z. Jonke, "NEVESIM: event-driven neural simulation framework with a Python interface," *Frontiers in Neuroinformatics*, vol. 8, no. August, pp. 1–20, 2014.

[23] M. Rudolph and A. Destexhe, "Analytical integrate-and-fire neuron models with conductance-based dynamics for event-driven simulation strategies," *Neural Computation*, vol. 18, no. 9, pp. 2146–2210, 2006.

[24] E. Ros, R. Carrillo, E. M. Ortigosa, B. Barbour, and R. Agís, "Event-driven simulation scheme for spiking neural networks using lookup tables to characterize neuronal dynamics," *Neural Computation*, vol. 18, no. 12, pp. 2959–2993, 2006.

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

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

[27] M. Stimberg, D. F. M. Goodman, and T. Nowotny, "Brian2GeNN: a system for accelerating a large variety of spiking neural networks with graphics hardware," *bioRxiv*, p. 448050, 2018.

[28] T. S. Chou, H. J. Kashyap, J. Xing, S. Listopad, E. L. Rounds, M. Beyeler, N. Dutt, and J. L. Krichmar, "CARLsim 4: An Open Source Library for Large Scale, Biologically Detailed Spiking Neural Network Simulation using Heterogeneous Clusters," *Proceedings of the International Joint Conference on Neural Networks*, vol. 2018-July, pp. 1158–1165, 2018.

[29] K. Fujita, S. Okuno, and Y. Kashimori, "Evaluation of the computational efficacy in GPU-accelerated simulations of spiking neurons," *Computing*, vol. 100, no. 9, pp. 907–926, 2018.

[30] B. Kasap and A. J. van Opstal, "Dynamic parallelism for synaptic updating in GPU-accelerated spiking neural network simulations," *Neurocomputing*, vol. 302, pp. 55–65, 2018.

[31] H. Hazan, D. J. Saunders, H. Khan, D. T. Sanghavi, H. T. Siegelmann, and R. Kozma, "BindsNET: A machine learning-oriented spiking neural networks library in Python," vol. 12, no. December, pp. 1–18, 2018.

[32] M. Mozafari, M. Ganjtabesh, A. Nowzari-Dalini, and T. Masquelier, "SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks with at most one Spike per Neuron," no. Mm, pp. 1–16, 2019.

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NeurIPS Autodiff Workshop*, 2017.

[34] K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and faster hlbvh with work queues," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, 2011, pp. 59–64.

[35] M. Taylor, "The problem of stimulus structure in the behavioural theory of perception," *S. African J. Psychology*, vol. 3, pp. 23–45, 1973.

[36] R. Kempter, W. Gerstner, and J. L. Van Hemmen, "Hebbian learning and spiking neurons," *Physical Review E*, vol. 59, no. 4, p. 4498, 1999.

[37] K. Kozdon and P. Bentley, "The evolution of training parameters for spiking neural networks with hebbian learning," in *Artificial Life Conference Proceedings*. MIT Press, 2018, pp. 276–283.

[38] Nvidia, "Nvidia cuda c programming guide," 2019, [Online; accessed 4-April-2020].

[39] G. Marsaglia *et al.*, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[40] T. P. Vogels and L. F. Abbott, "Signal propagation and logic gating in networks of integrate-and-fire neurons," *Journal of neuroscience*, vol. 25, no. 46, pp. 10 786–10 795, 2005.

[41] N. Brunel, "Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons," *Journal of computational neuroscience*, vol. 8, no. 3, pp. 183–208, 2000.

[42] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.