Accelerated Simulation of Spiking Neural Networks Using GPUs

Andreas K. Fidjeland and Murray P. Shanahan

***Abstract*— Spiking neural network simulators provide environments in which to implement and experiment with models of biological brain structures. Simulating large-scale models is computationally expensive, however, due to the number and interconnectedness of neurons in the brain. Furthermore, where such simulations are used in an embodied setting, the simulation must be real-time in order to be useful. In this paper we present a platform (nemo) for such simulations which achieves high performance on parallel commodity hardware in the form of graphics processing units (GPUs). This work makes use of the Izhikevich neuron model which provides a range of realistic spiking dynamics while being computationally efficient. Learning is facilitated through spike-timing dependent synaptic plasticity. Our GPU kernel can deliver up to 550 million spikes per second using a single device. This corresponds to a real-time simulation of around 55 000 neurons under biologically plausible conditions with 1000 synapses per neuron and a mean firing rate of 10 Hz.**

I. Introduction

Models of spiking neural networks (SNNs) are important tools in the quest for understanding the brain. Such models make use of the precise timing of spikes from discrete neuron firings to produce rich dynamic behaviours [1]. Simple neuron models allow the construction of large networks of sufficient scale and biological realism to produce network-level behaviour which can be studied to fill the gap between models at lower (cellular, molecular) and higher (functional) levels. The simulation of biologically inspired networks based on even simple neuron models is computationally demanding, however, due to the sheer size of such networks and their high levels of interconnectedness; simulating a whole primate brain (leaving aside the issue of how to connect it!) is beyond the capability of even the most powerful supercomputers of today.

The study of cognitive systems can be much aided by considering embodiment, where the system can interact with its environment through a (robotic) body. Indeed, a thorough understanding of cognitive systems may never be achieved unless the study is lifted out of the purely abstract in this way. Employing SNNs in such an embodied setting places additional demands on the neural simulator, as the simulation must now be at least real time in order to support interaction between the robot and its environment.

The need for fast simulation of large networks coupled with the naturally parallel nature of neural networks, has led practitioners to build simulators on a range of different platforms, exploiting parallelism on custom hardware [2], [3] and large-scale clusters [4], [5]. Such large-scale and customised solutions can be costly to build and design. The current technological trend, however, is towards increased levels of parallelism also within consumer-grade devices, which offers alternative opportunities for accelerating simulations. This trend is fully realised in the latest Graphics Processing Units (GPUs). Modern GPUs combine hundreds of cores with very high memory bandwidth at consumer prices. This paper presents a method for simulating networks of spiking neurons based on Izhikevich's phenomenological model [6] on such devices, with the aim to make large-scale modelling more accessible to the computational neuroscience community.

The key contributions of this work are:

- A method for simulating spiking neural networks with conduction delays on highly parallel graphics processing units (Section III). The simulator relies on data organisation tailored to the idiosyncrasies of the GPU memory architecture.
- An extension to the basic simulator to incorporate synaptic plasticity in a user-configurable manner (Section IV).
- An evaluation of the resulting simulation kernel using spiking networks with more than 30 thousand neurons and 30 million synapses (Section V). The kernel can deliver up to 550 million spikes per second on current hardware, outperforming existing GPU kernels by a factor of 4.5.

II. Background

A. Neuron model

In the spiking neural networks we consider, neurons are modelled as point-entities which produce discrete spikes. These spikes correspond to the action potentials resulting from the depolarisation of the neuron membrane. Neurons connect via chemical synapses through which spikes are communicated from the presynaptic to the postsynaptic neuron. The details of this neuron-to-neuron communication is greatly simplified with respect to full biological models, but two essential aspects are retained. First, the presynaptic neuron can induce a current in the postsynaptic neuron. The magnitude of this current varies between synapses and can be either positive (excitatory) or negative (inhibitory). Second, synapses have an associated conductance delay, i.e. a delay between the time a spike is generated in the presynaptic neuron and the time when the spike affects the postsynaptic neuron. These delays can range from one to a few tens of milliseconds.

The neurons in a spiking neural network can be simulated at multiple levels of abstraction, where there is a general

Andreas K. Fidjeland and Murray P. Shanahan are both with the Department of Computing, Imperial College London, London, United Kingdom, email: {andreas.fidjeland,m.shanahan}@imperial.ac.uk

trade-off between biological plausibility and computational efficiency. Among computationally tractable models, Izhikevich provides a good phenomenological model [6]. The model consists of a two-dimensional system of ordinary differential equations defined by

$$\dot{v} = 0.04v^2 + 5v + 140 - u + I \tag{1}$$

$$\dot{u} = a(bv - u) \tag{2}$$

with an after-spike resetting

$$\text{if } v \geq 30 \text{ mV, then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{3}$$

where v represents the membrane potential and u the membrane recovery variable, accounting for the activation of K^+ and the inactivation of Na^+ providing post-potential negative feedback to v. The parameter a describes the time scale of the recovery variable, b describes its sensitivity to sub-threshold fluctuations, c gives the after-spike reset value of the membrane potential, and d describes the after-spike reset of the recovery variable. The variables a–d can be set so as to reproduce the behaviour of different types of neurons. The term I in Equation 1 represents the combined current from spike arrivals from all presynaptic neurons, which are summed every simulation cycle.

Learning in spiking neural networks can be facilitated by altering the conductance of synapses in response to patterns of network activity. A number of mechanisms have been observed in biological networks, operating at different time scales. In spike-timing dependent plasticity (STDP) the synaptic modification depends on the relative timing of pre- and postsynaptic action potentials. This time-dependence allows the mechanism to learn aspects of causality, and to facilitate competition between synapses in their effect on the postsynaptic action potential [7].

STDP operates on the timescale of tens of milliseconds before and after the postsynaptic firing. Spikes arriving shortly before or shortly after this firing, modifies the synaptic conductance in a way that depends on the time difference Δt, between spike arrival and postsynaptic firing. The size of the temporal window before and after the postsynaptic firing, may be of different size, and the STDP function may take different shapes depending on the pairs of neurons involved [8].

B. Parallel SNN simulation

The need for high performance in the simulation of spiking neural networks naturally leads to the use of parallelism. Synapses outnumber neurons by several orders of magnitude, and even at low firing rates spike delivery dominates simulations. Since the current from incoming spikes is simply summed, only a small amount of computation is needed for each datum. The simulation is thus a problem bounded mainly by memory and communication rather than by computation.

For the simulation of very large networks a computer cluster is a natural choice, as the very large amount of memory required to store the connectivity data can be distributed across a number of nodes. Ananthanarayanan et al. [4] and Izhikevich et al. [5] are examples of such simulations, simulating 10^9 and 10^{11} neurons respectively. In both cases the focus is on overall simulation throughput rather than real-time performance. In this work we consider the performance on single devices, i.e. the performance on what would constitute a single node in the above clusters. Clearly, performance improvements at this level can be exploited in a larger cluster architecture, but the performance issues for single devices are different.

For very small networks, all the data can be stored on the chip. In field-programmable gate array (FPGA) implementations Thomas and Luk [2] and Cheung et al. [3] do this to achieve very high speedups (100 to 1000 times real-time). Even with the large amount of on-chip memory available, this is only possible for very small networks of no more than 1000 densely connected neurons.

For simulations of larger networks, the storage of connectivity must be done off-chip, in memory. Consequently the communication and storage challenge becomes a challenge of maximising the utilisation of memory. Without good memory performance, the resources available on a chip cannot be employed fully. Generally, memory should be organised such that data which are accessed at the same time are located close together. This is true for single- and multi-core CPUs, but is especially so for GPUs due to the idiosyncrasies of its memory architecture. Organising data such that accesses are optimal is hard due to the fact that neurons can have a large number of outgoing synapses to a dispersed set of postsynaptic neurons, and also a large number of incoming synapses from another dispersed set of presynaptic neurons. Some amount of spike-related operations must take place for both the presynaptic and postsynaptic neuron, and ensuring data locality is generally only possible for one of them.

Some existing spiking neural network simulators exist for GPUs. Nageswaran et al. describe the GPU-SNN simulator [9]. The main difference with our simulator is in the way the synaptic data is organised and processed. GPU-SNN stores synaptic data on per-postsynaptic neuron basis, pulling data in for each neuron for active synapses. This places limitations on the number of synapses each neuron can have. In contrast we store the synaptic data on a per-*pre*synaptic neuron basis, instead fanning data out only for the neurons which fire. Our approach should result in better memory access patterns. GPU-SNN has the advantage of using a highly compact storage format for the connectivity.

In previous work [10] we present an alternative GPU kernel, albeit one without STDP support. This kernel partitions the network in a way similar to our current work, but then makes a distinction between 'local' (within a partition), and 'global' (between partitions) connections, and uses different mechanisms for the spike delivery for these two classes. While the spike delivery for local spikes is efficient, per-

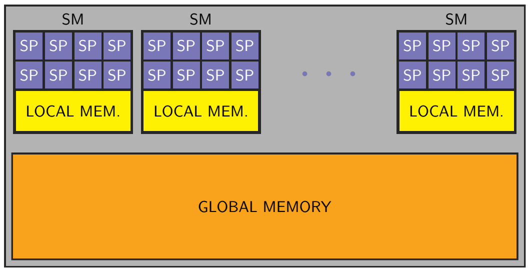

Fig. 1. Generic GPU architecture with a collection of loosely coupled streaming multiprocessors (SM), each composed of parallel scalar processors (SP), and a two-level memory hierarchy.

formance for global spike delivery is low, limiting overall performance. This is partly due to the queueing of individual spikes, requiring a large amount of memory bandwidth. Our current approach organises the connectivity data such that this memory bandwidth requirement is greatly reduced.

C. GPU architecture

Our simulator targets a parallel Graphics Processing Unit (GPU) based on the CUDA architecture (Figure 1). Such devices are divided into multiple single-instruction multiple-thread (SIMT) *streaming multiprocessors* (SMs), each of which consists of several *scalar processors* (SP). A *kernel*, the program executed by the device, is run in parallel using a large number of threads. These threads are split into blocks, each of which execute on a single multiprocessor. Thread blocks typically have far more threads than there are scalar processors on the multiprocessor, so the thread blocks are further subdivided into smaller groups. The multiprocessor switches between these groups, executing the threads within each group in lock-step. All threads execute the same program, but operate on different parts of the data.

Multiprocessors operate largely independently. They can communicate via global memory, but can only be synchronised via the host system (by invoking separate kernels). *Within* a multiprocessor, however, the scalar processors can coordinate their execution using barrier synchronisation. Furthermore, each multiprocessor has a small local memory, which scalar processors within the same multiprocessor can use for exchanging data. This local memory works as a programmer-managed buffer or cache.

One of the advantages of modern GPUs is the very large amount of off-chip (global) memory bandwidth available. Utilising this bandwidth, however, relies on specific memory access patterns being used. The requirements differ slightly between different devices, but generally memory should be accessed such that groups of consecutive threads on the same multiprocessor access groups of consecutive words in memory. In the Nvidia nomenclature, memory is accessed by a group of threads called a *warp* (consisting of 32 threads on current devices), and the well-aligned memory access just described is said to be be *coalesced*. Accessing non-consecutive words in global memory from within a warp is of course possible, but the accesses are serialised. Organising data and execution such that memory accesses follow these patterns is of prime importance for achieving high performance on memory-bound kernels, and is one of the main design principles behind our kernel.

III. Simulation Kernel

A. Overview

Our simulation kernel is discrete-time, with a temporal resolution of one millisecond. Each simulation step involves the invocation of a single kernel, consisting of the following five main steps:

1) *initialise* incoming current using a per-neuron random process;
2) *gather* incoming current for each neuron, resulting from previous firings of its presynaptic neurons;
3) *update* neuron state according to Equations 1–3;
4) accumulate *stdp* statistics (potentiation and depression) for plastic synapses;
5) *scatter* outgoing spikes for fired neurons

Of these, the steps related to spike delivery (2 and 5), and the accumulation of synaptic plasticity statistics (4) are by far the most computationally demanding.

To parallelise the simulation, the network is divided into a number of partitions each of which is executed on a single multiprocessor. The size of each partition is limited by the need for local storage during the different kernel stages. On current devices we set the partition size to 1024 neurons, but future devices will support larger partitions. Neurons are indexed using both the partition index and the index of the neuron within the partition.

The kernel supports an optional randomised input current for each neuron (step 1). This random input current is generated on the device using a Gaussian random number generator [11] whose parameters can be set individually for each neuron. The random number generator state is also stored on a per-neuron basis, so the input current generation can be trivially parallelised.

The update of each neuron (step 3) is independent of the others, given that the input current I has already been computed, and is thus also trivially parallelisable across the available cores. Furthermore, the required memory accesses to the state vectors (u and v) and parameter vectors (a–d), have ideal access patterns for this architecture.

Spike delivery is done in two parts (scatter and gather), and is facilitated by a spike queue. Our kernel organises synapses into small groups which can be accessed through coalesced memory accesses. Because of this grouping only indices to the groups need to queued, rather than individual spikes.

There are three data structures associated with spike delivery: the *forward synapse* matrix, which stores the raw synapse data; the *forward group* matrix, which provides an index into the *forward synapse* matrix; and the incoming *group queue*, which contains a compressed representation of the spikes due for delivery. These three data structures are organised so as to strike a balance between the competing goals of 1) achieving good coalescing of memory accesses

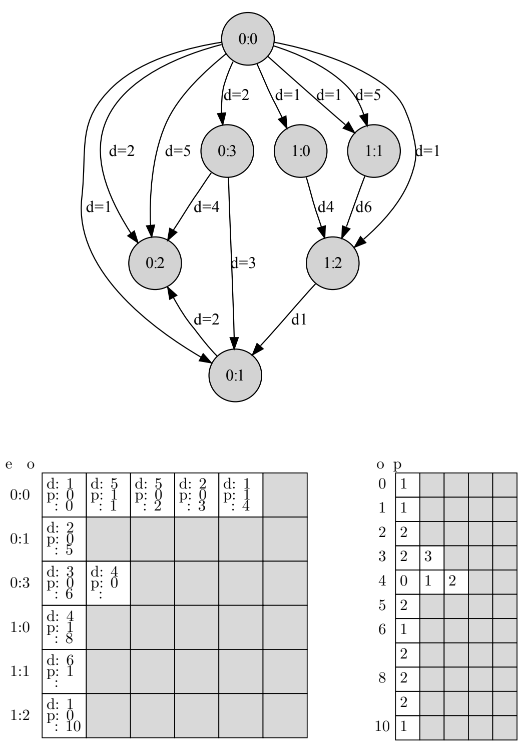

Fig. 2. Memory organisation of synapse data. A small network (top) with seven neurons (labelled with partition/neuron indices) in two partitions, and synapses with different delays. This network is stored in two data structures. The synapse groups (bottom left) stores triplets of target partition, delay, and group address. The synapse data proper (bottom right) stores the target neuron and weights (not shown).

during spike delivery; 2) reducing the overall number of costly global memory accesses; and 3) keeping the overall memory usage near-linear in the number of synapses.

B. Synapse data format

The raw synapse data, consisting of the weight and target neuron (within a partition), is stored in the *forward synapse* data structure (FCM in Algorithm listings). The weight is stored in a fixed-point format, with the number of fractional bits set based on the range of weights in the input network. Synapses are sorted into bundles sharing the same presynaptic neuron, delay, and target partition. These bundles contain synapses which always need to be delivered at the same time. Bundles are further split into synapse groups no larger than w, where w is the size of a warp on the device. These synapse groups are the basic units of spike delivery. Figure 2 shows an example of a simple network and its mapping to these data structures.

Neither the target partition nor the conductance delay are explicit in the forward synapse data, but are instead stored in the smaller *forward group* data structure (GROUPS in Algorithm listings). This is a matrix with one row per neuron, each specifying all the synapse groups associated with the relevant neuron, regardless of delay or target partition. Each entry within the row contains three data: the target partition, the conductance delay, and the address of the relevant group in the forward synapse data.

The data organisation described above has two advantages. First, the two-level organisation means that whole groups of synapses can be queued cheaply. Second, with suitable memory alignment, reading a synapse group always results in a coalesced read.

Memory for storing the synapse data can be a limiting factor to scaling the network size due the potentially very large number of synapses in networks of interest. Ideally, therefore, we would like memory to grow linearly in the number of synapses. The organisation of the forward connectivity matrix into synapse groups leads to a worst-case scaling that is super-linear, due to the presence of padding at the end of groups which are not full. The above data structure can be compacted into a linear-growth data structure, by combining several small groups into the same *forward synapse* matrix row, and adding mask data to the *forward group* data to indicate the valid synapses within the group.

C. Group queue

The *incoming group queue* (QUEUE in Algorithm listings) stores the synapse groups which are due for delivery. Each (target) partition has its own rotating queue, with the number of slots equal to the maximum conductance delay. Each entry within the slot contains the address of the relevant synapse group, and the slot as a whole contains all the synapse groups due for delivery during a particular cycle. Neither the source partition/neuron nor delay are relevant to the receiving partition/neuron.

The incoming group *queue* should be large enough to avoid overflowing. The fill rate of the queue, however, depends on the activity level in the network. In the worst-case scenario (every neuron constantly firing), the incoming spike queue will be of a size of the same order of magnitude as the *forward synapse* data. This worst-case scenario corresponds to pathological network behaviour; neurons will typically have some refractory period between firing, and even if a particular neuron has a high rate of firing for some period, it is unlikely that *all* other neurons also has so at the same time. The queue size can therefore normally be reduced by an order of magnitude, ensuring that its memory requirement is not a limiting factor. This will still support a sustained firing rate of 100Hz for the entire network.

D. Spike scatter

In the *scatter* step all the neurons which just fired (determined in the neuron update step) are processed sequentially (Algorithm 1, see Appendix A for notation). For each fired neuron, a whole row from the *forward group* matrix is loaded, and placed (in parallel) in the appropriate group queue slot for later processing in the target partition's *gather* step some simulation steps in the future. The reading of forward group data is fully coalesced. The writing of synapse groups to the queue is only partially coalesced, however. Neighbouring synapse groups may target different partitions, and hence end up being written to non-neighbouring queue entries. The *forward group* data structure can be organised so that groups from a single bundle, or from different bundles with the same target partition are placed in consecutive entries, and hence lead to *partial* coalescing. Determining the appropriate queue slot for each group is somewhat costly as it requires global memory atomic operations (in 'nextFree' in the listings). If many synapse groups target the same partition's queue, this can lead to serialisation.

Algorithm 1 Spike scatter step (for one partition)

```
Input: $p$: current partition; $\vec{n}_{\text{fired}}$: list of firing neurons
    in current partition, $t$: current cycle, $d_{\text{max}}$: maximum
    conductance delay
Output: QUEUE updated to reflect this cycle's firings.
1: for all $n \in \vec{n}_{\text{fired}}$ do
2:     $(\vec{p}_{\text{post}}, \vec{d}, \vec{g}) \Leftarrow \text{GROUPS}[p, n, :]$
3:     $\vec{q}_{\text{slot}} \leftarrow (t + \vec{d}) \bmod d_{\text{max}}$
4:     $\vec{q}_{\text{entry}} \Leftarrow_{\text{NC}} \text{nextFree}(p_{\text{post}}, \vec{q}_{\text{slot}})$
5:     $\text{QUEUE}[\vec{p}_{\text{post}}, \vec{q}_{\text{slot}}, \vec{q}_{\text{entry}}] \Leftarrow_{\text{NC}} \vec{g}$
6: end for
```

See Appendix A for notation

E. Spike gather

In the *gather* step (Algorithm 2), all spikes due for delivery in the current simulation cycle are accumulated in each neuron's incoming current buffer, thus computing the I-term in Equation 1. Since there is one queue for each partition, this is done in parallel on the device-level, with each multiprocessor dealing with one or more partitions. For each partition, all the relevant entries in the queue are located in consecutive memory entries. These incoming synapse group addresses are therefore loaded in parallel, in a fully coalesced read.

Algorithm 2 Spike gather step (for one partition)

```
Input: $p$: current partition; $t$: current cycle; $d_{\text{max}}$: maximum
    conductance delay
Output: per-neuron current vector $\vec{I}$
1: $\vec{q}_{\text{slot}} \leftarrow t \bmod d_{\text{max}}$
2: $\vec{g}_{\text{post}} \Leftarrow \text{QUEUE}[p, \vec{q}_{\text{slot}}, :]$
3: for all $\vec{i} \in$ groups of 8 from $0 : |\vec{g}_{\text{post}}|$ do
4:     $(\vec{w}, \vec{n}_{\text{post}}) \Leftarrow \text{FCM}[\vec{g}_{\text{post}}[\vec{i}], :]$
5:     atomicAdd$(\vec{I}[\vec{n}_{\text{post}}], \vec{w})$
6: end for
```

See Appendix A for notation

Each loaded group corresponds to several synapses which should induce a current in the target neurons. Several of these groups are processed in parallel, such that each thread processes one synapse. Because of the organisation of the forward connectivity matrix into synapse groups, the loading of synapse data is again fully coalesced.

To avoid a race condition when two synapses are incident on the same target neuron, the update of the current vector is done using atomic addition operations on local memory. These are supported in hardware, but only for integers. This is one of the reasons for using a fixed-point rather than a floating-point format for the synaptic weights, as ensuring atomicity otherwise requires computationally expensive workarounds. Arithmetic overflow in the current accumulation is avoided by using saturating arithmetic, with clamping to the minimum and maximum representable values in the current fixed-point format.

Some performance overhead is caused by threads running idle due to partially filled synapse groups. If the synapse groups are consistently smaller than a warp, this will affect a large number of threads. A high level of clustering within the network may thus lead to improved performance.

IV. Spike-timing dependent plasticity

A. Plasticity model

Our implementation of STDP (See Section II-A) supports a single user-specified STDP function, of arbitrary shape and variable length. This is specified by providing the function values at the integer values of Δt, both positive (for post-pre pairs) and negative (for pre-post pairs). The total size of the STDP window is currently limited to 64 ms - d_{max}, where d_{max} is the maximum conductance delay in the network. Synaptic plasticity can be enabled on a per-synapse basis. Pairs of spikes are considered, using a reduced symmetric nearest neighbour spike pairing scheme [12]. Our implementation also models weight saturation, which is specified by a maximal weight (for excitatory synapses) and a minimum weight (for inhibitory synapses).

B. Data structures

Dealing with synaptic plasticity requires the addition of three data structures: First, the reverse connectivity matrix ('RCM' in Algorithm listings) contains per-neuron synaptic data for the *incoming* plastic synapses. Each synapse in the reverse connectivity matrix specifies the presynaptic neuron (partition and neuron index), the index of the relevant synapse in the forward synapse matrix, and the conductance delay of the synapse. Second, the STDP accumulator ('ACC' in listings) stores the accumulated weight potentiation and depression for each synapse. The format of the accumulator mirrors the reverse connectivity matrix. Third, the *firing history* ('HISTORY' in listings) stores in a compact format all the recent firing activity on a per-neuron basis.

The firing history is a per-neuron bit vector where the set bits indicates that the neuron fired during a particular cycle. This firing history is updated during the neuron state update step, once the firing status for each neuron is known. For pairs of neurons connected by a synapse, the two relevant history bit-vectors along with the conductance delay contains sufficient information to determine all recent relevant pre-post and post-pre spike pairs which could result in potentiation or depression.

C. Accumulating STDP statistics

The firing of a particular (postsynaptic) neuron can lead to either potentiation or depression of its incoming synapses. The accumulation of these per-synapse statistics is only performed once all presynaptic firings relevant to the postsynaptic firing have taken place, i.e. once the whole STDP window surrounding the postsynaptic firing is in the past. During each simulation step, the kernel inspects the recent firing history and determines which (postsynaptic) neurons have firings which may now be processed.

These firing postsynaptic neurons are processed sequentially, and for each such neuron its incoming synapses are processed in parallel (Algorithm 3). Each synapse in the reverse connectivity matrix incident on the fired neuron is associated with a single presynaptic neuron. The recent firing history of these presynaptic neurons are loaded (again in parallel, but now non-coalesced). With the compact firing histories of both the pre- and postsynaptic neuron available, any relevant pre-post and post-pre pairs as well as the associated spike timing difference can be determined using simple bit-manipulation. Both potentiation and depression is determined at the same time, and the STDP accumulation matrix is updated with this combined value (using the function 'stdp' in the listings).

Algorithm 3 STDP accumulation step

```
Input: $t$: current cycle; $t_{\text{postfire}}$: length of postfire part of
    STDP window
1: $\vec{h} \Leftarrow \text{HISTORY}[p, :]$
2: $(\vec{n}_{\text{fired}}, \vec{h}_{\text{fired}}) \leftarrow \text{fired\_at}(\vec{h}, t - t_{\text{postfire}})$
3: for all $(n_{\text{post}}, h_{\text{post}}) \in \vec{n}_{\text{fired}} \times \vec{h}_{\text{fired}}$ do
4:     $(\vec{p}_{\text{pre}}, \vec{n}_{\text{pre}}, \vec{s}) \Leftarrow \text{RCM}[p, n_{\text{post}}, :]$
5:     $\vec{h}_{pre} \Leftarrow_{\text{NC}} \text{HISTORY}[\vec{p}_{\text{pre}}, \vec{n}_{\text{pre}}]$
6:     $\Delta w \leftarrow \text{stdp}(\vec{h}_{\text{pre}}, h_{\text{post}})$
7:     $\text{ACC}[\vec{p}_{\text{pre}}, \vec{n}_{\text{pre}}, \vec{s}] \Leftarrow_{\text{NC}} \text{ACC}[\vec{p}_{\text{pre}}, \vec{n}_{\text{pre}}, \vec{s}] + \Delta w$
8: end for
See Appendix A for notation
```

D. Applying the accumulated statistics

While the STDP statistics are continuously accumulated, they may not necessarily need to be *applied* continuously. Instead, the synapse weights are updated through a separate kernel (Algorithm 4). This may be invoked at a fixed frequency, or may be invoked in response to specific events to reward certain behaviours.

Since the synapse weights are stored in a forward order (organised by presynaptic neuron), while the per-synapse potentiation/depression accumulator is stored in reverse order (organised by postsynaptic neuron), applying the accumulated statistics involves a form of matrix transposition.

The update of each weight takes into account both the existing weight of the synapse, as well as the global upper and lower weight limits to ensure that weights do not stray out of bounds or change sign (the function 'bound' in the listings).

V. Results

A. Benchmarks

For benchmarking we use a randomised toroidal network, parametrised by network size (p) and a connection locality

Algorithm 4 STDP application kernel

```
Input: $r$: STDP reward; ACC: STDP statistics accumulator;
    $p$: current partition
Output: Synapse weights in FCM updated to include accu-
    mulated statistics.
1: for all $n \in$ partition neurons do
2:     $\vec{\Delta w} \Leftarrow \text{ACC}[p, n, :]$
3:     $\text{ACC}[p, n, :] \Leftarrow \vec{0}$
4:     $\vec{s}_{\text{fcm}} \Leftarrow \text{RCM}[p, n, :]$
5:     $\vec{w}_t \Leftarrow_{\text{NC}} \text{FCM}[\vec{s}_{\text{fcm}}]$
6:     $\vec{w}_{t+1} \leftarrow \text{bound}(\vec{w}_{t+1} + \vec{\Delta w} \times r)$
7:     $\text{FCM}[\vec{s}_{\text{fcm}}] \Leftarrow_{\text{NC}} \vec{w}_{t+1}$
8: end for
See Appendix A for notation
```

parameter (σ). This is designed to highlight the scaling properties and the potential performance bottlenecks of the kernel, while operating under a biologically reasonably plausible regime. The network is a ring torus which is constructed from an integer number of patches consisting of 32×32 neurons evenly spaced on a grid. The torus diameters are thus 32 and $32p$, and the number of neurons in a network is $1024p$. The patch size by design corresponds exactly to the maximum partition size used in the kernel. The network is split into excitatory (80%) and inhibitory (20%) neurons. These are organised randomly on the grid. The neuron parameters, including for random input current, are chosen as in Izhikevich's example network in [6].

Each neuron has 1000 synaptic connections with other neurons. These connections are randomised such that the distribution of distances (2D Cartesian distance along the torus surface) between pairs of neurons follow a normal distribution as suggested by [13] (although we deal only with two dimensions and arbitrary distance units). For excitatory connections the distance is drawn from $\mathcal{N}(0, \sigma)$, the weight is uniformly random between 0.0 and +0.5, and the delay is a linear function of distance, with a maximum delay of 20 ms. For inhibitory connections the distance is drawn from $\mathcal{N}(0, 16)$, the weight is uniformly random between 0.0 and -1.0, and the delay is fixed at 1 ms. The resulting network fires at a mean rate of around 7.5Hz, regardless of size or locality parameters.

The benchmarks are executed both with and without STDP. When STDP is used, all the excitatory synapses are plastic, whereas the inhibitory synapses are static. The STDP function is $\alpha \exp(\Delta t / \tau)$. For pre-post pairs $\alpha = +1.0$, for post-pre pairs $\alpha = -0.8$, and $\tau = 20$ for both.

In the following experiments we map one patch to exactly one multiprocessor. We use network sizes ranging from 8 patches (8K neurons) to 30 patches (30K neurons). At the largest network size all the multiprocessors on the device are active in parallel, saturating the device. The network size can be scaled further (available memory permitting), but throughput will not increase much further. Instead, larger networks will lead to longer latencies. This setup allows us to investigate the scaling properties of the kernel, up to the computational and memory bandwidth saturation point of the device. In practice, the kernel should also deal with load balancing issues.

B. GPU implementation

We run benchmarks on the Nvidia Tesla C1060 GPU, based on the CUDA architecture. The C1060 contains 30 multiprocessors and 240 scalar processors, clocked at 1.3 GHz, totalling a peak single-precision floating point performance of 933 Gflops. The device has 4 GB of global memory, with a peak bandwidth of 102 GB/s. The user-managed local memory available within a multiprocessor is limited to 16 KB.

C. Throughput and speedup without STDP

For performance we are interested in maximising throughput and speedup. We measure throughput in terms of spike deliveries (not firings) per wall-clock second. This measure is somewhat insensitive to the average firing rate and the number of synapses per neuron. Figure 3 shows this throughput scaling well for different network sizes with different levels of locality, peaking at 551M spike deliveries per second.

One might expect the locality parameter to have some effect on performance, as less clustered networks are likely to have smaller synapse groups due to a wider range of targets. This in turn will lead to poorer memory bandwidth utilisation for the main memory operation, namely reading the synapse data. For the networks constructed using the above method there is some variation in performance for networks with different levels on locality. However for these networks, the effect of partially-filled synapse groups (which lowers performance for less local networks) is balanced by the effect of global atomic memory operation collisions in the scatter step (which lowers performance for more local networks). Overall the effect is only minor differences in performance between the networks as currently constructed, even though there are differences in the distribution of synapse group sizes (Figure 3, bottom).

To evaluate the effect of underutilisation of the device due to small groups, we simulate a network which by design is a worst case in terms of group sizing. The 'uniform' network is similar to the above torus networks, but both the target neurons and the delays are (indpendently) randomly chosen from a uniform distribution. As can be seen in Figure 3 this results in predominantly very small groups. The throughput of the simulation of this network is less than 200M spike deliveries per second. Furthermore, the memory bandwidth is already fully utilised for a small network of 8K neurons, so simulating larger networks has little effect on performance. Networks where there is any amount of clustering at the scale of thousands of neurons, or where delay is distance-dependent should result in higher throughput than this worst-case.

The speedup for the benchmarks which fully utilise the device falls between 2.2 and 2.5 times real-time (for 'uniform' it is 0.96). There is thus scope for increasing network size further and still use the simulation in an embodied setting.

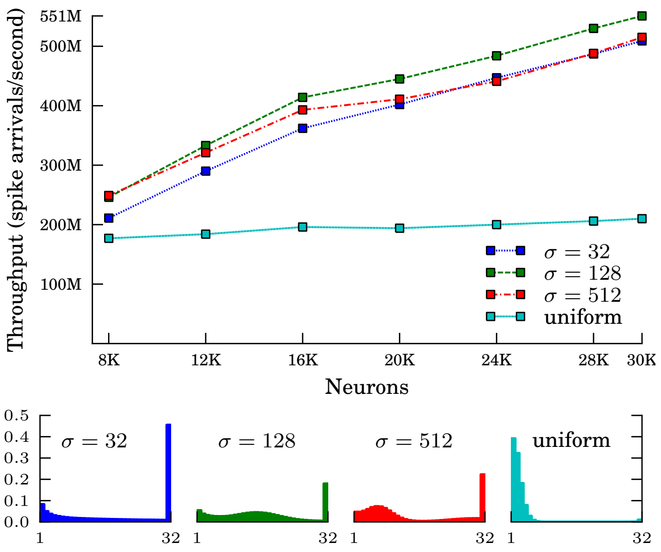

Fig. 3. Top: effect on throughput of scaling network size for different levels of locality in the network connections, for simulations with STDP disabled. Bottom: the distribution of synapse group sizes (between 1 and 32) for different network sizes and levels of locality. A larger proportion of "full" groups means fewer idle threads, and potentially better memory utilisation.

TABLE I
COST OF ACCUMULATING AND APPLYING STDP STATISTICS

Accumulation enabled	Application frequency	Throughput (spike arrivals/s)
✗	N/A	551M
✓	0	332M
✓	1Hz	308M
✓	10Hz	280M
✓	100Hz	150M

D. Performance when using STDP

The addition of STDP to the kernel incurs some overheads, as each neuron firing affects both its postsynaptic and presynaptic incident neurons. The *application* of the accumulated changes, incurs additional overheads on top of this. The current STDP application scheme (Algorithm 4) performs a costly full walk over all the synapses to determine which ones require updating. Table I shows the performance for a network with $p = 30$ and $\sigma = 128$ running with STDP accumulation both enabled and distabled, and varying STDP application frequencies. Our current application method is best suited to relatively infrequent application. If frequent or continous application is desired, a different scheme could avoid walking over the full set of synapses.

E. Comparison with alternative implementations

We evaluate the performance of our kernel compared with a CPU-based kernel, and GPU-SNN, the GPU-based kernel described in [9]. The results are summarised in Table II, for

TABLE II

COMPARISON OF SIMULATORS (NETWORKS OF 30K NEURONS)

Simulator	Platform	Cores	synapses/ neuron	Throughput
GPU (This work)	Tesla C1060	30	1000	510-550
GPU-SNN [9]	Tesla C1060	30	500	$\sim$121M
CPU	Xeon E5420	4	1000	$\sim$25M

networks with 30K neurons and 1000 or 500 synapses per neuron.

The CPU kernel uses similar data structures to the GPU kernel, but does not deliver spikes in fixed-size groups, since the memory system does not have the same coalescing constraints. The CPU kernel makes use of multi-core parallelism (using pthreads), and has had some memory-related optimisations applied. Specifically we have optimised memory access patterns by aligning data in the connectivity data structures to cache-line boundaries, and by setting thread affinity such that data need not migrate between the different cores' caches. We run the same benchmarks as in the throughput experiments above, on a 4-core 2.5GHz Xeon E5420 with 6MB of L2 cache. The CPU kernel is largely insensitive to variation in the locality parameter. The overall throughput is around 25M. Our kernel achieves up to 22 times this throughput.

For the GPU kernel described in [9] (GPU-SNN) we made use of the source code distributed with that paper. GPU-SNN limits the number of synapses per neuron to 550 and uses a different scheme for delivering random neuron input. We were therefore unable to recreate our benchmarks. To measure performance we made use of one the networks similar to the one described in [9], with 30K neurons and 500 synapses per neuron. The neuron parametrisation and the firing rate is similar to in our benchmarks. The time to compute random input current has been factored out from the reported results, so as not to penalise the GPU-SNN for using a more expensive method. For the current mix of benchmarks, our kernel achieves a throughput which is around 4.5 times higher, although a more thorough comparison with an updated version of GPU-SNN running the exact same network as us would provide a better comparision of their relative merits.

VI. CONCLUDING REMARKS

This paper has presented a GPU kernel which can simulate networks of a few tens of thousands of highly connected neurons in real time. This simulator improves on existing GPU-based simulators, by better exploiting the memory architecture on the GPU, and achieves up to 4.5 times higher throughput. The simulator provides a good basis on which to build a larger cluster-based simulator, which will allow us to simulate significantly larger networks. In other further work, we consider improving the mapping from neurons to processors to deal with less structured networks, and also changing the STDP implementation to use a more general trace-based approach [12], which would support a greater range of STDP protocols.

APPENDIX

A. Typographic Conventions in Algorithm Listings

The pseudocode in the algorithm listings distinguish between local memory ($italics$) and global memory (SMALL CAPS). The global memory data structures are described in the text. For local memory, 1D vectors are denoted like $\vec{x}$, and scalars like y. For accesses to multidimensional data in global memory, ':' indicates all indices along the relevant dimension. For global memory, coalesced operations are denoted by '$\Leftarrow$', and non-coalesced operations by '$\Leftarrow_{\text{NC}}$'. Operations on registers and local memory are denoted by '$\leftarrow$'. Some common variable names recur: p denotes a partition index, n a neuron index (within a partition), s a synapse index, g a group index, d a delay, and w a weight.

B. Software

The simulation software library, `libnemo`, is available as a C/C++ source library from the first author's website. It is published under a GPL v2 licence.

ACKNOWLEDGMENT

This work has been funded through EPSRC grant EP/F033516/1.

REFERENCES

[1] M. Shanahan, "Dynamical complexity in small-world networks of spiking neurons," *Physical Review E*, vol. 78, no. 4, pp. 041 924+, Oct 2008.

[2] D. B. Thomas and W. Luk, "FPGA accelerated simulation of biologically plausible spiking neural networks," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, April 5–7 2009.

[3] K. Cheung, S. R. Schultz, and P. H. W. Leong, "A parallel spiking neural network simulator," in *Proc. IEEE. Int. Conf Field-Programmable Technology*, 2009, pp. 247–254.

[4] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses," in *Proc. Conf. High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[5] E. Izhikevich and G. Edelman, "Large-scale model of mammalian thalamocortical systems." *Proc. Nat'l Academy Science USA*, 2008.

[6] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Trans. Neural Networks*, vol. 14, pp. 1569–1572, 2003.

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

[8] N. Caporale and Y. Dan, "Spike timing-dependent plasticity: a hebbian learning rule." *Annual review of neuroscience*, vol. 31, no. 1, pp. 25–46, February 2008.

[9] 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 Networks*, vol. 22, pp. 791–800, 2009.

[10] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk, "NeMo: A platform for neural modelling of spiking neurons using GPUs," in *Proc. IEEE Int. Conf Application-specific Systems, Architectures and Processors*, 2009, pp. 137–144.

[11] L. Howes and D. Thomas, *GPU Gems 3*. NVIDIA Corporation, 2007, ch. 37: Efficient Random Number Generation and Application Using CUDA, pp. 805–830.

[12] A. Morrison, M. Diesmann, and W. Gerstner, "Phenomenological models of synaptic plasticity based on spike timing," *Biol. Cybern.*, vol. 98, no. 6, pp. 459–478, June 2008.

[13] B. Hellwig, "A quantitative analysis of the local connectivity between pyramidal neurons in layers 2/3 of the rat visual cortex," *Biological Cybernetics*, vol. 82, no. 2, pp. 111–121, January 2000.