

NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs

Andreas K. Fidjeland*, Etienne B. Roesch*[†], Murray P. Shanahan*, Wayne Luk*

**Department of Computing, Imperial College London
London, United Kingdom*

Email: {akf,eroesch,mpsha,wl}@doc.ic.ac.uk

*[†]Swiss Centre for Affective Sciences, Department of Psychology
University of Geneva, Switzerland*

Abstract—Simulating spiking neural networks is of great interest to scientists wanting to model the functioning of the brain. However, large-scale models are expensive to simulate 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 NeMo, a platform for such simulations which achieves high performance through the use of highly parallel commodity hardware in the form of graphics processing units (GPUs). NeMo makes use of the Izhikevich neuron model which provides a range of realistic spiking dynamics while being computationally efficient. Our GPU kernel can deliver up to 400 million spikes per second. This corresponds to a real-time simulation of around 40 000 neurons under biologically plausible conditions with 1000 synapses per neuron and a mean firing rate of 10 Hz.

I. INTRODUCTION

Understanding the brain constitutes one of the greatest challenges of the century [19]. This long-term project mobilises researchers from many disciplines (e.g. computer science, neuroscience, psychology, philosophy), and requires the application of a variety of methodologies. The approach taken by computational neuroscience is to develop implementable models of brain functionality, and thereby to test existing theories and open new avenues for research. The complexity of the task, however, often demands a trade-off between biological plausibility and computational efficiency.

In addition, efforts aiming to produce embodied models in robots give rise to further technical obstacles, and bring out the need for real-time performance (see [1], [16]). In biologically faithful embodied models of cognition, large neural networks that can perform complex tasks in real-time have the potential to be more scientifically informative than small and computationally limited networks.

In this paper we present NeMo, a platform designed to respond to these challenges. NeMo allows modellers to simulate large-scale networks of biologically plausible spiking neurons, using the general-purpose computing power of graphics processing units (GPUs). More specifically, NeMo implements Izhikevich’s “simple model of spiking neurons” [10], which is capable of realistically modelling a wide range of spiking dynamics without sacrificing computational efficiency. The key contributions of this

work are:

- A method for simulating spiking neural networks with conduction delays on a parallel streaming processor (Section III). Novelty includes a just-in-time spike delivery in order to reduce memory bandwidth requirements while handling conduction delays, and a 2D grid of spike queues used to avoid synchronization problems.
- A prototype implementation of this novel method of simulation using the Tesla GPU platform (Section IV), exploiting an architecture with hundreds of processing cores.
- An evaluation of this simulation kernel using spiking networks with up to 32 000 neurons with a high rate of firing (Section V). The kernel can deliver nearly 400 million spikes per second.

II. BACKGROUND

The neuron is the basic computational unit in the brain. An adult human brain contains approximately 10^{11} neurons, and has approximately 10^{14} connections. When simulating a network of neurons, researchers face two issues. Firstly, available neuron models differ in level of details, and computational complexity, and as one would expect, realistic models are the most computationally demanding [5]. Secondly, the performance of the model is highly dependent on the structure and scalability of the network. The brain can be thought of as an intricate network of distinct structures [2]. To faithfully model even a subset of such structures — let alone an entire brain — is currently beyond our reach. To address these challenges, computational modellers aim to combine the most realistic, yet efficient model of neurons with the most powerful processing resources available [3], [15]. In this section, we explore available neuron models, and introduce Izhikevich’s spiking neuron model [10] as an appropriate candidate for simulation. This model realistically accounts for a wide range of spiking dynamics while being

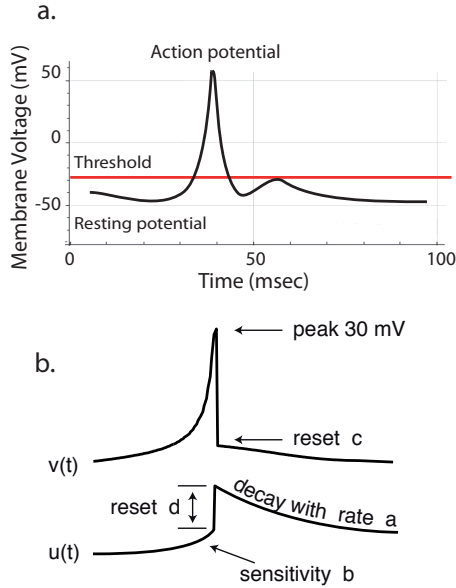


Figure 1. (a) Schematic action potential spreading from the soma of the neuron along the axon to neighbouring neurons. (b) Graphical representations of Izhikevich’s 2D system of differential equations over time t . (Reproduced from [10]. Figure and reproduction permissions are freely available at <http://www.izhikevich.com>.)

computationally efficient. We then introduce existing work on accelerating spiking neuron network simulation.

A. An ideal model of a neuron

Hodgkin and Huxley [7] proposed a thorough description of the neuron and its behaviour. Ion transport through the membrane of the neuron creates an electrical potential. Upon reaching a threshold, the integration of the incoming stimulation initiates a short current pulse (about 100 mV), the action potential (Figure 1a), which propagates from neuron to neuron through bridging synapses. Translating their observations into a set of nonlinear ordinary differential equations allows one to reproduce the evolution of the voltage across the membrane. However, while the Hodgkin-Huxley (HH) equations are regarded as accurate, using them to model more than a handful of interacting neurons in real time is presently unrealistic.

An alternative approach to HH exhaustive equations has been proposed by Izhikevich, who applied geometric bifurcation theory to identify a canonical model to cortical neurons’ spiking dynamics [9], [10]. The model consists of a two-dimensional system of ordinary differential equations (Figure 1b) of the form

$$v' = 0.04v^2 + 5v + 140 - u + I \quad (1)$$

$$u' = a(bv - u) \quad (2)$$

with an after-spike resetting

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

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 , and $' = d/dt$ where t is time. 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 produce the behaviour of different types of neurons. Typically these values are randomised around some default values, such as $a = 0.02$, $b = 0.2$, $c = -65$, and $d = 2$.

How the incoming current, I , is calculated based on incoming spikes is not directly specified in the model. In biological brains the spike delivery involves current flow along the axon of the presynaptic neuron, across a synapse, and along the dendrites in the postsynaptic neuron. In typical implementations, this is abstracted away, with connections represented by a single weight, (the current induced in the postsynaptic neuron), and a conduction delay, (the time between the presynaptic neuron fires and the spike arrives postsynaptic neuron).

In contrast to the HH equations, which aim at reproducing ionic currents, Izhikevich’s model is tailored to reproduce spiking responses. In particular, it faithfully reproduces several properties of biological spiking neurons spanning tonic spiking, bursting and chattering activity, commonly observed in the brain [11]. Along with simplified computation, these characteristics make it an ideal candidate for large-scale simulations, as shown by Izhikevich and Edelman [8].

B. Simulating neurons

Izhikevich and Edelman [8] report using a 60-node Beowulf cluster of 3 GHz processors, with 1.5 GB of RAM each, to simulate one second of activation for one million neurons in one minute. The simulation of one second of activation for 10^{11} neurons took 50 days — using a Beowulf cluster of 27 of such processors. Hence while Izhikevich and Edelman demonstrate the feasibility of the approach, managing real-time performance still remains an open issue.

Technology evidently plays a critical part in the performance, and modellers seek to employ the most powerful resources available. Researchers of the Blue-Brain project, for instance, partnered with IBM to faithfully model each biological component of a neuron using a Blue Gene/L supercomputer averaging 22.4 Tflops [14].

Other successes have made use of both full custom ASIC devices and FPGA-based reconfigurable architectures, e.g.

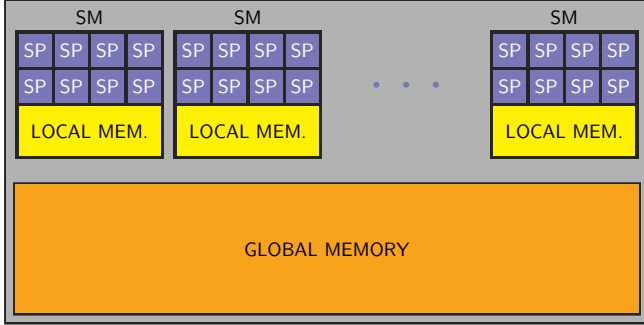


Figure 2. Generic GPU architecture with a collection of loosely coupled streaming multiprocessors (SM), each composed of parallel scalar processors (SP), and a two-level memory hierarchy.

[6], [15], [17]. Notable examples include, for instance, the SpiNNaker project that aims to build a universal platform to simulate large networks using a wide variety of spiking neuron models [3], [4], or solutions specifically tailored to implement Izhikevich’s model [21]. In contrast to customized hardware, FPGA-based architectures are more flexible and thus more adequate to modellers who need to explore wide parameter spaces. However, this flexibility has a cost in terms of resource requirements (memory bandwidth in particular), speed of execution and accuracy, which are dependent on the type of FPGA being used and the configuration intended. Furthermore, FPGA-based architectures remain only accessible to specialised research groups, and high-performance solutions can be expensive.

Implementations that take advantage of off-the-shelf resources — like Izhikevich and Edelman — are few. Recent advances in the field of general-purpose GPUs, however, offer an opportunity for high-performance systems [13]. Their architecture is ideal for simulating neural networks, if one can take advantage of the clustered parallel processing power available (Figure 2). A particular advantage of these GPU platforms is the large memory bandwidth available. GPUs are also more accessible than FPGAs, generally cheaper and require less set up, although they could be less power-efficient [22].

Our simulation platform, NeMo, makes use of GPUs, and aims at providing modellers with the ability to simulate biologically plausible spiking neurons in real time. It implements Izhikevich’s spiking neuron model, providing researchers with the flexibility required to model a wide range of spiking dynamics while ensuring processing efficiency.

III. SIMULATING SPIKING NEURONS ON STREAMING PROCESSORS

Given both the sheer size of biological neural networks and their parallel nature, it is clear that parallel computation is required for large-scale simulations. To that end, we consider a parallel architecture of the type shown in Figure 2, with a large number of processing cores grouped into distinct

multiprocessors residing on the same device and sharing access to a global memory space.

Simulating a network on a device consisting of several multiprocessors necessitates dividing the network into separate partitions. Each partition contains a set of neurons and the set of *local* synapses which only involve these neurons. The whole network contains a set of partitions, as well as the *global* synapses that straddle partition boundaries. This two-level topology of this partitioned network reflects the topology of the hardware architecture.

The amount of local memory available to each multiprocessor limits the size of such partitions. Smaller partitions result in faster computation, given that the current state of the partition can be stored locally. Such a set up may lead to hyper-real-time performance. However, given the fairly relaxed (with respect to the processor speeds) temporal constraints of embodiment in a robot for instance, going significantly beyond real-time at the expense of network size is wasteful. It is thus desirable to have a larger number of neurons rather than faster simulation, and consequently a single multiprocessor needs to process multiple partitions in sequence.

A. Neuron update and spike delivery

A discrete-time simulation of a spiking neural network consists of a repetition of two steps: neuron update and spike delivery.

For each simulation cycle, the neuron update step involves all neurons, with their associated parameters and state variables. The computational cost of the update step depends on the complexity of the neuron model, which in the case of Izhikevich’s model is quite low. When scaling a network to larger sizes, the cost of the neuron update step scales linearly with the number of neurons. This is generally true even if the network is scaled to cover multiple devices, as the neuron state and parameters can be stored locally.

The spike delivery step involves all synapses of all firing neurons. Typically, only a small proportion of neurons fire in a given cycle, but the number of synapses associated with each neuron can be quite high (about 10^3 is normal). For each postsynaptic neuron the incoming spikes are then integrated, for example using a weighted sum. Ignoring the underlying platform, this part of the simulation scales by $O(nfm)$, where n is the number of neurons, f is the average firing rate in the network, and m is the average number of synapses per neuron. When scaling a network above the capacity of a single processing element, however, the cost of each spike delivery increases as synapses between neurons residing on separate devices become increasingly prevalent. As the maximum distance within the system increases, spike delivery has to go via increasingly costly routes: from local memory, global memory, and system bus to the network. In this work we only consider spike delivery within a single device (via the memory system), albeit one with multiple

processing units and a multi-level memory hierarchy. In any event, it is clear that the main challenge for a parallel simulation of spiking neurons using Izhikevich’s model is in the spike delivery rather than in the neuron update.

B. Dealing with delays

The presence of conduction delays further complicates the process of spike delivery. Some simple models may not use delays at all, but rather deliver spikes in the time step after they are generated. Support for delays is a useful addition to a simulator, however, as it adds interesting behaviors to the system [12]. The presence of conduction delays means that each spike is “in flight” for some time before delivery. On a sequential implementation this can be dealt with by maintaining a global queue of spikes to be delivered. On a parallel architecture such as the one in Figure 2, however, maintaining such a single queue is undesirable for two reasons. Firstly, it requires synchronization when updating the queue, in order to avoid race conditions. Secondly, each spike delivery via the queue is costly in terms of memory bandwidth, as both the synaptic weight and target address need to first be written by the source and then subsequently read by the target.

We deal with the problem of synchronization by maintaining a 2D grid of spike queues, with one separate queue for each pair of source-target partitions. The synchronization involved in a queue update is then purely local, involving only processing units dealing with the same partition. This form of synchronization is significantly cheaper than a global synchronization.

We deal with the problem of memory bandwidth by delaying spike delivery to the cycle before it is required. This just-in-time delivery is achieved by splitting the connectivity matrix according to the conduction delay, and by keeping track of both each neuron’s recent firing history and its associated conduction delays. If doing this for spikes delivered via the 2D queue grid, each queue can be reduced to a double buffer large enough to hold one cycle’s worth of firing between partitions. The current implementation uses separate delay slots since there is enough memory for the networks we are running, and memory bandwidth usage, which is more important, is not affected. The main saving of this just-in-time delivery, however, is in dealing with spike delivery *within* a single partition. In this case the synapse data associated with a spike only needs to be loaded once, and does not need a round-trip to memory before being used at the target neuron, which now resides on the same multiprocessor.

C. Clustering

The spike delivery method introduced in the previous section yields lower costs for spike delivery within the same partition compared with spike delivery between different partitions. This is an interesting feature for the simulation

of biologically plausible artificial neural networks given the intrinsic structure of processing networks in the brain, which exhibit a high level of clustering. What is known as the amygdala, for instance, refers to a differentiated, heterogeneous region of the brain near the temporal pole, implicated in the orienting of attention to emotion-laden information in the environment [18]. Histologists describing the anatomy of this thumb-sized structure see six to nine nuclei, heavily interconnected and together forming a hub reaching out to several regions of the brain [20].

IV. GPU IMPLEMENTATION

Modern GPUs provide a highly parallel architecture of the type illustrated in Figure 2. A large number of simple parallel processing elements provide a high floating point performance, while a memory system designed for heavy graphics usage provides the bandwidth to match. We provide a computational kernel which maps a neural network onto such devices and exploits the architecture to achieve high performance simulation.

A. Architecture overview

We make use of the Nvidia Tesla C1060 GPU, based on the CUDA architecture. The device is divided into multiple single-instruction multiple-thread (SIMT) *streaming multiprocessors* (SMs), each of which consists of eight thread execution units, *scalar processors* (SP) (Figure 2). 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.

A kernel executes a large number of threads divided into *thread blocks*. Thread blocks execute independently and are scheduled onto the available SMs, either in parallel or in sequence, depending on the number of thread blocks and the number of SMs. Threads within a block execute concurrently on a single SM, where they can cooperate using shared memory and barrier synchronisation. Thread blocks are further divided into *warps*, within which threads execute in lock-step. The number of threads executing on an SM is generally larger than the number of SPs, typically at least 128, but thread-switching is cheap due to the presence of per-thread registers.

The device global memory is shared among all the SMs. It is high-capacity (4 GB) and high-bandwidth (102 GB/s), but also high-latency (approx. 600 cycles). Using a large number of threads to hide this latency is paramount to achieve high performance. Also, in order to make the best use of the available memory bandwidth, accesses should be organised such that threads belonging to the same warp access consecutive memory addresses (a *coalesced* access).

The threads running on a single SM share access to *local memory*. This is a small (16 KB in 16 banks) but fast memory, which acts as a user-managed buffer/cache. Effective use of this memory is also an important factor

in achieving high performance. The memory provides very high bandwidth as well; as long as no bank conflicts occur, each active thread in a warp can read one word per cycle.

B. Simulation data

Static simulation data reside in global memory and are stored on a per-partition basis. Each partition is associated with a set of floating-point vectors for the neuron parameters ($\vec{a}, \vec{b}, \vec{c}, \vec{d}$). Additionally, there are two connectivity matrices per network partition, one for local connectivity (within partitions, C_{local}), and one for global connectivity (between partitions, C_{global}). These are stored in a sparse form, with a list of synapses stored on a per-neuron (presynaptic) and per-delay basis (Figure 3). Synapse data consist of a weight and the index of the postsynaptic neuron. The delay is implicit in the connectivity matrix data structure. In our implementation the weights are single-precision floating point. The postsynaptic neuron index must identify the neuron *within* a partition (labelled N in the figure), and for global connectivity also the index of the target partition (P). In either case 18 bits are used to address the postsynaptic neuron. This is padded out to four bytes, giving a total of eight bytes of storage per synapse. The padding in each synapse will later be used when implementing synaptic plasticity. The connectivity matrices for each partition are complemented by firing delay vectors, \vec{f}_{delay} . For each neuron this is a bit-vector specifying at what delays that neuron has any outgoing synapses. We use four bytes for this, supporting up to 32 millisecond delays.

The dynamic state consists of the neuron state vectors (\vec{u} and \vec{v}), firing history, and the spike queues. The recent firing history for each neuron (\vec{f}_{recent}) is a per-neuron bit-vector of four bytes, with bits set to indicate at what cycles in the recent past the neuron fired. Each source-target partition pair has an associated spike queue. The size of each queue is set based on the maximum number of synapses crossing a source-target partition boundary.

In addition to the above per-neuron data, a number of per-partition parameters are stored, for example for the pitch of the connectivity matrices. Using such per-partition configuration, neurons sharing the same property, such as a similar connectivity matrix pitch, can be mapped to the same thread block, reducing the number of threads running idle. The partition parameters are stored in a special cache for constants provided by the architecture.

Some of this data is buffered in shared memory, including firing data and accumulated current. This limits the partition size to 1K neurons.

C. Local simulation step

The execution of a single simulation cycle can be split into a *local simulation step*, dealing with local spikes (within a partition) along with the neuron update, and a *global simulation step* dealing with global spike delivery

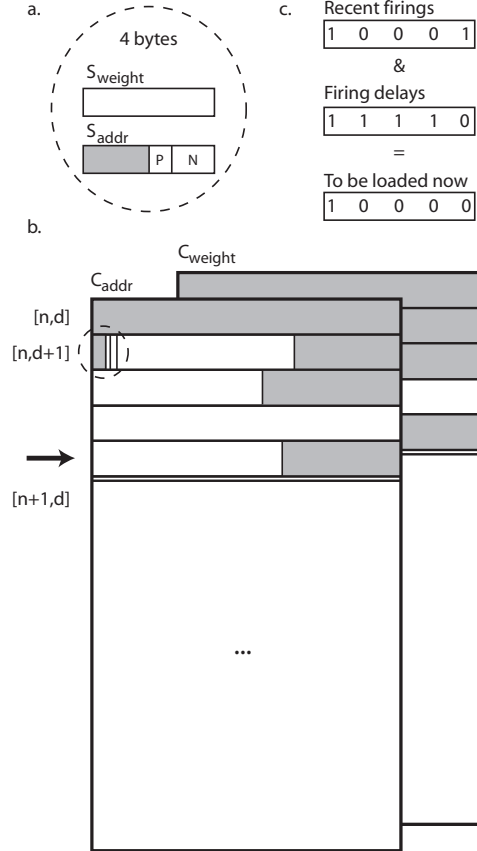


Figure 3. Organisation of connectivity data. (a) Synapses are represented using a synaptic weight as well as the address of the postsynaptic neuron. (b) The synapse data is organised according to presynaptic neuron and delay. Grey area represents padding or unused memory space. (c) The recent firing history, along with a bit-vector indicating non-empty rows in the connectivity matrix specify what rows contain spikes due to be delivered now – in our example, shown by the arrow.

(between partitions). Figure 4 shows an overview of the local simulation step. This can be further divided into three stages: determining what local spikes need to be delivered (lines 1–4), accumulating the postsynaptic current for those spikes (lines 5–8), and updating the neuron state (lines 9–10).

Determining what local spikes need to be delivered (lines 1–4) requires loading the recent firing data, \vec{f}_{recent} , and the firing delay data, \vec{f}_{delay} . A bitwise ‘and’ of each pair of entries will yield a new vector where each entry is non-zero for neurons which have at least some spikes arriving at this cycle, and whose bits indicate how long these spikes have been in flight. Each bit indicates a row (indexed by presynaptic neuron and delay) in the local connectivity matrix that needs to be processed (Figure 3c). This vector is reduced in parallel using shared memory atomics (in ‘arriving’, line 4) to a compact list of neurons and the relevant delays, \vec{f}_{load} .

Accumulating the postsynaptic current resulting from the incoming spikes (lines 5–8) is done by iterating over the

```

1  $\vec{f}_{recent} \leftarrow \vec{F}_{recent}$  (n)
2  $\vec{f}_{delay} \leftarrow \vec{F}_{delay}$  (n)
3  $\vec{i} \leftarrow \vec{0}$  (n)
4  $\vec{f}_{load} \leftarrow \text{arriving}(\vec{f}_{recent}, \vec{f}_{delay})$  (n)
5 foreach  $n \in \vec{f}_{load}$ 
6   foreach  $d \in \text{delays}(n)$ 
7      $(\vec{s}_{weight}, \vec{s}_{addr}) \leftarrow C_{local}[n, d]$  (s)
8      $\vec{i}[\vec{s}_{addr}] \leftarrow \vec{i}[\vec{s}_{addr}] + \vec{s}_{weight}$  (s)
9  $(\vec{U}, \vec{V}, \vec{f}_{new}) \leftarrow \text{update}(\vec{U}, \vec{V}, \vec{i}, \vec{A}, \vec{B}, \vec{C}, \vec{D})$  (n)
10  $\vec{F}_{recent} \leftarrow (\vec{f}_{recent} \ll 1) \& \vec{f}_{new}$  (n)

```

Figure 4. Main steps of local simulation step, performing local spike delivery and neuron update. For variables, upper-case indicates global memory storage, while lower-case indicates local memory storage. The steps are either parallel by neuron (n) or by synapse (s).

presynaptic neurons whose spikes are arriving, and then over the delays associated with those spikes. The incoming current is accumulated in a vector (initialised to zero) with one entry per neuron using the synaptic weights which are loaded in parallel for each neuron/delay pair. Both these loops can be unrolled if the number of threads is greater than the pitch of the local connectivity matrix C_{local} . A potential race condition in updating current for the same postsynaptic neuron from two threads is handled by separating writes from different warps with a barrier synchronisation. Potential race conditions *within* a warp can be handled off-line when constructing the connectivity matrix.

Once all the accumulated current has been computed, the neuron state can be updated (line 9) according to equations 1, 2, and 3 (Section II). The updated firing data are the bit-shifted version of the old data with the bit corresponding to zero delay set depending on whether the neuron just fired (line 10).

D. Global simulation step

The local simulation step is wrapped in a global simulation step that deals with incoming spikes from global connections (before the local step) and distributes outgoing spikes (after the local step). Since the global spike delivery involves partitions executing either on different multiprocessors or on the same multiprocessor at different times, the delivery is managed via global memory using a separate queue for each source-target partition pair. Each simulation cycle is a separate kernel invocation, which ensures the coherence of the global memory.

To store the spikes that are in flight, we use a separate queue for each source-target partition pair. Within each such queue there are separate slots for each delay. The source partition can update several of these slots when distributing spikes, but the target only needs to read one slot per simulation cycle.

Writing spikes to the spike queue involves first reading a

block of synapses from C_{global} in a coalesced read for each firing neuron, and then writing each synapse to the relevant spike queue entry. This second step results in a number of scattered writes, as synapses with the same presynaptic neuron may target multiple partitions with varying delays, and hence are delivered via different queue slots.

Spikes resulting from recent firings in other partitions are integrated at the beginning of each kernel invocation. As there is a separate queue for each source-target partition pair, each partition has to check one queue slot for each *other* partition to determine if any spikes need to be delivered in the present cycle. Many of these reads can be issued concurrently.

The entries in the global spike queues contain the data required for the current accumulation, namely synaptic weight and postsynaptic index. The current vector in local memory, \vec{i} , can thus be updated in parallel once a block of data from the queue has been loaded.

The cost of delivering a single global spike is thus: the read of eight bytes as part of a coalesced read in the source partition, the write of the same eight bytes as a partially coalesced write in the source partition, and the read of eight bytes as part of a coalesced read in the target partition, for a total of 24 bytes of memory traffic.

V. PERFORMANCE

A. Experiments

We run experiments using two scalable networks, displaying different levels of clustering. The first one, coined *local*, contains a number of highly connected clusters with no connections between them. The size of each cluster is 1K neurons, matching exactly the partition size. Each cluster contains 80% excitatory neurons and 20% inhibitory neurons, each with 100 synapses that are randomly connected within the clusters, and which have randomised synaptic weights. Both the neuron parameters and synaptic weights are taken from the reference network in [10]. When scaling the network size we simply add new clusters.

The second network, coined *uniform*, uses the same neuron parameters as *local*, including number of synapses per neurons and distribution between excitatory and inhibitory neurons. This network, however, exhibits no clustering. Again we use 100 synapses which are randomly connected, but this time within the whole network rather than within a cluster. When mapping this to the GPU, each partition will thus have synapses requiring global spike delivery. The proportion of global to local synapses increases with the network size. Both the networks are stimulated with a few initial spikes, after which the network firing sustains itself.

These two networks are simulated on the Tesla C1060 GPU. Additionally we developed a reference implementation in Matlab (running on an Intel Xeon E5420 processor). We use this simulation to explore the scaling properties of a CPU implementation. This reference implementation

is significantly slower than the GPU and does not in itself provide a strong indication of what this CPU is capable of. However, we expect an optimised simulation kernel for the CPU to display similar scaling properties.

B. Scaling and throughput

Figure 5 shows the throughput of both the CPU and GPU simulations for different network sizes. The results are normalised to the throughput for 1K neurons, as we are interested in the trends rather than a direct comparison between the two.

For the CPU simulation the throughput is constant or decreases somewhat. This is expected on a sequential implementation, where the computational capacity is already being fully utilized. Some decrease in throughput when increasing network size may be a consequence of caches filling up.

For the GPU, the smaller networks under-utilize the device, as only one or a few multiprocessors are used. This explains the increase in throughput for *local*, as the network size is increased. For *uniform*, the throughput decreases somewhat when going from 1K to 2K neurons. For 1K neurons all the synapses are local. For 2K, however, half the synapses are global, and for larger networks an even larger proportion are. Delivering the spikes associated with these global synapses is more costly compared with local spikes, both because each spike requires an additional round-trip via global memory, and because these writes are more scattered, which incur a bandwidth penalty. This additional cost decreases the throughput, and this is only compensated for in larger networks.

The increase in throughput decreases somewhat between 16K and 32K. The C1060 contains 30 multiprocessors. For a network of up to 30K neurons no partition needs to share a multiprocessor. For 32K neurons, however, four partitions end up sharing multiprocessors, which explains the shape of the graph at this point.

In absolute terms the GPU processes up to 397 million spike deliveries per second of real time for the *local* network, and 112 million for the *uniform* network. For comparison, [13] achieves around 80 million spike deliveries per second (10 000 neurons with 100 synapses per neuron), using the Nvidia GTX280. This device is similar to the C1060, but provides around 40% more memory bandwidth. However, in contrast to our present kernel, they support spike plasticity, which should roughly double the computational cost.

C. Theoretical limits to performance

Ultimately, the simulation of spiking neural networks on this platform is limited by the memory bandwidth available for spike delivery. The C1060 specification reports a global memory bandwidth of 102 GB/s. This bandwidth is only attainable under highly idealized memory access patterns,

which will not be found in real applications. It does, however, provide us with an upper limit of performance.

Consider that a local spike delivery requires 8 bytes of memory traffic, while a global spike delivery requires 24 bytes. In a highly clustered network where 90% of synapses are local, an average spike requires 9.6 bytes of memory traffic. Under idealized conditions this memory could thus deal with 10 billion spikes per second, corresponding to one million neurons, with 1000 synapses each, firing at 10 Hz. A network with little or no clustering would be dominated by global synapses, and would thus average nearly 24 bytes per synapse. For this network the upper limit is around 4 billion spike deliveries per second, corresponding to 400 000 neurons, with 1000 synapses each, firing at 10 Hz.

Our kernel currently falls short of these upper limits. We therefore hope to further increase the performance of our kernel by improving the memory access patterns.

VI. CONCLUSION

We have presented NeMo, a novel platform for biologically plausible neural modeling. This platform targets a generic parallel streaming processing device of the type found in GPUs, which are available off-the-shelf. We thus hope to enable computational neuroscientists to make full use of low-cost accelerated hardware to simulate large-scale networks in real-time.

In future work we consider making use of multiple GPUs in a cluster of processors, introducing additional challenges with spike delivery over the system bus or the network. Additionally, replacing the Matlab implementation with an optimised CPU-based simulation is required to better assess the capability of the GPU-based kernel. The effectiveness of the simulator would also be enhanced by adding support for dynamically changing synapses, e.g. synaptic spike-time dependent plasticity.

ACKNOWLEDGMENT

This work has been funded through EPSRC grant EP/F033516/1 (to MS and WL), and a Research Fellowship by the Swiss National Science Foundation (to ER). The authors would like to thank Dustin Connor for useful discussions throughout this work.

REFERENCES

- [1] R. A. Brooks and L. A. Stein. Building brains for bodies. *Autonomous Robots*, 1:7–25, 1994.
- [2] R. J. Dolan. Neuroimaging of cognition: past, present, and future. *Neuron*, 60(6):496–502, 2008.
- [3] S. Furber and S. Temple. Neural systems engineering. *J. Royal Society: Interface*, 4:193–206, 2007.
- [4] S. Furber, S. Temple, and A. D. Brown. High performance computing for systems of spiking neurons. In *Proc. AISB 2006, workshop on GC5: architecture of brain and mind*, volume 105, pages 29–36, 2006.

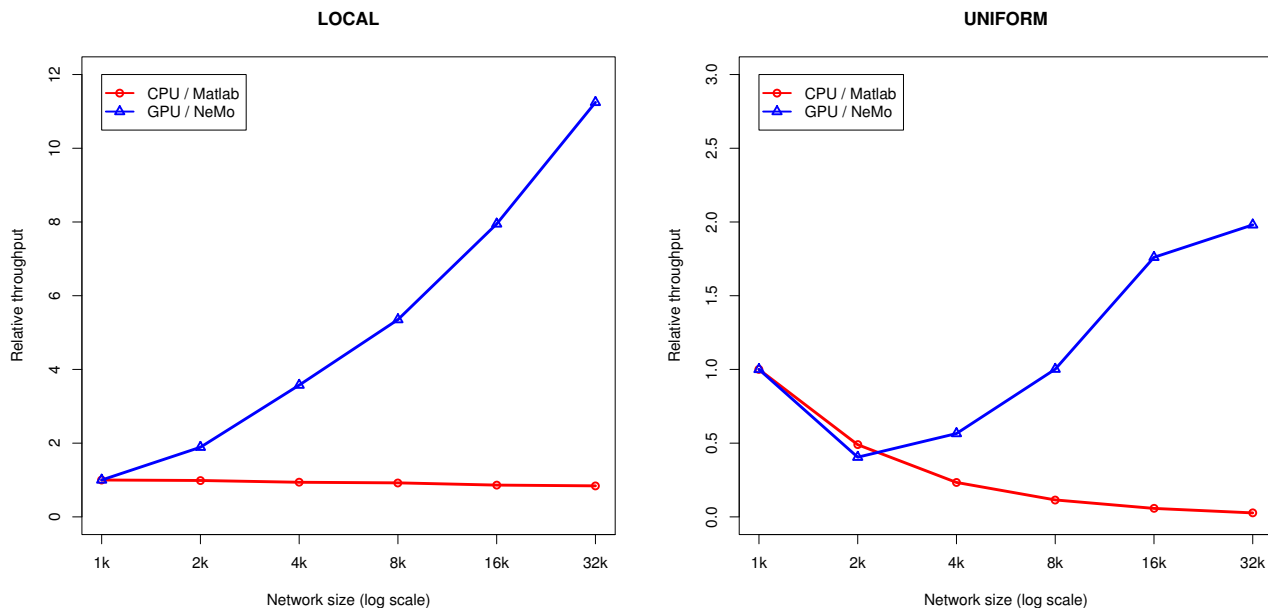


Figure 5. Simulation throughput for two different networks: *local*, which is highly clustered, and *uniform*, which is not.

- [5] W. Gerstner and W. M. Kistler. *Spiking neuron models: single neurons, populations, plasticity*. Cambridge University Press, Cambridge, U.K., 2002.
- [6] H. H. Hellmich, M. Geike, P. Griep, P. Mahr, M. Rafanelli, and H. Klar. Emulation engine for spiking neurons and adaptive synaptics weights. In *Proc. Int'l joint Conf. Neural Networks (IJCNN)*, pages 3261–3266, 2005.
- [7] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiology*, 117(4):500–544, 1952.
- [8] E. Izhikevich and G. Edelman. Large-scale model of mammalian thalamocortical systems. *Proc. Nat'l Academy Science USA*, 2008.
- [9] E. M. Izhikevich. Neural excitability, spiking and bursting. *Int. J. Bifurcation and Chaos*, 10(6):1171–1266, 2000.
- [10] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Trans. Neural Networks*, 14:1569–1572, 2003.
- [11] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Trans. Neural Networks (Special issue on temporal coding)*, 15:1063–1070, 2004.
- [12] E. M. Izhikevich. Polychronization: computation with spikes. *Neural Computation*, 18:245–282, 2006.
- [13] M. N. Jayram, N. Dutt, J. L. Krichmar, A. Nicolau, and A. Veidenbaum. Efficient simulation of large-scale spiking neural networks CUDA graphics processors. In *Proc. Int'l joint Conf. Neural Networks (IJCNN)*, 2009.
- [14] H. Markram. The blue brain project. *Nature Reviews: Neuroscience*, 7(2):153–160, 2006.
- [15] A. R. Ormondi and J. C. Rajapakse. *FPGA implementations of neural networks*. Springer, Dordrecht, The Netherlands, 2006.
- [16] R. Pfeifer, J. Bongard, and S. Grand. *How the body shapes the way we think: a new view of intelligence*. MIT Press, Cambridge, Mass., 2007.
- [17] S. J. Prange and H. Klar. Cascadable digital emulator IC for 16 biological neurons. In *Proc. Int. Solid State Circuits Conf. (ISSCC)*, volume 294, pages 234–235, 1993.
- [18] E. B. Roesch, D. Sander, and K. R. Scherer. The link between temporal attention and emotion: A playground for psychology, neuroscience, and plausible artificial neural networks. In J. Marques de Sá, editor, *Int. Conf. Artificial Neural Networks (ICANN)*, volume 2 of *LNCS*, pages 859–868. Springer-Verlag, 2007.
- [19] A. Sloman. GC5: The architecture of brain and mind. In C. A. R. Hoare and R. Milner, editors, *UKCRC grand challenges in computing research*, pages 21–24. British Computer Society, Edinburgh, UK, 2004.
- [20] L. Swanson. The amygdala and its place in the cerebral hemisphere. *Annals of the New York Academy of Sciences*, 985:174–184, 2003.
- [21] 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 2009.
- [22] A. H. T. Tse, D. B. Thomas, and W. Luk. Accelerating quadrature methods for option valuation. In *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, April 5–7 2009.