

# Streaming Parallel GPU Acceleration of Large-Scale filter-based Spiking Neural Networks

Leszek Ślaziński<sup>1</sup>, Sander Bohte<sup>1</sup>

<sup>1</sup>Department of Life Sciences, Centrum Wiskunde & Informatica,  
Science Park 123, NL-1098XG Amsterdam, NL

{leszek,sbohte}@cwi.nl

## Abstract

The arrival of graphics processing (GPU) cards suitable for massively parallel computing promises affordable large-scale neural network simulation previously only available at supercomputing facilities. While the raw numbers suggest that GPUs may outperform CPUs by at least an order of magnitude, the challenge is to develop fine-grained parallel algorithms to fully exploit the particulars of GPUs. Computation in a neural network is inherently parallel and thus a natural match for GPU architectures: given inputs, the internal state for each neuron can be updated in parallel. We show that for filter-based spiking neurons, like the Spike Response Model, the additive nature of membrane potential dynamics enables additional update parallelism. This also reduces the accumulation of numerical errors when using single precision computation, the native precision of GPUs. We further show that optimizing simulation algorithms and data structures to the GPU's architecture has a large pay-off: for example, matching iterative neural updating to the memory architecture of the GPU speeds up this simulation step by a factor of three to five. With such optimizations, we can simulate in better-than-realtime plausible spiking neural networks of up to 50,000 neurons, processing over 35 million spiking events per second.

## 1 Introduction

A central scientific question in neuroscience is understanding how roughly 100 billion neurons wired together through hundreds of trillions of connections jointly generate intelligent behavior. As psychologists, biologists and computer scientists try to capture and abstract the essential computations that these neurons carry out, it is increasingly clear that each individual neuron contributes to the joint computation. At the same time, canonical computational structures in the brain, like cortical columns and hypercolumns, comprise of tens to hundreds of thousands of neurons (Oberlaender et al., 2011). Simulating neural computation in many millions of neurons on very large computers is a great challenge, and the subject of considerable high-profile effort (Markram, 2006).

Still, the scientific workflow tremendously benefits from “table-top” neural modeling, both for testing ideas and for rapid prototyping. The arrival of affordable graphics processing cards suitable for massively parallel computing - General-Purpose Graphics Processing Units, GPGPU-computing - make it potentially possible to scale “table-top” neural network modeling closer to the network sizes of canonical brain structures. At such a scale, neural network modeling can simulate sizable parts of large-scale dynamical systems like the brain’s visual system, and enable research into the computational paradigms of canonical brain circuits as well as real-time applications of neural models in for example robotics.

State-of-the-art GPGPU architectures have evolved from Graphical Processing Units (GPU), where the parallel computations inherent to three-dimensional graphic processing were expanded to allow for the processing of more general computational tasks (Owens et al., 2008). GPGPU architectures are characterized by a many-core streaming architecture, with memory-bandwidth and computational resources exceeding that of current state-of-the-art CPUs by at least an order of magnitude. To fully exploit these resources however, algorithms have to be parallelized such as to fit the peculiarities of many-core streaming architectures.

A number of efforts have focused on developing efficient GPU-based simulation algorithms for specific differential-based spiking neuron models like the Hodgkin-Huxley model (Lazar and Zhou, 2012; Mutch et al., 2010), or variants of integrate-and-fire spiking neurons, like the Izhikevich model (Brette and Goodman, 2011; Fidjeland et al., 2009; Fidjeland and Shanahan, 2010; Han and Taha, 2010a,b; Krishnamani and Venkittaraman, 2010; Nageswaran et al., 2008; Vekterli, 2009; Yudanov, 2009; Yudanov et al., 2010). The advantage of differential-based spiking neuron models is that the parameters describing the neural state tend to be few, and evolving neural dynamics can be computed from the differential equations.

Here, we consider GPU-acceleration for filter-based spiking neuron models. Filter-based models approximate integrated versions of differential-based spiking neuron models, expressing the neural state as a superposition of spike-triggered filters. The most prominent example of a filter-based spiking neuron model is the Spike Response Model (Gerstner and Kistler, 2002). Filter-based spiking neuron models offer a different balance between computation and memory usage for spiking neuron simulation, and allow for parallel updates of the neural dynamics. As a neuron’s state in a filter-based formulation is defined as a weighted sum of filters, there is the additional benefit that in formulation, given the input spikes, the numerical error in the membrane potential does not accumulate over time like in models based on differential equations (Brette et al., 2007; Yudanov, 2009), thus making it more suitable for single precision computation, the native precision of GPUs.

In this paper, we present in detail a number of data structures and algorithms for efficiently implementing spiking neural networks comprised of filter-based spiking neuron models on state-of-the-art GPGPU architectures. The algorithms are implemented in OpenCL, and performance is measured for two modern GPUs: an NVidia GeForce GTX 470 and an AMD Radeon HD 7950. With GPU specific optimizations, we demonstrate real-time simulation for filter-based spiking networks of up to 50,000 neu-

rons with sparse connectivity and neural activity on the AMD Radeon GPU; the GPU then processes some 35-40 million spiking events per second. With a higher degree of connectivity and higher network activity, the GPU is able to process up to 600 million spiking events per second, at roughly one third realtime performance.

Network simulation comprises of three major simulation steps: updating neural states, determining which neurons generate spikes, and distribution of these spikes to connected neurons. For asymptotically large neural networks, spike distribution always dominates the simulation complexity (Brette et al., 2007). However, without optimization, we find that the operation of updating filter-based neurons to account for the future influence of current spike-events dominates the running time for network sizes up to several tens of thousands of neurons. Therefore, we optimize the neural updating step to maximize both parallelism and tune memory access patterns to the GPU architecture. We show that doing so speeds up neural updating by about a factor of four. After this optimization, spike-distribution becomes the dominant computation for very large networks.

Spike generation and distribution require the efficient collection of data from a subset of all neurons, a typical list structure operation. We implement efficient parallel list data structures, which greatly improves performance, and then further tune these algorithms for specific GPU characteristics, increasing performance of the data structure by another factor of three for typical problem sizes.

This paper is setup as follows: first, in Section 2, we introduce filter-based spiking neuron models for neural computation, and in Section 3 we introduce the general concepts for parallel computing on GPUs. In Section 4, we describe the principal phasing of filter-based spiking neural network simulation, and we analyze the complexity of simulating such networks. In Section 5, we describe the principal data structures for filter-based spiking neural network simulation, and some GPU-specific optimizations. In Section 6, we describe GPU-specific parallelized algorithms for simulating filter-based spiking neural networks. In Section 7, we simulate large scale filter-based spiking neural networks, and quantify the improvement derived from the algorithmic optimizations. We discuss some technical pitfalls in GPU-computing in Section 8, and in Section 9, we discuss our contributions in the context of large-scale neural simulation.

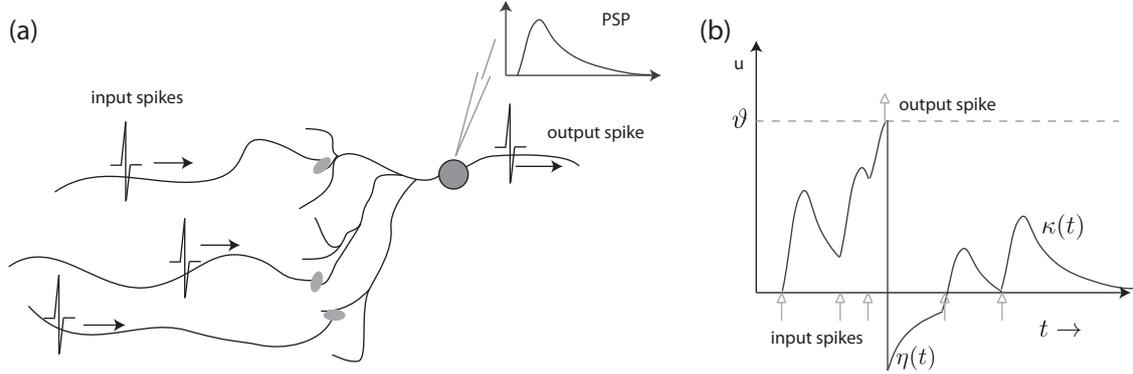


Figure 1: (a) Spiking neurons communicate through brief electrical pulses – action potentials or *spikes*. (b) Illustration of the main characteristics of the Spike Response Model, after (Gerstner and Kistler, 2002). The membrane potential  $u(t)$  is computed as a sum of superimposed post-synaptic potentials (PSPs) associated with impinging spikes, where the time evolution of each PSP is determined by a characteristic filter:  $\kappa(t)$ . The neuron discharges an action potential as a function of the distance between the membrane potential and a threshold  $\vartheta$ ; the discharge of an action potential contributes a (negative) filter  $\eta(t)$ .

## 2 Filter-based Spiking Neuron Models

Spiking neuron models describe how a neuron’s membrane potential responds to the combination of impinging input spikes and generated outgoing spikes (Figure 1a). In most standard models, like the Hodgkin-Huxley model (Hodgkin and Huxley, 1952), the voltage dynamics are defined as differential equations over the membrane potential dynamics and associated variables like voltage-gated channel proteins. Filter-based models integrate approximations of these differential equations to obtain formulations where the neuron’s membrane potential is expressed as a superposition of filtered input-currents and refractory responses, centered respectively on input and output spikes (Figure 1b). Output spikes are generated either deterministically, as the potential exceeds a threshold, or stochastically as a function of the membrane potential. The most well-known of such formulations is the Spike Response Model (Gerstner and Kistler, 2002), which, in various adaptations, closely fits real neural behavior (Brette and Gerstner, 2005; Jolivet et al., 2006).

**Model.** Formally, for a neuron  $i$  receiving input spikes  $\{t_j\}$  from neurons  $j$ , the membrane potential  $u_i(t)$  in the SRM is computed as a superposition of weighted input filters  $\kappa(t)$  and refractory responses  $\eta(t)$ :

$$u_i(t) = \sum_j w_j \sum_{\{t_j\}} \kappa(t - t_j) - \sum_{\{t_i\}} \eta(t - t_i) + u_r, \quad (1)$$

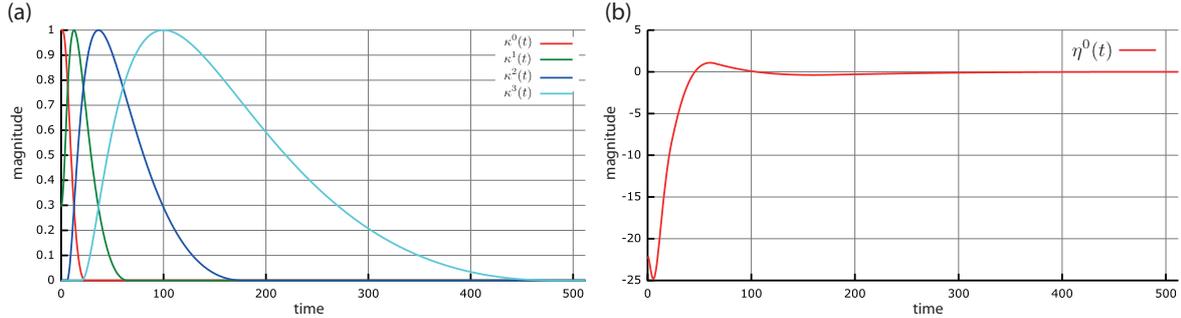


Figure 2: (a) Four cosine bump basis functions used to represent the input filters and refractory post-spike response. (b) Example post-spike filter.

where  $u_r$  is the resting potential, and  $w_j$  denotes the synaptic strength between neuron  $i$  and a neuron  $j$ . In more involved SRM models, the filters  $\kappa(t)$  and  $\eta(t)$  may each change as a function of spike-intervals (Gerstner and Kistler, 2002). Spikes can be generated deterministically, when the potential crosses a threshold  $\vartheta$  from below, or stochastically, as a function of the membrane potential (Clopath et al., 2007; Gerstner and Kistler, 2002; Naud, 2011).

Here, we consider a filtered SRM, where input spikes are filtered with a weighted set of  $k$  basis functions  $\kappa^k(t)$  (e.g. Figure 2a for cosine bump basis functions) and the refractory response is composed of a set of  $m$  weighted basis functions  $\eta^m(t)$  (Figure 2b):

$$u_i(t) = \sum_j \sum_k w_j^k \sum_{\{t_j\}} \kappa^k(t - t_j) - \sum_m w_i^m \sum_{\{t_i\}} \eta^m(t - t_i) + u_r, \quad (2)$$

where each basis function is associated with a weight:  $w_j^k$  and  $w_i^m$ . Such a formulation effectively allows the filtered SRM neuron to have more complex temporal receptive fields, mimicking multiple synaptic inputs with multiple time-constants, or input from highly correlated neurons Goldman (2009). The standard SRM<sub>0</sub> from (Gerstner and Kistler, 2002) can be recovered by choosing a single basis-function for the input filters and refractory response.

We choose the filter-based Spike Response Model for large-scale spiking neural network simulation for two reasons: first, it has inherent parallelism for update dynamics, as each filter has a finite, relatively short temporal extent and past input spikes can be efficiently accounted for by maintaining and updating a buffer. Second, filter-based spiking neuron models are less sensitive to numerical error: given the input spikes, the numerical error in the membrane potential does not accumulate over time like in models based on differential equations, where the time-resolution for integrating fast potential dynamics is also problematic (Yudanov, 2009). This suggests that filter-based spiking neurons could be more amenable for single precision computation, the native precision of GPUs.

### 3 GPU Computing

The architecture of a GPU determines both the challenges and the opportunities for using the GPU as a massively parallel streaming co-processor. Where a CPU typically consists of four to eight CPU

<b>Vendor</b>	NVIDIA	AMD	Intel
<b>Brand and Model</b>	GeForce GTX 470	Radeon HD 7950	Core i7 2600
<b>Number of Compute Units</b>	14	28	4/8 <sup>[1]</sup>
<b>Clock Frequency [MHz]</b>	1215	800	3400
<b>Memory Capacity [GB]</b>	1	3	[16]
<b>Memory Bandwidth [GB/s]</b>	134	240	21
<b>Single Precision FP Perf [GFLOPS]</b>	1088	2870	216
<b>Double Precision FP Perf [GFLOPS]</b>	136	717	108
<b>SIMD Width</b>	32	64	256

Table 1: Basic parameters of GeForce GTX 470 and Radeon HD 7950 GPUs used in the experiments, compared to an Intel Core i7 2600 CPU. <sup>[1]</sup> Although there are four CPU-cores, they are presented as 8 cores in OpenCL due to hyperthreading.

cores, a GPU consists of hundreds or even thousands of smaller computational cores. Consequently, algorithms suitable for GPUs have to be parallelized to a very fine grain to keep all the computational cores busy. To complicate matters further, and unlike typical parallel algorithms for typical CPUs, many tasks have to be scheduled to each core to hide memory latency. Consequently, not all algorithms are suitable for acceleration by GPUs.

GPUs are a part of a host computer system which already has a main processor (CPU) and main memory (RAM). As co-processors, GPUs have processors and memory of their own. To carry out any computation, the program and the data have to be copied from the system main memory to the GPU memory. Then the GPU carries out the computation; after that, the results need to be transferred back from the GPU memory to the main memory. In some emerging architectures, the GPU and a CPU share a memory; current state-of-the-art GPUs however are discrete, therefore this work focuses on discrete architectures.

There are currently two leading general purpose GPU manufacturers, AMD and NVidia. With the AMD GCN-based GPUs and the NVIDIA Fermi/Kepler-based GPUs, the compute architectures are converging, although each still has its own set of technical details. In our experiments, we use high-end cards from both manufacturers, see Table 3 for detailed specifications. Abbreviating, the GPUs are referred below by the respective brands, Radeon and Geforce.

OpenCL has emerged as the open standard for heterogenous computing in general, and GPU-computing in particular. Recent studies (Du et al., 2011; Fang et al., 2011) have shown that the level of performance attained with OpenCL is essentially the same as with the NVidia proprietary CUDA framework. As both vendors support OpenCL on their hardware, we choose this standard for implementation of the proposed algorithms.

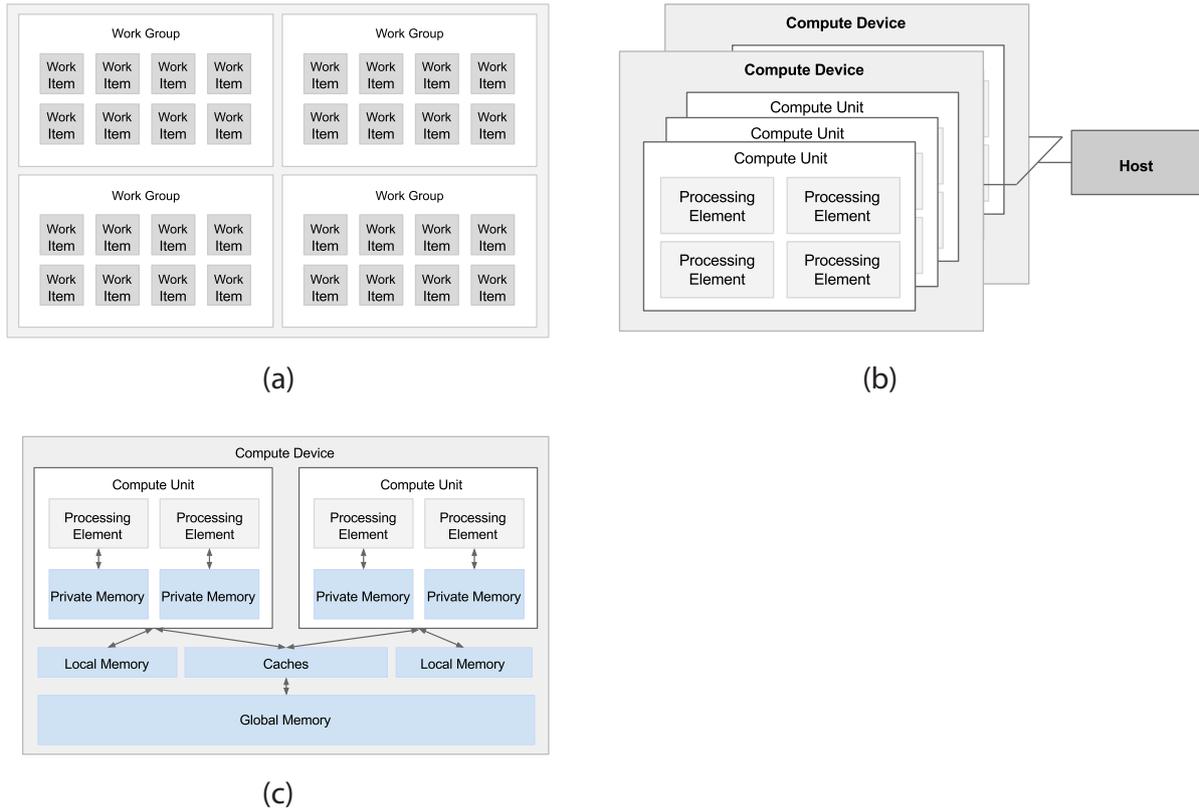


Figure 3: GPU Computing Architecture. (a) Function parallel work items and work groups. (b) Many-core organization. (c) Memory hierarchy.

### 3.1 Architecture

Here, we describe the main architectural concepts for GPU computing, also depicted in Figure 3. For compatibility with the literature, we use the OpenCL terminology (Stone and Gohara, 2010). There is no OpenCL term for a set of *work-items* working in a lock-step, which is called a *warp* in NVIDIA's terminology and a *wavefront* in AMD's. To avoid using vendor specific terminology, we will use the name *bunch* to refer to such a set.

**Kernels, work items, work groups, local/global sizes.** A computation on a GPU is performed by running multiple instances of the same function, a *kernel*. As depicted in Figure 3a, each instance runs in its own thread, and is called a *work item*. *Work items* can be grouped together into *work groups*. Both global size (all *work items*) and local size (*work items* per *work group*) can have up to three dimensions for convenient decomposition of the problem. Each *work item* can access its own data structure and adjust the computation based on it - e.g. perform some calculation for the  $i^{th}$  neuron.

**Cores, lock-step, divergence, group-mapping, latency hiding.** Modern GPUs consist of many *compute units* (Figure 3b), each containing multiple *processing elements*, all of which execute the same

instructions in lock-step. Each *processing element* runs a *work item*, therefore a small set of *work items* need to execute the same code, we call this set a *bunch*. If there is any divergence in execution within a *bunch*, all the elements have to walk through all the execution paths. This also means that there is no need for any synchronization within this set. To hide memory latency, a few sets can occupy one *compute unit* at the same time: if one set is waiting for a memory transfer, another can execute in the mean time.

**Memory levels, cooperation, local synchronization.** GPUs have several memory levels - large but slow global memory; read-only constant memory; spatially cached read-only texture memory; local memory and registers (Figure 3c). The local memory is orders-of-magnitude faster than the global memory and is accessible to all *work items* running on one *compute unit*. This allows for cooperative memory access by *work groups* (which for this reason always run on one compute unit), and also for local synchronization within these groups.

**Global synchronization, kernel launch overhead.** GPUs generally lack capabilities for global synchronization within *kernels*. Applications that need to synchronize are usually realized by *kernel* execution followed by the launch of another. There is, however, a noticeable overhead associated with launching a *kernel*. Although AMD hardware supports global synchronization through Global Wave Synchronization, this is not exposed to the programmer in the current OpenCL SDK v2.7. Upcoming NVIDIA hardware (GK110) and software (CUDA 5) will also include support for a similar feature, Dynamic Parallelism, additionally allowing running kernels to dynamically launch additional kernels, and also have the parent kernels wait for completion of the spawned child processes.

Summarizing, effective GPU-algorithms have to attain a very high degree of parallelization with functionally identical work items, and sufficiently many work items have to be scheduled for each core to hide memory latency. Both vendors offer programming and optimization guides: (AMD, 2012) and (NVIDIA, 2012), respectively. Additionally, the memory access itself should be accessed in a way that optimizes the bandwidth and the number of cache hits.

## 3.2 Optimizing Memory Usage

**Coalesced Access Patterns.** According to the respective programming guides, on both AMD and NVidia hardware the maximum transfer performance is achieved if subsequent *work-items* access subsequent elements in `global` memory. On NVidia hardware the memory transfers requested by a *bunch* and done in a set of 32- 64- or 128-Byte transactions, therefore it is called coalesced if each *bunch* accesses a 32- 64- or 128-Byte aligned segment of memory; otherwise the *bunch* performs series of transactions *serially*. On AMD the memory is organized into channels and banks and one must minimize bank and channel conflicts; in this case, a conflict also causes *serialization* of the operations. Ensuring that access

patterns are as close to optimal as possible is therefore important.

**Staggered Offsets.** On AMD hardware if the global access pattern is coalesced then each *bunch* accesses only one channel. This pattern is actually efficient if all the *compute-units* access different channels, so the *compute-units* can access memory in parallel; the worst-case scenario occurs when a single *compute-unit* accesses all the channels. A common access pattern in processing a two-dimensional array is when consecutive *work-groups* (so consecutive *compute-units*) access data with a large power-of-two stride, leading to channel conflicts. In this case a simple coordinate transformation minimizes the number of such conflicts. An example of such a change in the order of accessing an array’s tiles is illustrated in Table 2. We use staggered offsets where possible, as they improve performance on AMD hardware and are neutral on NVidia hardware.

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 3)	(1, 0)	(1, 1)	(1, 2)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 2)	(2, 3)	(2, 0)	(2, 1)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 1)	(3, 2)	(3, 3)	(3, 0)

Table 2: Order of processing before (left), and after (right) applying coordinate transformation. Tiles which are processed at the same time are marked by the same color.

## 4 Simulating Neural Populations

The simulation of large scale neural networks can be divided up naturally in a number of main steps that each allow for fine-grained parallel simulation. Neural simulation typically progresses in three major steps: neural state updating, spike generation, and spike distribution. Neural updating and spike generation are often taken together, giving two major steps (Brette et al., 2007). As depicted in Figure 4, the neural state update accounts for new inputs to the neuron, and the evolution of the internal dynamics due to past inputs and outputs. The spike generation step then evaluates whether the new internal neural state causes the neuron to generate an outgoing spike. In the final step, the generated spikes are delivered to the neurons that each spiking neuron is connected to.

As streaming parallel GPU computing requires that the same function is applied in parallel, we separate the steps of state updating and spike generation. For state updating in filter-based spiking neurons, the treatment of new input spikes requires that a weighted set of basis function filters should be added to its membrane potential for each new spike. Likewise, the presence of an output spike in the

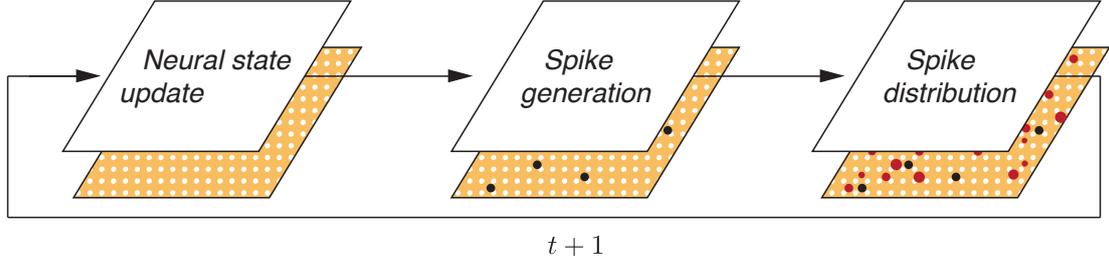


Figure 4: Neural Simulation Flow Diagram

previous time step has to be similarly accounted for using the post-spike filters.

One can either accumulate all such past events and calculate the current value of the membrane potential on-the-fly as needed, or keep a buffer with current predictions for the future and modify it for new incoming events. The first approach requires storing a history of past spikes and update weights associated with them. The update step requires only storing of the sum of update weights for the current time step, as the state itself is dynamically calculated from them. To calculate the current value of the membrane potential of a single neuron a large sum over the past spikes needs to be computed, each element of which is a dot product of vector of update weights and vector of basis function values. The sum itself can be efficiently computed using a parallel sum algorithm. The second approach requires storing of a prediction of membrane potential values for the future. The update step needs to update the predictions for many future time steps, which can be done in parallel. The update values are calculated just as in previous case, as a dot product of update weights and basis function values. To determine the membrane potential of a given neuron requires only loading of the current prediction.

We chose the second approach for our implementation for a number of reasons. First, the amount of memory needed for the storage of the data structure that can possibly fit incoming spikes for every time step and for every neuron is smaller by a factor of  $K$ , where  $K$  is the number of used basis functions. Furthermore, if a neuron receives spikes in  $S$  time steps, each spike influences the neural dynamics for some number  $T$  of time steps. In the first approach,  $K$  stored update weights need to be read  $T$  times in the future for every spike:  $K * T * S$ . In contrast, the second approach requires 1 reads and 1 writes of  $T$  prediction values instantly in each of those  $S$  steps:  $2 * T * S$ . This fact alone means that already with  $K = 2$  the overall amount of memory to be transferred is equal, even despite the fact that the same predictions are read and written many times. Second, and importantly, it is much easier with this method to achieve memory access coalescence. The second approach is furthermore straightforward to optimize, without increasing the divergence, for cases when not every neuron receives a spike in every time step, which is often the case even in large-scale simulations. Additionally, we use the buffer with predictions also for providing a buffered external input current to the network.

## 4.1 Computational Analysis

To analyze the complexity of the simulation, we consider a single iteration and assume that the simulation consists of  $N$  neurons, with  $S$  synaptic connections each. The length of both the post-spike filters  $\eta^m(t)$  and input filters  $\kappa^k(t)$  is  $T$  and there are  $K$  basis functions each. From the  $N$  neurons,  $N_{pre}$  are emitting spikes in a given iteration and  $N_{post}$  neurons will receive those spikes via synaptic connections,  $N_{update}$  is the total number of neurons that need to be updated, the joint set of sending and receiving neurons.

**Neural Update.** In the simulation of filter-based spiking neuron models, only  $N_{update}$  of all the  $N$  neurons need to update their internal state. That requires an update of  $N_{update} \cdot T$  values. The  $K \cdot T$  values of basis functions need to be read (or calculated). Each of the update values is calculated from a dot product of weights and basis functions values - two vectors of length  $K$ .

**Spike Generation.** Generating spikes, in general, requires checking each of the  $N$  neurons, to determine whether it generates a spike at given time  $t$ . If we consider  $I$  iterations, at least  $N \cdot I$  such checks must be done. The checks can also be done during the update, but since the internal state values may change, it means roughly  $k$  times more checks, where  $k$  is the number of spikes sent or received during the interval of  $I$  iterations.

**Spike Distribution.** This phase requires all  $N_{pre}$  spiking neurons to consider the weights associated with post-spike filter and input filters for all  $S$  outgoing synaptic connections ( $K$  weights each). Those  $N_{pre} \cdot (S + 1)$  values need to be accumulated for all the  $N_{update}$  neurons and the total update weight values need to be calculated from them.

From this analysis, we see that all of the phases are memory intensive, as the ratio of computation to memory transfers is fairly low. This suggests that the main optimizations should be concerned with improving memory access patterns and reducing the amount of memory that has to be accessed.

## 5 Data Structures

While we divide up the neural simulation process in three steps, three data structures are essential for parallel neural computation: the weight matrix, which is a directed graph representing the neural network structure; the potential buffer, which for each neuron stores the superposition of filters resulting from past inputs and outputs; and the compact list structure, which is used in many places in the simulation where elements need to be collected or added for further processing. Below, we describe these data structures, and how we implement them for GPU-acceleration of filter-based spiking neural network simulation.

**Weight Matrix.** Representing synaptic connection weights corresponds to representing a weighted graph, or a dense matrix in general. As neural network models tend to be sparse, a dense representation (one including all  $N^2$  values, including empty elements) of the weights is not feasible. Assuming that four basis functions are used, it would require 1.5GB to store weights of just 10,000 neurons. For truly large, but sparsely connected networks, more efficient representations are needed.

We use a compressed sparse row (CSR) format to store the weights. In this format a matrix is stored in three one-dimensional arrays. The first array stores all non-zero values as read from a full matrix left-to-right, top-to-bottom; the second stores the original column indices for all those elements; the third stores for each of the original rows the index of its beginning in the two previous arrays, and then at the end of the array the total number of non-zero elements. For  $N$  neurons with  $S$  synaptic connections each and  $K$  synaptic weights such representation requires  $N \cdot (S + 1) \cdot (4 \cdot K + 4) + 4 \cdot (N + 1)$  bytes of memory. For 10,000 neurons with 100 synapses each this is less than 20MB.

Since this representation allows for more efficient processing of matrix in per-row than per-column basis, a row can correspond to either a spike emitting (presynaptic) or receiving (postsynaptic) neuron. For efficiency of the spike distribution phase, we chose the row to correspond to a presynaptic neuron, so that it lists all the postsynaptic neurons and the associated weights in contiguous memory for efficient access.

**Potential Buffer.** The internal neuron state buffer – the potential buffer – is a two-dimensional array of size  $N \times B$  where  $N$  is the number of neurons and  $B$  is the length of the filters. As the time advances in the simulation, there is always need to keep next  $B$  values in the buffer. For a single neuron this can be implemented using a circular buffer, in which the effective index is the time modulo length of the array. For all neurons together the corresponding buffer is a so-called cylindrical buffer in which one of the dimensions is circular. The array can be stored so either subsequent time steps for a single neuron are contiguous in memory, or so that subsequent neurons for given time are. We cover algorithms for neural updating in more detail in Section 6.1.

**Compact Lists.** In many places in the simulation there is a need to have a list of elements and an operation for appending an element at the end: for example a list of neurons currently spiking, or the accumulated weights for postsynaptic neurons. On a CPU the standard approach is to employ a suitable data structure, such as a vector or a list; for a multi-threaded CPU application a synchronized version of such data structure would be used.

On a GPU, a vector or list data structure can be maintained similarly using atomic counters. The use of atomic counters however hurts performance, since append operations issued at the same time are serialized. A more common approach is to first gather all the data in a sparse format and then transform this format to the desired dense format. This problem is usually called *Compact* and can be solved by using a parallel algorithm for a *Scan* operation - that is, calculating all the prefix sums for an array.

We tested two algorithms, one using a parallel *Reduction* (Harris, 2008) and following with sequential

*Reduction* and *Scan* (Sengupta et al., 2008) - resulting in two kernel launches, and the recursive *Scan* algorithm - requiring more additional memory and resulting in up to three kernel launches for hundreds of thousands of neurons. In both cases we modified the algorithms so that there is no need to explicitly write the final *Scan* results. We find that performance of either algorithm is similar, where for relatively small problems, with tens and hundreds of thousands of operations, Reduction+Scan usually performs somewhat better than recursive Scan; therefore we choose Reduction+Scan in our implementation.

Having selected the *Reduction+Scan* for the Compact list data structure, we consider three further technical optimizations that are known to accelerate GPU computing:

**Opt1** loop unrolling and barrier removal.

**Opt2** reading and writing two elements per *work item*.

**Opt3** not writing all the results, just the list and a marker.

We incrementally added these optimizations to the *Reduction+Scan* data structure, and incremental performance improvement is plotted in Figure 5. Cumulatively, these optimizations improve the performance of the *Reduction+Scan* data structure, for a list of size 250,000 elements by a factor of three compared to baseline performance (Table 3).

Table 3: Comparison of running times for an array of 250,000 elements on the Radeon HD7950.

Version	Time [ms]	Improvement [%]	
		Step-wise	Total
<b>Base</b>	0.137	100	100
<b>Opt1</b>	0.099	138	138
<b>Opt2</b>	0.052	190	263
<b>Opt3</b>	0.045	116	304

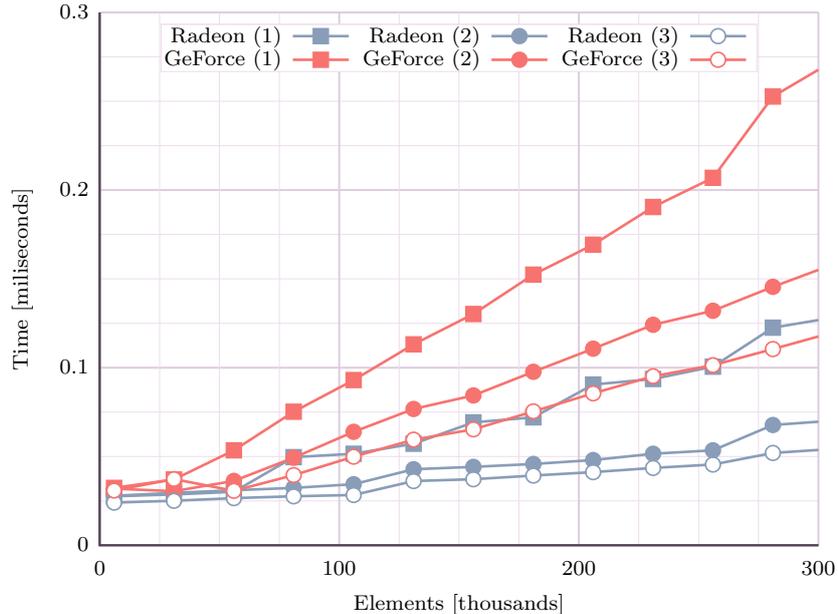


Figure 5: Running times for *Reduction+Scan* with successive optimizations **Opt1** (1), **Opt1+Opt2** (2) and **Opt1+Opt2+Opt3** (3). Optimizations incrementally improve performance, altogether reducing run-time by a factor of three. Both Radeon and GeForce GPUs exhibit similar linear performance, where the ratio of performance approaches the ratio of the respective peak bandwidth of 1.8.

## 6 Algorithms

We consider each of the three steps in the neural simulation process for both algorithmic and technical optimizations, given the data structures described in Section 5. As noted, for large neural networks, spike distribution always dominates the simulation complexity (Brette et al., 2007). We find however that the expensive operation of forward updating in filter-based neurons dominates running times for network sizes of up to several tens of thousands of neurons with sparse connectivity and moderate, realistic average network activity (see also Section 7.3). Therefore, we here focus on optimize the neural updating step to maximize parallelism and tune memory access patterns to the GPU architecture. Below, we describe these optimizations, and how we implement them for GPU-acceleration.

### 6.1 Neuron Update

The potential buffer can be stored in two ways: neuron-major order, where the subsequent time bins for the same neuron are stored contiguously; and time-major order, where states of subsequent neurons for a given time step are stored contiguously.

The buffer order affects the memory access patterns. In the time-major order case, synchronous spike checks have coalesced read patterns, and updates can have partly coalesced writes when neighboring neurons are updated at the same time. In the neuron-major order case, the read in a synchronous check

is scattered, but the update of a single neuron results in writes to contiguous locations - possibly allowing for an efficient parallelization. Similar to array storage, the algorithm used for the update may use either neurons ( $N$ ) or time ( $T$ ) as the primary dimension.

**Neuron-major (1).** In the simplest case of a *neuron-major* approach, we launch  $N$  *work items*: one for updating each neuron (Listing 1). To achieve partly coalesced write patterns, a neuron-major representation of the buffer is used. Since every *work item* accesses the basis function values in the same order, storing those values in constant memory, optimized for access patterns like this, greatly improves the performance.

---

```

1 foreach n in neurons in parallel
2   sum = get_sum_of_weights(n)
3   if (sum != 0)
4     for dt = 0..T-1
5       state[n, t+dt] += dot(sum, basis[dt])

```

---

Listing 1: Pseudocode for Neuron-major neuron update

**Compact neuron-major (2).** In filter-based spiking neural networks, for any given time step, only those neurons that receive incoming spikes need to be updated, as well as those neurons that spiked in the previous time step. One optimization then is to first use *Compact* on the input array with sums of weights. This approach reduces divergence among the threads resulting in more efficient execution, but also results in lower coalescence in the writes.

**Time-major.** The simplest case of a *time-major* approach would be launching  $T$  *work items*: one for each time bin. Every *work item* processes (sequentially) the neurons that need to be updated from the *compact* list. This results in coalesced writes to the array in time-major representation. *Work items* can also buffer the neuron list in local memory to achieve coalesced reads. The number of time bins usually is, however, small compared to the number of neurons, and implementations like this tend to under-utilize the GPU, despite theoretically achieving very good write and read efficiency.

**Two-dimensional (3).** Unifying the best features of time-major and neuron-major approaches is launching *work items* in both dimensions. The primary dimension is time ( $T$ ) - to ensure that on a low-level the *work items* working in lock-step are the ones writing to contiguous memory locations. Since the whole length of the array needs to be updated, the writes are fully coalesced. The second dimension is neurons ( $N$ ) to ensure enough parallel workload is given to the GPU (Listing 2, for  $X = T$  and  $Y = N$ ).

**Technical optimizations.** In the two-dimensional case we spawn  $X \cdot Y$  *work items* and test for optimal values of  $X \leq T$ ,  $Y \leq N$ . When  $X < T$  and/or  $Y < N$ , each of the *work items* contains a loop over a few elements (either over  $T$ ,  $N$ , or both). Because of different access patterns, in this case

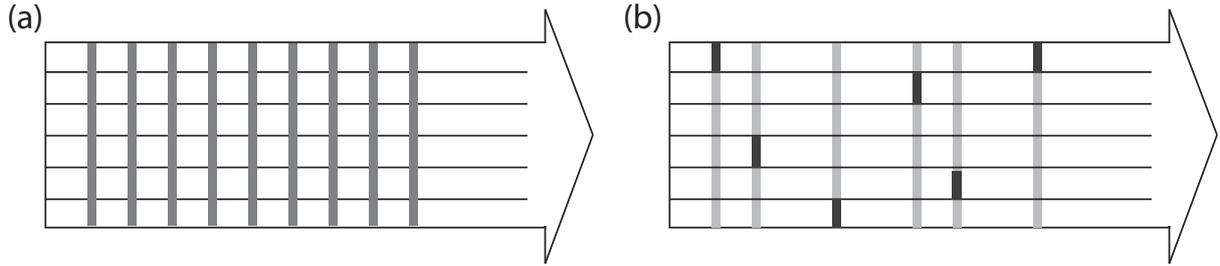


Figure 6: (a) Synchronous spike-generation: all neurons are checked at each time step; constant time steps. (b) Asynchronous updating: all neurons are checked only when (at least) one neuron generates a spike; variable time steps.

it is also more efficient not to store the basis function values in constant memory, as the use of the global cache results in better performance. The reads and writes are done using Staggered Offsets. We additionally eliminate the need for scattered reads in synchronous spike checks by introducing a separate buffer, which is maintained on-the-fly during the update and can be checked in an efficient way.

---

```

1 for x = 0..X-1 in parallel
2   for y = 0..Y-1 in parallel
3     i = y
4     while i < size(active_neurons)
5       (n, sum, j) = (active_neurons[i], active_sums[i], x)
6       while j < T
7         state[n, t+dt] += dot(sum, basis[dt])
8         j += X
9       i += Y

```

---

Listing 2: Pseudocode for two-dimensional neuron update. The variables  $X$  and  $Y$  determine the number of work items that is spawned.

## 6.2 Spike-Generation

In principle, there are two main approaches for determining which neurons are spiking at a current time step. In a synchronous approach, there is a check at every time step to determine whether a neuron spikes (Figure 6a). This check results in a boolean vector of spikes, which is used in later simulation stages. In an asynchronous approach, every neuron keeps track of the nearest time at which it might spike (if nothing occurs until that time). Such information can be updated on-the-fly along with the neurons' internal state. This results in an array of possible spike times, which then can be effectively reduced to extract the nearest time at which any neuron will spike, as well as all the neurons that spike at this time (Figure 6b).

**Synchronous checks.** The synchronous approach is straightforward. It is optimized by ensuring efficient memory access patterns – processing of a few subsequent elements by a *work item* result in larger reads and writes. Additionally the workload is balanced by moving the work into the kernel - addition of an outer loop resulting in processing of a few such bunches with a large stride in sequential manner.

**Asynchronous events.** For asynchronous event processing, the current value for the nearest spike is read every time a neuron is updated. Then, while updating, the minimum is updated in local memory and at the end written back to global memory if needed.

If the asynchronous event processing algorithm is used, the simulation effectively has variable time steps - in theory it can advance by an arbitrary amount of time in a single iteration. If simulation requires some periodic action (e.g. input of external current, visualization, gathering of statistics) this poses a problem, as it requires time-consuming GPU-CPU synchronization just to determine current simulation time. An additional mechanism was implemented optionally allowing to enforce a maximum time step, thus allowing for implementation of such actions.

For large-scale neural networks simulated with time-steps of for example 1 millisecond, at least some neurons will be generating spikes at every time step and synchronous updating will be more efficient. However, for smaller simulations, or simulations with (much) smaller time-steps, the asynchronous method will have higher performance. We implemented both synchronous and asynchronous spike-generation methods, where we observed the expected behavior. Since we are focusing on large scale simulations, the results we further report are obtained using synchronous checks.

### 6.3 Spike-Distribution

Given the list of pre-synaptic neurons, the simulation needs to calculate for each neuron the synaptic weights for the next update. Here, we assign *work items* on a per-output basis, so that the output is an array with  $N$  entries, which later is subjected to *Compact*. While assigning the *work-items* on per-synapse basis would result in a better algorithmic complexity, this approach eliminates the need for synchronization and allows for coalescing of both reads and writes. The fact that both the spiking neuron list and synaptic lists are sorted allows for efficient processing. The possibility of cooperation within *work-groups* also partly compensates for the added complexity.

**Post-Synaptic List Check.** We implement Spike-Distribution using a Post-Synaptic List Check: each of the outputs inspects the post-synaptic lists of each spiking neuron. As every neuron needs to process the same lists, these lists can be loaded and processed efficiently in the `local` memory. Since only a fraction of the neurons are spiking at any given time point, the use of a post-synaptic list results in a smaller amount of memory to load, even though each of the groups needs to load the lists separately. The load into local memory does not limit the number of synapses per neuron, as lists larger than local

memory can be loaded in parts sequentially, at a small performance impact. As all the groups process the same lists concurrently, the access is likely to have a high cache-hit ratio.

---

```
1 foreach n_post in all_neurons in parallel
2   sum = 0, idx = 0
3   foreach n_pre in spiking_neurons
4     list = get_list(n_pre), lsz = get_list_size(n_pre)
5     idx = lower_bound(list + idx, list + lsz, n_post)
6     if (list[idx] == n_post)
7       sum += get_weight(get_index(idx))
8   update_weights[n_post] = sum
```

---

Listing 3: Pseudocode for the Post-Synaptic List Check

where `lower_bound(first, last, value)` returns an offset of the first element in the sorted range `[first,last)` which does not compare to less than `value`.

## 7 Results

In this section, we assess the performance impact of the various algorithmic improvements for a given experimental setup. Combining those results, we then present run-time results for large-scale spiking neural networks.

### 7.1 Experimental Setup

**Hardware Setup.** All experiments were performed on two machines with identical software and hardware except for the different GPUs and therefore also different OpenCL runtimes actually used. The machines featured an Intel Core i7-2600 CPU with 4 physical cores running at 3.40GHz, and 16GB of 1333 MHz DDR3 RAM. The machines were running Fedora Core 16 with kernel 3.3.5-2. Both machines used the following OpenCL runtimes and versions: NVIDIA CUDA 4.2.9 (OpenCL 1.1), AMD APP SDK v2.7 (OpenCL 1.2), Intel OpenCL SDK 2012 (OpenCL 1.1). The GPUs used are detailed in Table 3.

**Time Measurements.** To measure performance of both single algorithms and the whole simulation, we measure kernel execution times – time spent by the GPU while executing different kernels. This measure is defined by the difference in time of execution start and finish, i.e. it does not include the initial transfer of data to the GPU and the overhead from queuing and launching the kernel itself. In the cases where we consider algorithms spanning several sequential kernel launches we consider the difference between the time of execution finish of the last kernel and start of the first - i.e. the measure includes some of the overhead, for fair comparison with algorithms spanning less kernels. The measurements are taken using OpenCL `clGetEventProfilingInfo` function. All of the presented times are averaged over 10 runs, error bars in the graphs denote one standard deviation, when not show it is due to their small size.

**Numerical Verification.** For each of the algorithmic optimizations we verified the results with the reference implementation. Since the optimizations should not introduce any changes that affect accuracy of the results, the verification is done with zero error tolerance policy. Using standardized multiply-add instruction, we verified that for identical simulation settings, the results are always the same for all optimized algorithmic versions on all the platforms; we cover this in more detail in Section 8.

### 7.2 Optimizing Neural Updating

We first explore the degree to which performance is influenced by memory access patterns when updating the neural state. For filter-based spiking neurons, the principal concern is the order in which the potential buffer is accessed.

Shown in Figure 7a is the running time of a single time-step for various network sizes when updating neurons using a neuron-major kernel (1), a compact neuron-major kernel (2), and two-dimensional kernel (3). Results are shown for both Radeon and GeForce GPU, for a network with low activity, where

randomly 10% of the neurons in the network are updated in this single time-step. By comparing (1) and (2) we can see that adding a *compact* step indeed increases performance, here by 125%. We also see that using a two-dimensional approach to achieve coalesced access patterns further improves performance nearly 300% when comparing (2) and (3). Additionally, by introducing additional parallelism the two-dimensional kernel (3) greatly improves performance for smaller networks, up to 16,000 neurons, where the other two versions exhibit similar running times regardless of the network size.

We further test the algorithms by setting the number of neurons at 50,000 and varying the fraction of neurons that needs to be updated. Again, we choose neurons to be updated in a random fashion. Resulting running times can be seen in Figure 7b. We find that the compact version of neuron-major algorithm (2) slightly improves performance for lower activities but for larger activities it performs worse than the baseline. The speedup is due to the reduced divergence when using a compacted data structure, while the slowdown is caused by ineffective memory access patterns: in the baseline case the access pattern from entire *work-group* is always coalesced, even if some *work-items* might not participate; in the compact algorithm, the accesses from a single *work-group* refer to distant elements, resulting in serialization.

The two-dimensional kernel eliminates this problem by reducing the divergence while maintaining optimal memory access patterns. We find that the best performance is achieved when both local and global sizes in the time dimension are chosen to be exactly at the *bunch* size. In neuron dimension, we find that processing more elements per *work-item* improves performance at lower activities, we settle for 4 neurons per *work-item* resulting in nearly linear scaling of runtime depending on fraction of neurons to be updated. This is an important property, as the real simulation is likely to exhibit non-uniform network activity patterns with phases of lower and higher activity.

### 7.3 Large-scale Simulations

So far, we studied GPU specific optimizations to data structures and algorithms for the different neural simulation steps. Ultimately, the optimizations are aimed at improving the runtime of large scale spiking neural networks; here we present run-times and related performance measures for large-scale filter-based spiking neural networks. We report numbers only for the faster Radeon GPU; we find that the GeForce GPU is generally slower by a factor of around 1.8, consistently with the bandwidth ratio of the devices.

**Model Setup.** While our implementation allows for an arbitrary basis set, in all of the tests we use the basis function set of raised cosine bumps as shown in Figure 2 to describe both input filters and post-spike filters; both basis function sets consist of four such bumps and have a length of 1024 time bins. Simulations are run for 1 ms time steps. The connections are therefore described by four randomized weights each. Input into the network is driven by 10 constantly stimulated neurons firing at approximately 100Hz. Each neuron is connected to 100 other randomly selected neurons; 80% of

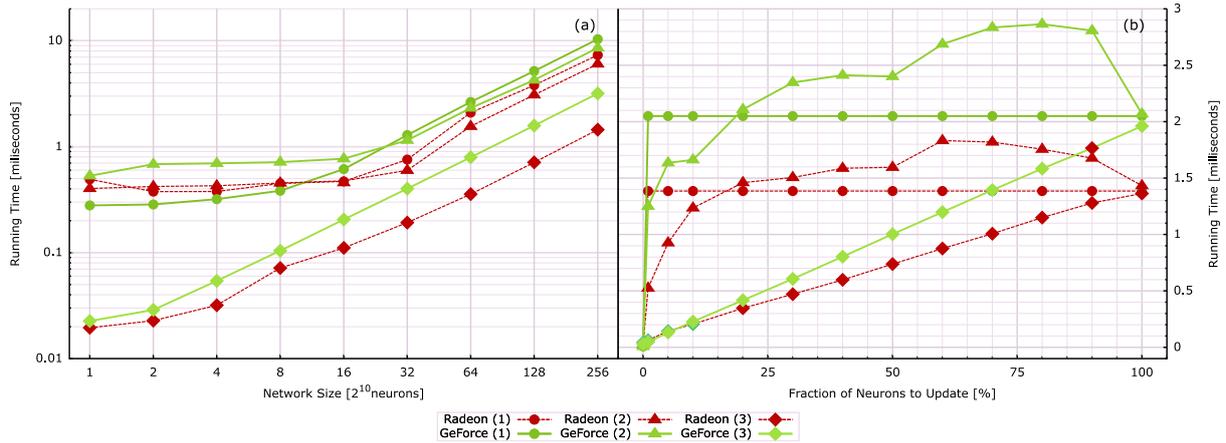


Figure 7: Optimizing Neural Updating on both GPUs. (a) Running times for (1) neuron-major kernel, (2) compact neuron-major kernel, (3) two-dimensional kernel for different network sizes when 10% of neurons is updated. The two-dimensional kernel significantly outperforms the other neural updating methods. (b) Running time of the same algorithms for 50,000 neurons depending on the fractions of neuron to update. The two-dimensional kernel shows linear scaling performance, and also outperforms the other neural updating methods when a larger fraction of the neurons requires updating in single time step.

the neurons are excitatory in the sense that the filter weights are positive; the remaining 20% of the neurons are inhibitory, projecting negative filter weights. To balance excitation and inhibition, and thus ensure network stability, the inhibitory weights were taken to be three times as large as the excitatory weights. For these conditions, the network neurons fired on average at 7-8Hz during simulation. We choose a constant number of connections per neuron rather than a fixed fraction to ensure that simulation performance for increasing network sizes is determined by linearly increasing workload, rather than the quadratically increasing weight matrix. Measurements are taken over 2 seconds of network simulation.

**Execution-time breakdown.** For the model setup, we examine the degree to which the three main simulation steps account for network simulation time for various network sizes, shown in Figure 8a. We see that the optimized neural updating step dominates the running time for network sizes up to 60,000 neurons. Beyond this network size, spike propagation becomes the dominant term.

**Processing Rate.** We also plotted the number of neurons processed by the GPU per second in Figure 8b. We see that for small network sizes the GPU runs the simulation at two times real time, and the throughput increases almost linearly as a function of network size up to networks of about 50,000 neurons. This observed “filling” effect shows that up to 50,000 neurons, the GPU is not fully utilized, and increasing the workload affects running time only slightly. For larger network size, throughput plateaus as spike-propagation becomes the dominant simulation activity. At this plateau, the GPU processes about 35 million spike events per second, counted as the number of spikes per neuron per second times

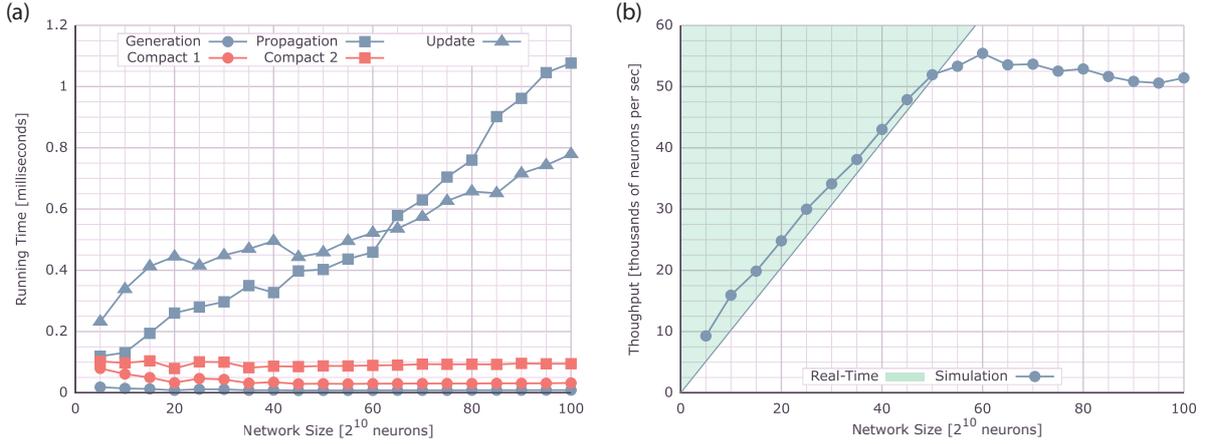


Figure 8: (a) Breakdown of runtime in simulation processing steps: neural updating (blue triangles) spike generation (blue circles), and spike propagation (blue squares). For increasing network sizes, spike propagation starts to dominate the running time. Red circles and squares show the running time for the collection of spike events (circles), and collecting update weights (squares). Measurements are taken over 2 seconds of network simulation, repeated ten times. (b) Neurons processed per second as a function of network size. Shaded area is (better than) real-time processing, where real-time is defined as the point where a millisecond of real time is simulated in less than a millisecond. For increasing network size, throughput increases almost linearly before reaching a plateau.

the number of connections times the number of neurons. Reducing the number and length of the basis function filters only improves the initial linear ramp of the graph, as spike-propagation starts to dominate the workload at larger network sizes.

**Network Activity.** One of the most influential factors on the simulation performance is the network activity. We explore the performance for different levels of uniform activity varying from 1 to 100 Hz for network comprised of 50,000 neurons with 100 synaptic connections each. The running times of different phases and the resulting simulation throughput are shown in Figure 9.

The runtime of Neural Update phase rises with increasing activity to later reach a plateau at higher activities. This effect is due to fact that all the incoming spikes are effectively aggregated in the Spike Propagation phase, and once at least one spike is received by each neuron each time step, the Update phase no longer depends on the number of received spikes per neuron. The Propagation phase runtime rises rapidly in linear fashion, and quickly becomes the dominant phase above 30 Hz. The effective throughput of the simulation also rises and slowly nears a plateau as the Propagation phase becomes increasingly dominant.

The fact that the throughput of the simulation actually rises with the increase of activity has an important consequence: the performance for a given level of uniform activity cannot be treated as an approximation of performance for non-uniform activity at the same mean level. This is an important

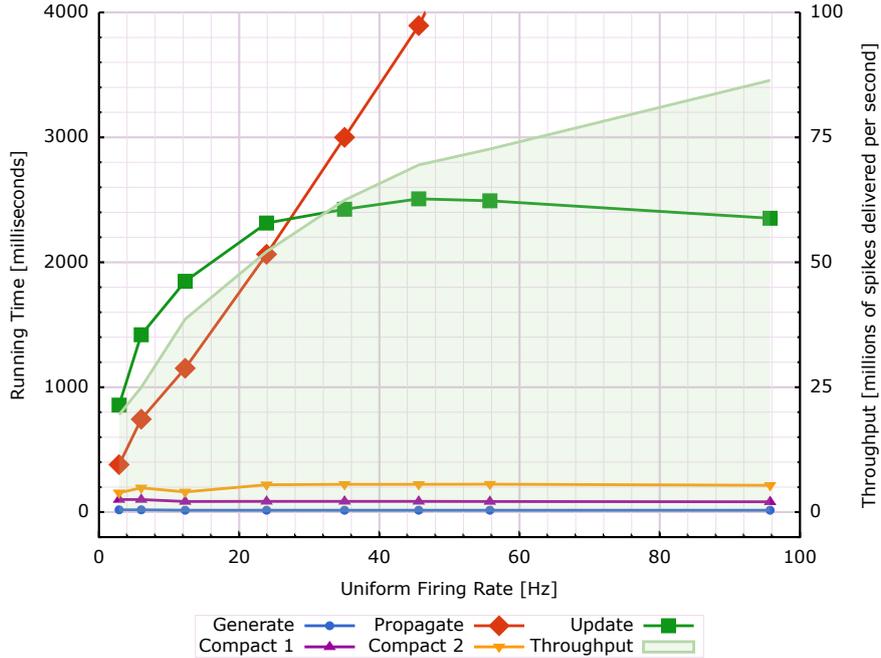


Figure 9: Dependency of Running Times (left axis, point series) and Throughput (right axis, shaded area) on the level of uniform network activity in the network for 50,000 neurons and 100 synaptic connections per neuron. For increasing network activity, Spike Propagation starts to dominate the computation; Neural Updating plateaus as each neuron is updated each time-step.

difference between filter-based and differential based approaches to spiking neuron simulation, like that in (Brette et al., 2007): in the latter, each neuron is updated each time step, at a fixed, relatively small cost. In the filter-based approach, the cost per update will be higher, but especially for lower or non-uniform activity levels *per time step*, potentially much fewer updates need to be applied.

**Connection Density.** We also explore the influence of the number of synaptic connections per neuron on the simulation performance. We run a network comprised of 50,000 neurons with varying number of uniformly distributed random connections varying from 10 to 1000. The simulations are run at two uniform activity levels of 8 and 31 Hz. The breakdown of execution time into simulation phases and the effective throughput can be seen in Figure 10.

Figure 10 shows that time spent in the Neural Update phase rises with the number of synaptic connections per neuron but then plateaus at approximately the same level for both activity levels, again due to aggregation of incoming spike events. The time spent in the Spike Propagation phase rises significantly with increasing activity, while overall throughput for a larger network still increases, which in both cases increases nearly linearly with the number of connections.

As throughput increases with neural activity, we achieved throughput as high as 600 million events per second in simulations when using 1000 connections per neuron and approximately 60 Hz of uniform activity. While such simulation scenario might not be plausible, and this number cannot be used to

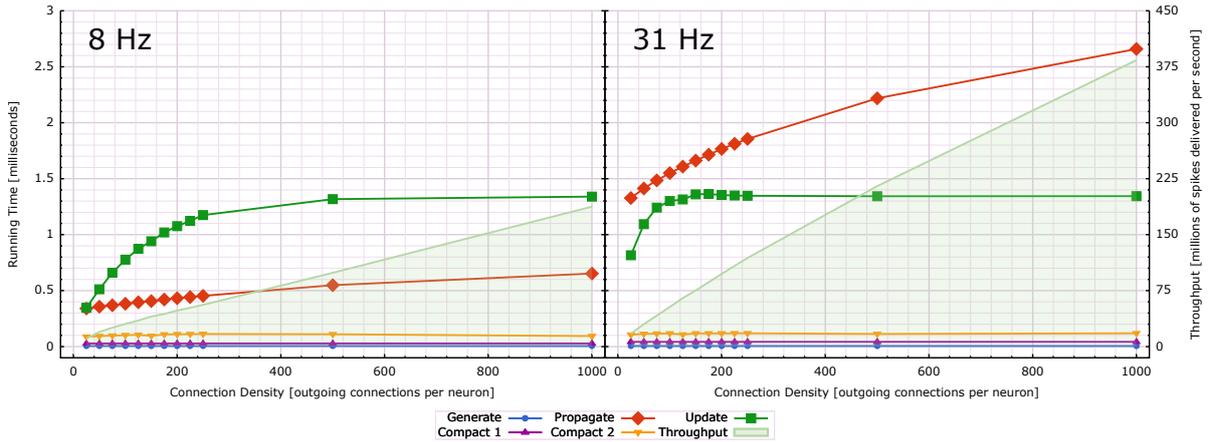


Figure 10: Dependency of Running Times (left axis, point series) and Throughput (right axis, shaded area) on the level of uniform network activity and connectivity in the network for 50,000 neurons and: (left) 8 Hz moderate activity level; (right) 31 Hz medium-high activity level. Graphs are aligned to the same scale for a clear comparison. While the computational cost of Neural Updating plateaus once each neuron is updated each time-step, the cost of Spike Propagation keeps growing. Combined, the throughput measure keeps increasing.

conclude performance in other setting, periods of very high activity do happen in realistic simulations, and this fact allows for faster processing of simulations with varying activity.

We further investigate in detail the performance of the spike propagation algorithm as a function of network connectivity. The comparison of running times for network with *uniform* 8 Hz activity and 100, 250 and 500 connections per neuron is shown in Figure 11. The small difference in performance of the post-synaptic list algorithm for 250 and 100 connections is due to fact that that the bigger list fits as well into memory as the smaller one, while in the latter case the memory could be used more efficiently.

## 8 Pitfalls

In this section we describe in more depth some of the encountered pitfalls related to parallelization and optimization for GPU computing. The issues presented here can greatly affect performance and correctness of an application; they apply to any kind of GPU-enabled application and are not limited to spiking neural network simulations.

**Numerical Verification.** Unlike earlier hardware, numerical accuracy is a lesser concern with current GPUs as they now fully conform to the IEEE 754 standard. Still, the same code compiled for different platforms might give different results, while still not violating this standard in a strict sense.

One place in our implementation sensitive to numerical accuracy is the addition of the kernel value from the basis functions and the weights. This is a dot-product operation followed by an addition

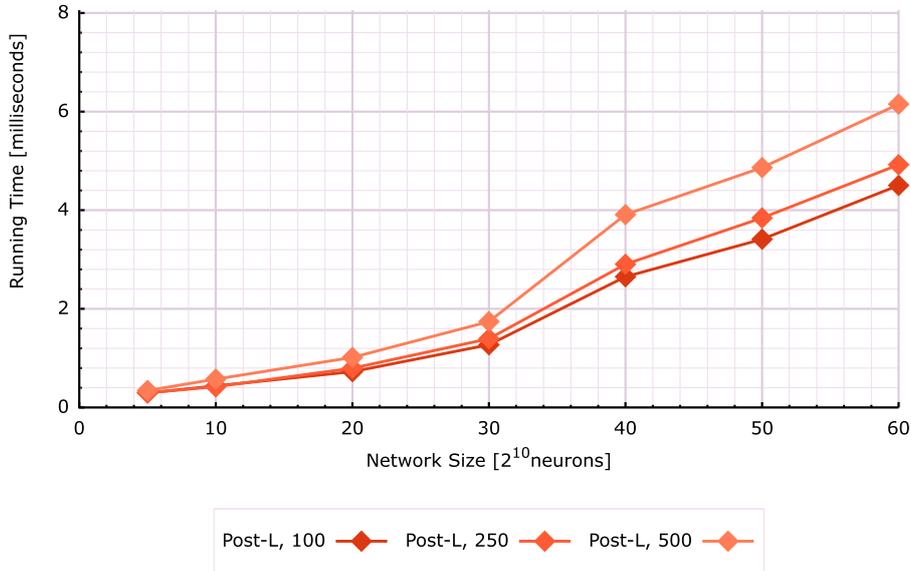


Figure 11: Performance of Spike Propagation Post-synaptic List (Post-L) algorithm at uniform 8 Hz activity network with 100, 250 and 500 connections per neuron. Increasing connectivity incurs only a modest performance penalty (results shown are for the Radeon GPU shown only, the same trends were observed on the GeForce GPU).

`value += dot(weights, basis)`. OpenCL allows for expressing this operation exactly with `dot` function. The math can also be assigned explicitly, using `+` and `*` operators. Alternatively, the shortcut `mad` function for multiply-add can be used, or the `fma` function which calculates a fused multiply-add. The latter is both faster and more accurate on the GPU, because it has only one rounding step instead of two in `a*x + b`. As a result of intermediate rounding, the algorithmic results can differ between cases with and without the usage of `fma`. Still, even the `dot` itself for a 4-vector can be in theory computed in a different order by different implementations, resulting in different results.

We computed checksums of all of the values computed in a simulation of three seconds of 51,200 neurons to determine the differences. As can be seen in Table 4, the results do differ among different OpenCL runtimes and platforms. All the platforms, however, yield exactly the same results when explicitly using `fma`, since the operations and their order is explicitly specified. Consequently, we use `fma` by default if the hardware supports it.

**Barrier Removal.** The removal of `barrier` calls, while usually not recommended for portability reasons, can greatly improve performance, and if done correctly does not introduce errors<sup>1</sup>. One of the optimizations we implemented is the removal of `barrier` calls when we know that there is no need for

<sup>1</sup>To maintain portability, versions of the code with limited use of `barrier` calls could be created based on preprocessor macro checks when compiling for a specific device or a device class.

	Radeon	GeForce	Core i7
<b>Explicit</b>	13C	A9A	13C
<b>DOT</b>	5B7	13C	C51
<b>FMA</b>	8B7	8B7	8B7
<b>MAD</b>	4F9	8B7	4F9

Table 4: Different checksums (color-coded) for the simulation values depending on the used method on different platforms, using the default compiler optimization settings. Simulations were identical when using thefma function.

<pre> 1 if (my_id &lt; 64) 2   buffer[my_id] += buffer[my_id+64]; 3 barrier(CLK_LOCALMEMFENCE); 4 if (my_id &lt; 32) 5   buffer[my_id] += buffer[my_id+32]; 6 barrier(CLK_LOCALMEMFENCE); 7 if (my_id &lt; 16) 8   buffer[my_id] += buffer[my_id+16]; 9 barrier(CLK_LOCALMEMFENCE); 10 ... </pre>	<pre> 1 if (my_id &lt; 64) 2   buffer[my_id] += buffer[my_id+64]; 3 barrier(CLK_LOCALMEMFENCE); 4 if (my_id &lt; 32) { 5   buffer[my_id] += buffer[my_id+32]; 6   mem_fence(CLK_LOCALMEMFENCE); 7   buffer[my_id] += buffer[my_id+16]; 8   mem_fence(CLK_LOCALMEMFENCE); 9   ... 10 } </pre>
(a) Original Code	(b) Code Without Barriers

Figure 12: Example of barrier removal.

the synchronization. This is the case when all the working *work items* are in lock-step, i.e. belong to one warp in NVidia, or wavefront in AMD terminology.

Care should be taken when removing `barrier` calls however, even when synchronization is not needed. Without `barrier` calls, the OpenCL compiler can actually break the code as optimizations can reorder the instruction execution order, or the results may be cached instead of being written to the local memory. This pitfall can be avoided in OpenCL by using instructions to explicitly order memory read and write operations, using `mem_fence`, `mem_read_fence` and `mem_write_fence` functions. Those functions ensure that all reads or writes made by the *work item* are visible to all other *work items* in the *work group*, but they do not cause synchronization as the `barrier` does, so only *work items* that are guaranteed to be able to read the values after encountering this instruction are the ones that are working in a lock-step with *work item* in question.

**Synchronization Interval.** We have found that on AMD hardware on the Linux operating system there is a large overall overhead - time not actually spent in the kernels - during the simulation. We were able to greatly reduce this overhead, gaining speedups from 20% to over 40% in simulations of up to hundreds of thousands of neurons. This overhead is associated with what we call *synchronization interval*

$I_s$ : the number of simulation time steps submitted to the GPU before waiting for their completion.

We measured the real running time using standard UNIX `gettimeofday` function for several long simulations in order to determine the size of this overhead depending on  $I_s$ . The simulations did not include any additional operation after the synchronization (which would usually be the case in a real network simulation). As can be seen in Figure 13 the running time decreases significantly on the Radeon HD 7950 with increasing interval; on the GeForce GTX 470 the tendency is the same but the change itself is barely measurable. Percentage-wise, reducing the overhead obviously aids the smallest network sizes the most, although for the Radeon we still see a 20% performance increase for a network with 130,000 neurons.

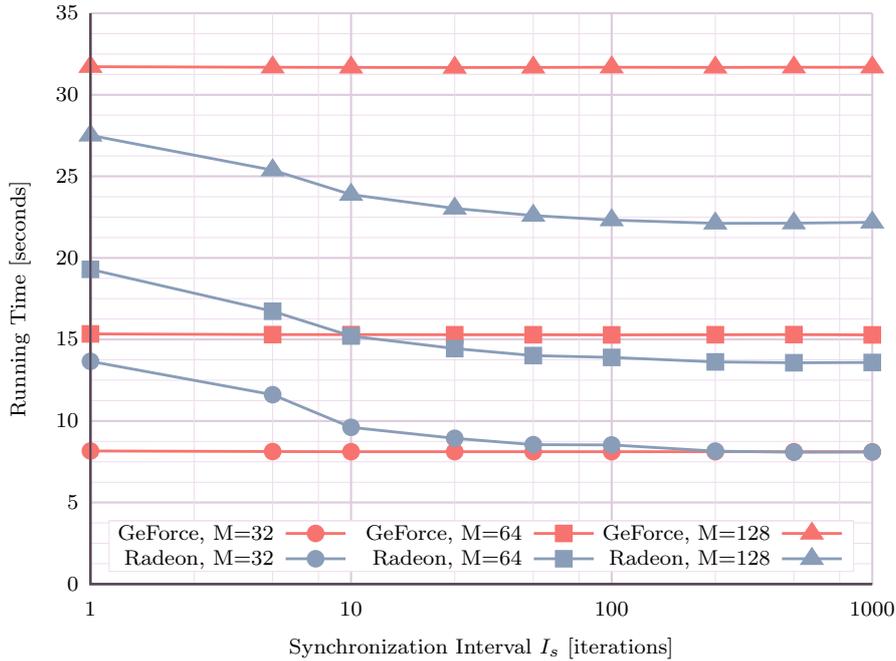


Figure 13: Running Times for  $N = M \cdot 1024$  neurons depending on  $I_s$ . The Radeon benefits significantly from increasing the synchronization interval to at least once every 10 iterations.

In the current implementation, there are only a few cases when such synchronization between GPU and CPU is necessary. First there is visualization of the network activity, which requires GPU/CPU synchronization every few time steps to ensure synchronization between OpenCL and OpenGL contexts. Second is the profiling mode - gathering performance measurements after every iteration. Another important case is when there is a need to send data between the CPU and GPU, e.g. downloading results for validation, or uploading of external input to the network. In this case, if buffering and sending data less often is acceptable it can highly improve performance because of both this overhead and the efficiency of the transfer itself.

We also found that for intervals bigger than 250-500 iterations the overhead slowly begins to rise again. Bigger values of  $I_s$  also have a side effect of increasing the memory usage. With this in mind,

we settled for a default interval  $I_s = 250$  in our simulations, even when there is no actual need for synchronization.

## 9 Discussion

We developed algorithms for implementing large scale neural networks with filter-based spiking neuron models on general purpose GPUs. While parallel algorithms for spiking neurons have been developed, in particular for differential based neuron models, we focus on adaptations of the standard simulation workflow specific for the many-core streaming computation architecture of GPUs. Additionally, we focus on filter-based models for two reasons: first, filter-based models allow an extra dimension of parallel neural state updating. Second, as filter-based models compute the internal neural state as a superposition of filters, such computations are less sensitive to numerical errors as compared to integration methods for differential equations. Since GPUs are typically much more effective for single precision computations as compared to double precision, a significant speedup can be achieved at little cost in terms of simulation accuracy.

A substantial body of work has focused on efficient ways of computing spiking neurons, see e.g. the review in Brette et al. (2007), More recent work has also explored vectorized algorithms suitable for streaming computing using for example the SIMD units on CPUs, and also with GPU computing in mind (Brette and Goodman, 2011). We specifically focused on algorithms that improve on this work by taking into account the specifics of general purpose GPU computing, such as optimal work set sizes, and data structures and update algorithms optimized for a GPU's memory hierarchy. We then considered the effect of technical optimizations that further optimize the simulation for the quirks of current hardware and software implementations. We find that both the architectural optimization and the technical optimization improve the running speed of the simulation significantly. With these optimizations, we can run in real-time (or better) networks comprised of up to 50,000 neurons, with each neuron spiking at on average 7-8 Hz (and the GPU thus processing 35-40 million spike-events per second).

Having implemented our optimizations, spike-distribution starts to dominate the run-time of simulations. Here, network activity clustering like in (Fidjeland et al., 2009) may help scale the networks further. This would also be an promising approach for dividing large scale simulations over multiple GPUs. While our optimizations are generic for the simulation workflow, algorithms could further be tuned to specific network structures and neuron models, such as configurations of special biological interest.

A commonly reported benchmark is the speedup that is achieved by running simulations on a GPU versus the same simulation running on a CPU. To a large degree this is a meaningless number since it is not clear to what degree optimization efforts on either GPU or CPU are exhaustive. At the same time, theoretical peak numbers for either GPU or CPU are hard to compare as the extent to which actual peak performance can be achieved depends greatly on the degree to which an algorithm can be parallelized

and also on specific optimizations.

For GPUs acceleration to be useful for a broader public, the intricacies of GPU optimizations have to be largely hidden from neural modelers (and the same holds for CPU optimizations). For example, the performance penalty for synchronization in current AMD Linux drivers requires some buffering of input and output requests to CPU-driven parts of the simulation such as visualization, or inter-GPU communication when multiple GPUs are present. The memory access optimizations, algorithms and buffering solutions presented here are fairly generic for spiking neuron simulations, and the algorithms serve as showcases for the amount of improvement that detailed optimizations can provide<sup>2</sup>.

---

<sup>2</sup>The sources are available as open source from <http://www.cwi.nl/~sbohte>

## References

- AMD 2012. AMD Accelerated Parallel Processing OpenCL Programming Guide, Revision 2.2.
- BRETTE, R. AND GERSTNER, W. 2005. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiology* 94:3637–3642.
- BRETTE, R. AND GOODMAN, D. F. M. 2011. Vectorized algorithms for spiking neural network simulation. *Neural computation* 23:1503–1535.
- BRETTE, R., RUDOLPH, M., CARNEVALE, T., HINES, M., BEEMAN, D., BOWER, J. M., DIESMANN, M., MORRISON, A., GOODMAN, P. H., HARRIS, F. C., AND OTHERS 2007. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience* 23:349–398.
- CLOPATH, C., JOLIVET, R., RAUCH, A., LÜSCHER, H. R., AND GERSTNER, W. 2007. Predicting neuronal activity with simple models of the threshold type: Adaptive Exponential Integrate-and-Fire model with two compartments. *Neurocomputing* 70:1668–1673.
- DU, P., WEBER, R., LUSZCZEK, P., TOMOV, S., PETERSON, G., AND DONGARRA, J. 2011. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing* .
- FANG, J., VARBANESCU, A. L., AND SIPS, H. 2011. A Comprehensive Performance Comparison of CUDA and OpenCL. *In Proceedings of International Conference on Parallel Processing (ICPP)*, pp. 216–225. IEEE.
- FIDJELAND, A., ROESCH, E., SHANAHAN, M., AND LUK, W. 2009. NeMo: A platform for neural modelling of spiking neurons using GPUs. *In Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pp. 137–144. IEEE.
- FIDJELAND, A. AND SHANAHAN, M. 2010. Accelerated simulation of spiking neural networks using GPUs. *In Neural Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1–8. IEEE.
- GERSTNER, W. AND KISTLER, W. 2002. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press.
- GOLDMAN, M. 2009. Memory without feedback in a neural network. *Neuron* 61:621–634.
- HAN, B. AND TAHA, T. M. 2010a. Acceleration of spiking neural network based pattern recognition on NVIDIA graphics processors. *Applied Optics* 49:B83.
- HAN, B. AND TAHA, T. M. 2010b. Acceleration of Spiking Neural Network on General Purpose Graphics Processors. Master of science in electrical engineering, University of Dayton.
- HARRIS, M. 2008. Optimizing Parallel Reduction in CUDA. *NVIDIA Developer Technology* .

- HODGKIN, A. L. AND HUXLEY, A. F. 1952. Propagation of electrical signals along giant nerve fibres. *Proceedings of the Royal Society of London. Series B, Biological Sciences* 140:177–183.
- JOLIVET, R., RAUCH, A., LUESCHER, H. R., AND GERSTNER, W. 2006. Integrate-and-Fire models with adaptation are good enough, pp. 595–602. *In* Y. Weiss, B. Schölkopf, and J. Platt (eds.), *Advances in Neural Information Processing Systems* 18. MIT Press, Cambridge, MA.
- KRISHNAMANI, P. AND VENKITTARAMAN, V. 2010. Acceleration of spiking neural networks on single-GPU and multi-GPU systems. Master’s thesis, Clemson University.
- LAZAR, A. A. AND ZHOU, Y. 2012. Massively Parallel Neural Encoding and Decoding of Visual Stimuli. *Neural Networks* 32:303–312.
- MARKRAM, H. 2006. The Blue Brain Project. *Nature Reviews Neuroscience* 7:153–160.
- MUTCH, J., KNOBLICH, U., AND POGGIO, T. 2010. CNS: a GPU-based framework for simulating cortically-organized networks. Technical Report MIT-CSAIL-TR-2010-013/CBCL-286, MIT, Cambridge, MA.
- NAGESWARAN, J. M., DUTT, N., WANG, Y., AND DELBRUECK, T. 2008. Computing Spike-based Convolutions on GPUs. *Architecture* .
- NAUD, R. 2011. The Dynamics of Adapting Neurons. PhD thesis, EPFL Lausanne.
- NVIDIA 2012. OpenCL Programming Guide for the CUDA Architecture, Version 4.2.
- OBERLAENDER, M., DE KOCK, C. P. J., BRUNO, R. M., RAMIREZ, A., MEYER, H. S., DERCKSEN, V. J., HELMSTAEDTER, M., AND SAKMANN, B. 2011. Cell Type-Specific Three-Dimensional Structure of Thalamocortical Circuits in a Column of Rat Vibrissal Cortex. *Cerebral Cortex* .
- OWENS, J. D., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J. E., AND PHILLIPS, J. C. 2008. GPU computing. *Proceedings of the IEEE* 96:879–899.
- SENGUPTA, S., HARRIS, H., AND GARLAND, M. 2008. Efficient parallel scan algorithms for GPUs. Technical Report NVR-2008-003, NVIDIA, Santa Clara, CA.
- STONE, J. AND GOHARA, D. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12:66–72.
- VEKTERLI, T. 2009. Parallelization of Artificial Spiking Neural Networks on the CPU and GPU. Master’s thesis, Norwegian University of Science and Technology.
- YUDANOV, D. 2009. GPU-based implementation of real-time system for spiking neural networks. Master’s thesis, Rochester Institute of technology.

YUDANOV, D., SHAABAN, M., MELTON, R., AND REZNIK, L. 2010. GPU-Based Simulation of Spiking Neural Networks with Real-Time Performance & High Accuracy. *Computational Intelligence* pp. 18–23.

*Declaration of interest: The authors report no conflicts of interest. The authors alone are responsible for the content and writing of the paper.*