Published in final edited form as:
Neurocomputing. 2018 May 2; 302: 55–65. doi:10.1016/j.neucom.2018.04.007.

Dynamic parallelism for synaptic updating in GPU-accelerated spiking neural network simulations

Bahadir Kasap and **A. John van Opstal**[*]
Donders Centre for Neuroscience, Department of Biophysics, Radboud University, Heyendaalseweg 135, HG00.831, 6525 AJ, Nijmegen, The Netherlands

Abstract

Graphical processing units (GPUs) can significantly accelerate spiking neural network (SNN) simulations by exploiting parallelism for independent computations. Both the changes in membrane potential at each time-step, and checking for spiking threshold crossings for each neuron, can be calculated independently. However, because synaptic transmission requires communication between many different neurons, efficient parallel processing may be hindered, either by data transfers between GPU and CPU at each time-step or, alternatively, by running many parallel computations for neurons that do not elicit any spikes. This, in turn, would lower the effective throughput of the simulations. Traditionally, a central processing unit (CPU, host) administers the execution of parallel processes on the GPU (device), such as memory initialization on the device, data transfer between host and device, and starting and synchronizing parallel processes. The parallel computing platform CUDA 5.0 introduced dynamic parallelism, which allows the initiation of new parallel applications within an ongoing parallel kernel. Here, we apply dynamic parallelism for synaptic updating in SNN simulations on a GPU. Our algorithm eliminates the need to start many parallel applications at each time-step, and the associated lags of data transfer between CPU and GPU memories. We report a significant speed-up of SNN simulations, when compared to former accelerated parallelization strategies for SNNs on a GPU.

Keywords

Spiking neural network; Simulations; Graphical processing unit (GPU); Dynamic parallelism

1 Introduction

1.1 Neurocomputing on GPUs

Early GPUs were initially developed and produced for computer graphics, and in particular for video processing and computer gaming. They were built to maximize the device throughput by computing the same function on large quantities of data in parallel. GPUs can speed up computations by running a single instruction on multiple data points simultaneously (SIMD). As such, GPUs have been shown to accelerate computationally

[*]Corresponding author. j.vanopstal@donders.ru.nl (A.J. van Opstal).
Prof. Duan Shukai

demanding complex problems, ranging from game physics to computational biophysics [1]. Theoretical neuroscientists have exploited the use of general purpose computing on GPUs, in neural field model computations and spiking neural network simulations [2]. Using GPUs as vector processors has recently been adopted for SNN simulators, in order to speed up large-scale simulations, such as NeMo [3], NCS6 [4], and GeNN [5]. However, the advancements in general purpose GPU computing is not yet fully adopted by these simulators.

Time-driven SNN simulations follow a simple routine at every time-step that can be broken down into three major steps: (i) state update, (ii) spike thresholding, and (iii) spike propagation. The state update changes the time-dependent variables of all neurons in the network, according to a set of differential equations, in which each neurons membrane potential is computed on the basis of its internal dynamics, synaptic inputs and externally applied currents. Spikes are detected from the updated membrane potentials: if a neurons membrane potential exceeds its spiking threshold, it is reset to its resting state, and a spike event is stored in memory. The spike-propagation step calculates the post-synaptic effect of each spike on the connected neurons. Usually, this step is implemented by a weight-matrix multiplication to the synaptic input values of the post-synaptic neurons.

Parallel computing can vastly accelerate the calculations, when these processes are carried out simultaneously. In the optimal scenario, the calculated variables are independent of each other. For SNN simulations, the membrane-potential update and the spike thresholding steps are so-called embarrassingly parallel problems. The state-update and thresholding functions (*kernels*) can therefore readily run in parallel for individual neurons with different input values or parameters that specify each neurons biophysical properties. However, synaptic communication across the network is considered to be the bottleneck in parallelization [6,7], as it requires a pass through all synapses of the network to update the effect of spikes on the post-synaptic neurons.

Different parallelization strategies have been designed for GPUs (reviewed in [8]) with the aim to vectorize these calculations: across neurons [9], or across spikes and synapses [3]. However, these strategies all have in common that they run many obsolete operations in each time-step, as they typically include also the silent (non-spiking) neurons in the network. Especially, since spikes are relatively infrequent events compared to the size of the network and to the number of time-steps, most computations in existing algorithms introduce substantial additional idle time that merely keeps the computing cores busy. The same problem exists in spiking neural network simulations on other parallel computing architectures [10].

This problem has partly persisted as a result of technical limitations in GPU programming. General-purpose GPUs have become common for large-scale computational problems. Yet, they pose limitations on the implementation of parallel algorithms as a consequence of the hardware architecture. In particular, memory-handling on the GPU differs from the serial applications that run on the central processing unit (CPU). As GPUs have their own memory, they require that all the data, used for the computations, are available on the devices memory. Even though the GPU (device) parallelizes the computations, the CPU (host) manages the applications, such as the data transfer between host and device, memory initialization on the device, the initiation of new parallel processes, and the synchronization between parallel processes. This task requires either the device-to-host memory transfer of the spiking neuronsâ;; indices at each time-step, and to initiate the synaptic update kernels for spikes, or to check all synapses in the network to update the spike effects, in case there was a presynaptic spike.

CUDA (Compute Unified Device Architecture) allows the implementation of dynamic parallelism [11], which allows a CUDA kernel to create nested parallel processes on the GPU. When applied to a SNN, this would allow the start of a new parallel operation to update the synaptic values of post-synaptic neurons, only if a neuron emitted a spike. In this way, it would potentially speed-up the simulations, by eliminating idle calculations.

Here, we test an implementation of dynamic parallelism, applied to spike propagation across a SNN. We demonstrate a significant speed-up from dynamic parallelism in a pulse-coupled network of Izhikevich neurons [12]. The network consists of randomly connected excitatory and inhibitory neurons, which are driven by stochastic input. The same network has recently been used as a benchmark to test the SNN simulator GeNN [5] on different GPU devices.

1.2 Parallel computing on GPUs

A GPU comprises of a GPU chip, and a synchronous graphics RAM (SGRAM, Fig. 1A). The GPU chip contains organized sets of streaming multiprocessors, coupled with on-chip registers and read-only texture memories that are private to each processor. The shared memory can be read and written by all processors belong to the same multiprocessor. The SGRAM is used for processor-specific local memory, and for global memory to which each processor has access rights. The access speed and allotted size of these memories will differ. While global memory has the largest space, it has the narrowest bandwidth. Yet, the host can only access the global memory on the SGRAM.

General purpose GPUs typically use C-language programming with application programming interfaces (APIs). Commonly used APIs are NVIDIA CUDA and OpenCL. Here, we will focus on CUDA terminology, for consistency. CUDA provides a set of extension functions to allow the programmer to use computing and memory resources of the GPU. These helper functions allow programmers to allocate memory on the device, transfer memory between device and host, and manage the parallel execution of kernels written in C ++.

Parallel computing follows SIMD parallelism (single-instruction multiple data points), where the individual processors run the same instructions on different data points. The instruction code is termed a *kernel*, as it is the building block of a parallel application. A kernel executes its code simultaneously across a set of parallel threads. A threading structure consists of the arguments and data addresses on the device that will be used by the kernel, and determines a hierarchy of grids of blocks (Fig. 1B) that run in parallel. Each *thread* runs the same kernel, with its unique id, which is used to access and manipulate unique elements in an array or matrix. A *thread block* is a set of threads that can cooperate through barrier synchronization and access a shared memory (private to that block). A *grid* is a set of thread blocks that can be executed independently, and only share access to the global memory. Different thread blocks can be executed independently, in arbitrary order. However, within each block, 32 threads (*warps*) run in parallel, and multiprocessors regulate their execution. When a warp is stuck, the multiprocessor can quickly switch to another available warp to reduce idle time. The GPU scheduler will map the thread blocks onto the multiprocessors, based on the threading structure, and it maintains task efficiency by keeping busy as many cores as possible at any given time [13].

These conceptual differences introduce new challenges that affect programming style. For instance, a *race condition* arises when concurrent threads need to write to the same memory address. Hypothetically, both may read the same value at the same time, do their own computations on the data, and write one after another to the same location (Fig. 1C). In this case, the result from the thread that wrote last will survive, and the computations by earlier threads will be discarded, leading to erroneous results. Such conflicts should be foreseen during code development; writing into a given memory location should thus be sequenced. CUDA API provides atomic operations and memory locks to handle such often-encountered programming problems.

Coalesced memory access refers to combining multiple memory accesses into a single transaction (Fig. 1D). When data is organized in the global memory such that the concurrent threads in a warp access contiguous memory locations, then, the whole chunk of memory can be called at once for all threads in a warp. While on earlier GPUs the computing capabilities required aligned and sequential memory calls from a warp (128 bytes for 32 threads), for coalesced memory access, compute capability 3.0 also supports non-sequential accesses if they are aligned (Fig. 1D, bottom). Unaligned access patterns do not benefit from memory coalescing for efficient memory calls. Further, coalesced memory access may not always be applied for all algorithms, while un-coalesced access may not be critical for enhanced performance. Yet, especially the algorithms that require repetitive memory accesses will benefit from coalesced memory accesses to improve performance.

In the optimal scenario, (1) the calculated variables would be independent of each other, (2) the data size handled by each processor, and the computational load on the functions (kernels) that process the data, would be balanced, (3) memory access within the device, and memory transfers between device and host would be optimized.

1.3 Parallel synaptic updating schemes

We propose a novel parallelization strategy, which utilizes dynamic parallelism for synaptic updating in SNN simulations. To evaluate performance of our algorithm, we compared it to two earlier applied parallel updating algorithms (Fig. 2): (1) parallelization across neurons, in which the synaptic currents are calculated for individual neurons in parallel [N-algorithm; Fig. 2A; 9,14], and (2) parallelization across synapses, which updates the synaptic currents for each synapse in parallel [S-algorithm; Fig. 2A; 15]. In contrast, our new algorithm updates all post-synaptic currents for each action potential in parallel (AP-algorithm; Fig. 2B). We compared the performance of the three algorithms for different network sizes, by varying the number of neurons (N), and the number of synapses per neuron (S) in the network, and for different spiking regimes, by varying the activity states in the networks.

Each algorithm updates the neural states by N threads. Based on the total current acting on a neuron at a time-step, membrane potential is updated by the differential equation describing the neuron model. If the membrane potential crosses spiking threshold, the spike is recorded to be propagated to the postsynaptic connections. While N- and S-algorithms update synapses at a separate step after state updates are finished for all the neurons, AP-algorithm starts nested processes (Fig. 2). This paradigm difference already decreases computation time, because the neural state update computations must be completed for all neurons to continue with spike propagation in N- and S-algorithms. The threads which complete their calculations earlier wait for the rest of the threads to finish. Therefore, synchronization between neural state update and synaptic update steps hurts throughput. However, the main novelty of the AP-algorithm is the use of dynamic parallelism for spike propagation and decreasing number of running threads per time-step.

Both the N-algorithm and the S-algorithm parallelize the matrix multiplication for synaptic updates. They both calculate an update for each existing synapse in the network (Fig. 2A). The N-algorithm starts N threads (across neurons), which each iterate over S synapses to update postsynaptic currents for the neurons that elicit a spike. The S-algorithm recruits $N \times S$ threads (across synapses), which each updates the postsynaptic current if there was a presynaptic spike. It is apparent that these two algorithms allot the work in different ways to individual threads. Yet, both algorithms check if there was a presynaptic spike at a connection, and update the postsynaptic current for each synapse with a presynaptic spike. The difference is; the N-algorithm updates the postsynaptic currents with fewer threads, but with more computations per thread, when compared to the S-algorithm. Therefore, the computation duration increases with the number of synapses.

The AP-algorithm combines neuron state update and postsynaptic update steps. It utilizes dynamic parallelism to update all postsynaptic currents from a neuron, whenever it produces an action potential. Each time a neurons membrane potential crosses the spiking threshold, a new set of children threads are triggered (Fig. 2B). Postsynaptic updates are delivered by S threads, each updating one synaptic end. Therefore, the number of spikes become the main determinant of the number of calculations to be done. AP-algorithm starts $S \times$ (# of spikes) threads in total per time-step. Each thread updates a postsynaptic current as in S-algorithm. Compared to the N- and S-algorithms, the AP-algorithm combines spike thresholding with synaptic updating, and thus eliminates the overhead synchronization delays as well. AP-algorithm executes synaptic updates as the spikes occur.

We will demonstrate that each algorithm will have its own optimal performance conditions. As we define algorithm performance by the computation time needed to update the postsynaptic currents, the fastest algorithm is considered the best. The execution time of each time-step is determined by two factors: (i) the time needed for a thread to complete its task, and (ii) the occupancy of GPU multiprocessors. A threads runtime depends on the computational load of its kernel; when a kernel must perform many calculations and memory accesses per time-step, it increases processing time. The occupancy of GPU multiprocessors deduces to how well the task is distributed over the streaming cores to increase throughput. Since the threads are mapped onto the multiprocessors by the GPU scheduler, the more threads there are, the longer it takes for the network to finish.

2 Methods

2.1 Network architecture

Performance of the three different algorithms was tested on a SNN that consisted of pulse-coupled Izhikevich neurons, which were driven by stochastic input [12]. The change in each neurons membrane potential is updated by the following differential equation:

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

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

where x' designates the time derivative of x, $v(t)$ is the cells membrane potential, $u(t)$ is the so-called recovery variable, $I(t)$ is the external (stochastic) input; parameter a (in s^{-1}) is the recovery time scale, and b (dimensionless) is the recovery sensitivity to sub-threshold fluctuations of the membrane potential. A neuron emits a spike if its membrane potential crosses its spiking threshold (here set to $v = 30$). At the next time step, the membrane potential, v, is reset to its resting value, c, and the recovery variable, u, is increased by a spike-triggered recovery reset, d:

$$\text{when } \; v > 30: \;\; v = c \;\; \text{and} \;\; u = u + d \quad (3)$$

Because the recovery variable, u, acts on the membrane potential change, v', as an inhibitory current (Eq. (1)), its increase is bounded by the internal dynamics of the neuron. When u reaches high values, the neuron will be hyperpolarized and it will require more synaptic input to elicit another spike. While the neuron is silent, u will decay exponentially (following Eq. (2)).

The input current, $I(t)$, for each neuron in the network consists of two sources: a stochastic input current, and the synaptic currents that it receives from active presynaptic neurons. Formally:

$$I_j(t_{n+1}) = g_{\text{exc,inh}} \cdot q_j(t_n) + w_s \sum_i^S S_{ij} \delta_i(t_n) \quad (4)$$

where q_j is a random input to the neuron scaled by an excitatory or inhibitory conductance $g_{\text{exc, inh}}$, which determines the networks activity state (either quiet, balanced or irregular, see below). The total synaptic current is determined by summation over the connectivity matrix elements S_{ij} from neuron i to j for all presynaptic neurons i that have elicited a spike $\delta_i(t_n)$ at the previous time step ($\delta_i = 1$ if there was a spike at t_n, and 0 otherwise). w_s is a fixed synaptic scaling factor that modulates the synaptic input current based on the total number of synapses in the network. Note that N-S randomly selected entries in the connectivity matrix, S_{ij}, had been set to 0 for each input neuron *i*; the remaining S entries were drawn at random from a uniform distribution for the excitatory and inhibitory neurons.

In the simulations, we varied the number of neurons, *N*, and the number of synaptic connections, *S*, per neuron, with $S \leq N$, to compare the performance of the three algorithms for different network sizes and activity states. While the neuronal parameters (*a, b, c, d*) determine the spiking regimes of the individual neurons, the network dynamics are configured by the randomly distributed input conductances, $g_{\text{exc, inh}}$; the synaptic scaling, w_s, ensures that the activity of each neuron remains stable for different numbers of input synapses.

To set up the network, the initial values of the neuronal variables (*v, u*), the neural parameters (*a, b, c, d*), and the connectivity strengths, S_{ij} were selected at random [12, see Table 1]. Excitatory cells were tuned for regular spiking and bursting activity with $(a, b) = (0.02, 0.2)$ and $(c_i, d_i) = (-65, 8) + (15, -6)\, r_i^2$, where r_i is a random variable, uniformly distributed on the interval [0,1]. $r_i = 0$ corresponds to a regular spiking regime, whereas $r_i = 1$ corresponds to a bursting cell. Taking r_i^2 (instead of, e.g., $|r_i|$) introduces a bias towards regular spiking neurons in the network. Inhibitory cells, on the other hand, were tuned by parameters $(a_i, b_i) = (0.02, 0.25) + (0.08, -0.05)$ and $(c, d) = (-65, 2)$. Therefore, inhibitory neurons are fast spiking (fast-recovery with $a = 0.1$ for $r = 1$) and low-threshold spiking (with $b = 0.25$ for $r_i = 1$). In this way, we constructed a heterogeneous network, with different dynamics for each neuron.

In the default network ($N = 2500$), all neurons were connected to $S = 1000$ randomly selected postsynaptic neurons, and the $N \times S$ values in S_{ij} were initialized randomly from a uniform distribution on [0, 0.5] for excitatory neurons and on [−1, 0] for inhibitory neurons. The ratio of excitatory to inhibitory neurons was kept fixed at 4:1, when we varied the total number of neurons in the network (default SNN: 2000:500). For varying numbers of synapses, we scaled the connectivity matrix with the factor $w_s = 10^3/S$, in order to keep the total input strength to the postsynaptic neurons (Eq. (4)) constant, and having the default $w_s = 1$ for $S = 1000$ synapses [12].

At each time-step, each neurons membrane potential was calculated by Eq. (1) based on its input current and internal state. The randomly selected input currents were drawn from a uniform distribution on the interval [0, 1] and scaled by $[g_{\text{exc}}, g_{\text{inh}}]$ for excitatory and inhibitory neurons, respectively. Different $[g_{\text{exc}}, g_{\text{inh}}]$ values result in different firing regimes in the network. $[g_{\text{exc}}, g_{\text{inh}}] = [2.5, 1.0]$ for quiet networks, [5.0, 2.0] for balanced networks, and [7.5, 3.0] for irregularly firing networks [5].

2.2 Parallelization algorithms

The pseudo-codes for the three algorithms are provided below. The state update and thresholding steps were kept identical for all three algorithms. All simulations were performed on a Tesla K40 GPU; the code is made available as open access under https://bitbucket.org/bkasap/dynamicparallelismsnn.

```
start timer
start a thread for each neuron $i$:
    update state variables:
        $V_i(t_{n+1})$ based on $V_i(t_n)$, $u_i(t_n)$ and $I_i(t_n)$
        if $V_i(t_{n+1}) > V_\theta$:
            add $i$ to spike list
synchronize: wait until all threads are finished, and ensure that the spike list is complete
(1) N-algorithm
start a thread for each presynaptic neuron $i$:
    for each postsynaptic neuron $j$ (sequentially over $S$ synapses):
        if there is a spike:
            update $I_j(t_{n+1})$ by $S_{ij}$
synchronize: wait until all threads finish their calculations to proceed to the next time-step
end timer for N-algorithm
(2) S-algorithm
start a thread for each synapse $ij$:
    if there is a spike from the presynaptic neuron $i$:
        update $I_j(t_{n+1})$ by $S_{ij}$
synchronize: wait until all threads finish their calculations to proceed to the next time-step
end timer for S-algorithm
```

```
(3) AP-algorithm
start timer
start a thread for each neuron $i$:
    update state variables:
        $V_i(t_{n+1})$ based on $V_i(t_n)$, $u_i(t_n)$ and $I_i(t_n)$
        if $V_i(t_{n+1}) > V_\theta$:
            add $i$ to spike list
            start a thread for each postsynaptic neuron $j$:
```

```
        update $I_j(t_{n+1})$ by $S_{ij}$
    *synchronize:* wait until all threads finish their calculations to proceed to the next time-step
    *end timer for AP-algorithm*
```

2.3 Quantifying performance

To quantify algorithm performance, we calculated the total simulation duration, as function of the three different spiking regimes (quiet, balanced, and irregular), spike-propagation algorithm (the N-, S-, or AP-method), and the total number of spikes emitted in the simulation for different network sizes (N: the number of neurons. S: the number of synapses per neuron). The execution times of the time-steps are measured by the time-stamp differences between the start of the state update calculations, until all synaptic currents in the network have been calculated for the next time-step. Even though the neurons were driven by stochastic input, for a given number of neurons and spiking regime, the total number of spikes was fixed. Therefore, a direct comparison is possible between the algorithms by considering their throughput as the number of spikes processed within a millisecond.

3 Results

Fig. 3 depicts the network dynamics of an SNN containing N=2500 neurons for the three different activity regimes: quiet (Fig. 3A), balanced (Fig. 3B) and irregular (Fig. 3C) firing. These regimes were obtained by modulating the random input currents, as specified in Table 1. Each neuron in the network had 1000 randomly assigned synapses (S), with the ratio of excitatory and inhibitory connections set as 4 to 1.

In the quiet regime (A), the network was silent for the majority of time steps. The entire network elicited only 194 spikes during a full second of neural simulation. In that scenario, the execution time for the N-algorithm (D) depends only on the existence of a spike at a given time-step. Whenever a neuron spikes, the N-algorithm (Eq. (3) and Table 1) starts N threads, each of which passes sequentially through the S postsynaptic neurons. Therefore, this algorithm is the slowest of the three when there is a spike, taking about 2.2 ms to complete the cycle. For the balanced and irregular firing regimes (B and C), the execution times of a simulation time-step are not affected for the N-algorithm. In case of many active neurons at a given time-step (as in E and F), the individual threads run sequentially over distinct postsynaptic connections. Thus, the time spent to update the postsynaptic neuronsâ current inputs remain the same in the first few milliseconds of high-intensity neuronal firing. When there are no spikes at a time step, as for most time-steps in the quiet regime, or during the silent period after the initial high firing rate in the irregular regime (e.g., between 25–90 ms in Fig. 3C and F), this algorithm takes around 0.8 ms to complete a cycle.

The S-algorithm starts an application with a higher number of parallel processes (given by $N \times S$). Each thread works on an individual synapse and updates the input current of its postsynaptic neuron, whenever the presynaptic neuron is active. Thus, also this algorithm is insensitive to the number of spikes at a given time-step (light-blue lines). Rather, it is bounded by the number of parallel processes that a GPU device can handle simultaneously. In all three firing regimes, the execution time of the S-algorithm stabilizes at around 0.5 ms per time-step.

The AP-algorithm (dark-blue lines) initiates a parallel application with S threads to update the current input to all postsynaptic neurons, whenever a neuron elicits a spike (Fig. 2C). Each thread in this case will update a postsynaptic neurons input current. The total execution time of this algorithm is most sensitive to the number of active neurons, as compared to the other two algorithms, as each spike will trigger a new parallel process. Yet, taken together, the execution times of each time-step for the three different algorithms show that the dynamic parallelism algorithm is overall the fasted method for spike propagation in the SNN, under all three regimes. The differences become also pronounced in the total execution times.

We noted that the execution times could fluctuate substantially, depending on the number of spikes at a time-step. In the balanced firing regime, between 30 and 50 ms simulation time (in Fig. 3B and E), both the N-algorithm and the AP-algorithm take longer than their mean execution time. Especially, in the beginning of the simulation for irregular firing, where half of the neurons resulted to be active between 0 and 25 ms (in Fig. 3C and F), even the S-algorithm took a longer time. However, this initial high activity in the irregular firing regime seems to be atypical. It is not realistic to have more than half of the neurons active at a time-step in spiking neural network simulations. We therefore investigated the underlying cause for this high-level transient activity through a phase-plot analysis of the temporal dynamics of a single neuron in the network.

Fig. 4 shows the dynamics of an excitatory neuron at the first 300 ms of simulation in the irregular firing regime. The transient high firing activity of the network (Fig. 3C) results from a high discharge of the neurons in the beginning of the simulation, which resulted to be due to the stochastic input to individual neurons. The temporal dynamics of the three neuronal variables (Fig. 4A) show that the initial burst at the start of the simulation results from the high synaptic current input to the network. Nullclines of the neuron model for $u(t)$ and $v(t)$ are shown in Fig. 4B. These nullclines (defined by $v' = 0$ and $u' = 0$, Eq. (1) and (2)) intersect at the stable and unstable fixed points of a dynamical system, and describe how the state variables would evolve at a given state. The neurons stable point (resting state) lies at $(u, v) = (-68, -15)$ where the u- and v-nullclines intersect below the reset value of the membrane potential, c. The nullclines depict a snapshot of the neurons dynamics at a given time for $I = 0$. However, the external current input is also a time-dependent variable, $I(t)$, and is directly added to the membrane potential change, v' (Eq. (1)), at each time-step. As a result, the input current shifts the v-nullcline along the u-axis. For a positive input current, the v-nullcline shifts upwards, and the nullclines can lose their intersection points for sufficiently high current values. In this way, the system can become unstable, and the membrane potential starts to increase towards the spiking threshold. With the increase in the recovery variable, $u(t)$, and the associated membrane-potential reset after each spike, the neurons state variables follow a trajectory in the phase plane shown by the blue dots (connected by thin lines for spikes and by dashed lines for decay to the resting potential).

With the initial high rise in the input current, the v-nullcline shifts upward and drives the neuron into a repetitive firing state. Therefore, the first 5 spikes are accompanied by an increased recovery variable, u, within the first 25 ms of the simulation. The high firing rate is followed by a decay of u to its resting value, and the following spikes, which occur at irregular intervals, do not increase u as much as during the initial transient firing. For repeated simulations with different initial parameters in this firing regime the transient high-frequency bursts re-occurred each time, but they were absent if simulations were continued after a deliberate interruption. Thus, the high firing rates at the start of the simulations result from the initial network configuration, rather than from an interesting network effect. In what follows, we therefore discarded this abnormal, transient firing pattern at the start of the simulation, when quantifying the performance of the different algorithms.

Fig. 5 quantifies the performance of the three algorithms for SNNs with two different numbers of synapses (S=1000 and S= 2000 outbound synapses) per neuron, as function of network size (number of neurons, N, from $2.5 \cdot 10^3$ to $5 \cdot 10^5$ neurons). For all three algorithms and firing regimes, the simulation runtime increases with the number of neurons, albeit at different rates. The N-algorithm takes relatively longer for networks with fewer neurons (below $N = 2.5 \cdot 10^4$) and starts to be slower with increasing N, in the same way as the S-algorithm (i.e. according to a power law), for the balanced and irregular firing regimes. For the quiet firing regime, the N-algorithm is faster than the S-algorithm for $N > 10^4$ neurons. Note that the simulation execution times are not affected by the different firing regimes, for either the N-algorithm, or the S-algorithm. In the quiet regime, the N-algorithm outperforms the S-algorithm, since it is faster when there are no spikes at a given time step (Fig. 3D). However, both algorithms become slower with increasing number of synapses per neuron. In contrast, the AP-algorithm is insensitive to the variation in S, but is strongly affected by the spike count, as it starts new parallel child processes for each spike. Yet, up to networks with $N = 2 \cdot 10^5$, the AP algorithm outperforms the other two computational schemes, when they are densely connected (high S). When the number of processes exceeds the capacity of the GPU, they have to wait for each other to complete, which will increase the simulation time, also for the AP algorithm.

In Fig. 6 we compared the performance of the three algorithms (their throughput, and mean execution time) as function of the number of synapses per neuron (between S=256 and 8192), under the three activity regimes, for networks with $N = 10^4$ (blue) and 10^5 (green) neurons, respectively, and for a total neural simulation of 5 s. We ensured that the spike counts and neural dynamics of the networks did not vary with the number of synapses, by keeping the total synaptic current fixed in the network. This was achieved by scaling the range of the uniform weight distributions according to $w_S = 1000/S$, which also ensured that the neural dynamics of the network remained unaffected. Only for networks with a few synapses per neuron some fluctuations in the spike counts may be expected, since the post-synaptic effect of the spikes, and the associated effects of the stochastics, on the postsynaptic currents will be coarser.

The simulations in Fig. 6 show that for the AP algorithm the mean execution time per time-step, and total simulation duration were independent of the number of synapses. In contrast, this performance indicator increased steadily with S for the N- and S-methods. Note, that since we kept the spike count fixed for the different configurations, the throughput (top panels) is inversely related to the simulation duration. Taken together, the AP algorithm outperformed the N- and S-algorithms for the smaller networks under all conditions. As the networks grew in size, the AP-algorithm resulted to outperform the other two algorithms for highly connected networks (large *S*).

As our goal was to speed-up the SNN simulations through parallelism, we considered the fastest algorithm for a given simulation condition (determined by the number of neurons, synapses, and spikes) as the winner for that condition. Figs. 5 and 6 indicate that none of the three algorithms wins for all simulation conditions. To provide an overview of the optimal conditions for each algorithm under a wide range of network settings, we varied both the number of the neurons, and the number of synapses per neuron in the network, and simulated the networks for the three different firing regimes (Quiet, Balanced and Irregular). For each (*N, S*) bin we then determined the fastest algorithm, and assigned the associated winners color code at that bin. Fig. 7 shows the results. From these simulations, it is clear that in the quiet regime (Fig. 7A), the AP algorithm performs best, regardless the network size and its connectivity. In line with the simulations shown in Figs. 5 and 6, the AP method is the most efficient algorithm for sparse spiking activity, because it does not trigger the synaptic updating computations when there are no spikes. But when the network activity increases, as in the balanced and irregular network states (Fig. 7B and C), the AP algorithm outperforms the N and S methods especially for the highly-connected networks. In contrast, the N-method is the winner for large networks with relatively sparse connectivity (up to S=1024 synapses/neuron in Fig. 7B, and up to S=2048 synapses/neuron in Fig. 7C, for networks with N=250,000 neurons), whereas the S-method best suits small and sparsely connected networks. As the S-algorithm requires more threads to be completed at each cycle to update synaptic currents (Fig. 2A), the device queues their execution, and start a new batch each time the processors finish their calculations. This introduces additional overhead, because each thread should access memory even when the computation is cheap (in this case, only addition). For the same reason, also the AP-algorithm is hindered by high spike counts per time-step. Instead, the N-method runs fewer threads, as each thread loops over *S* synapses. As a result, the N-algorithm performs best for lower *S*, although its performance is sensitive to the computational load. Thus, if more calculations per synapse were to be required, the AP algorithm would outperform the N algorithm also in these cases. This happens, for example, when synaptic plasticity would be included in the network, as such a mechanism would require additional calculations to account for the synaptic dynamics at each updating time step.

4 Discussion

In this paper, we quantified the performance of three different parallelization algorithms for the simulation of spike propagation within spiking neural networks on a GPU. We showed that the simulation runtimes were highly susceptible to the number of synapses for simulations with the N- and S-algorithms, whereas the spike count was the prominent determinant of simulation runtime for the AP-algorithm. As a result, the AP-algorithm outperforms the other two algorithms when the spike occurrence is sparse in relation to the network size (the total number of neurons and synapses), and to the number of simulation time-steps.

We employed a network architecture of pulse-coupled Izhikevich neurons for the SNN simulations [using the same implementation on CUDA as in 12], because this approximate network model allows for easy scalability by varying the number of neurons (N) and synapses (S), while preserving sufficient complexity and variation of different neural states within the network, and easy control of the total spike counts.

However, the simulations had a relatively poor time-resolution (time-steps at 1 ms intervals), while at the same time this simple neuronal model had already been computationally optimized [12] to explain a variety of complex physiological behaviors of neurons under different input and biophysical conditions. The network is thus able to capture different states of synchrony within populations of randomly connected neurons (as coupled nonlinear oscillators).

Note that alternative neural models, which require much higher time precision, will result in many more computations per thread for the neural-state updating steps. This would happen, for instance, when the research question demands more computations per time-step, by including ion-channel-specific computations as in Hodgkin–Huxley model neurons [4,16], or when considering current propagation through geometrically complex dendritic trees [17,18]. Such architectures and models would require more computations per time-step simply because of the increasing complexity of the models to update neural states or synaptic propagation. Accounting for spike-time-dependent plasticity [19], or when modeling the high-frequency bursting behavior of neurons in the midbrain Superior Colliculus [20,21] would also require additional computations or fine-grained time resolutions, and thus more computations and performance. Also the new class of evolving SNNs require additional computations per time-step [22] and multiple network classes. As long as the spike propagation follows delivery of discrete pulses to a subset of the all neuron population in the network, dynamic parallelism would accelerate GPU based simulations. Because, also under these more demanding dynamic requirements, spikes would be elicited more sparsely during the whole simulation. Because the AP-algorithm eliminates the need to compute synaptic updates for neurons that do not elicit a spike, it will readily speed-up such more demanding simulations. However, this is only valid for spiking neural network implementations. Most of the other neural network modelling frameworks for deep neural networks and machine learning applications are already utilizing GPUs (Torch [23], Tensorflow [24], supported by CUDA cuDNN library in the backend talking to GPU devices [25]).

We explored the idea of dynamic parallelism for synaptic updating in SNN simulations, by comparing its performance to the two parallelization strategies that are currently available in the literature. However, it should be noted that the actual simulation durations for all three algorithms were longer than reported here because of the considerable time needed for the random number generations, and memory transfers prior to, and following the main simulation loops. The generation of random numbers to initialize the neural parameters and their connectivity within the network introduced considerable latencies, and depended strongly on the number of neurons and synapses in the network. Furthermore, the random number generators that were used for each time-step to provide the time-varying stochastic input current to each neuron, occupied a large portion of the device memory. However, since here we focused on performance differences between the three algorithms, we merely considered the execution time of each time-step from the start of the state updates until all synaptic currents had been calculated for the next time-step.

Our proposed algorithm can readily speed up the computer simulations on GPU where the spike propagation is the limitation factor. Also, the simulation code can be further improved by optimizing the use of device memory during the simulations. However, in this simple network implementation, the comparative performance of the different algorithms would not be affected, since an ongoing thread reads the connectivity matrix element, and writes the synaptic input current only once. Using shared memory and coalesced memory access will potentially accelerate the simulations for repetitive computations on the same data point. This would be the case when GPUs are used to speed-up the neuro-computational simulations with more computations at each synapse updating step, for instance, under synaptic plasticity calculations [19], or for current propagation within complex dendritic tree geometries [17].

For computationally demanding SNN simulations, different GPU-based simulation frameworks have been introduced: CARLsim [26], Nemo [15], NC6 [4], and GeNN [27]. The GeNN simulator was developed to implement different SNN architectures with the least amount of code on a GPU [5]. The simulator contains a code-generation process: the user defines a network model, and specifies the neural parameters by a set of predefined functions, upon which the simulator generates and compiles the associated C++/CUDA code for a GPU. Memory usage and access on the device are optimized for various example cases. The GeNN simulator is independent of the operating system and of the GPU device model, and can also be used to generate C++ code for the same network configuration on CPUs. These characteristics make GeNN a versatile simulation tool. However, it limits the user friendliness in easy extensions with new neuron models, in manually specifying the neural dynamics, or in changing the simulator source code. In addition, the GeNN simulator can be optimized by utilizing dynamic parallelism for its synaptic updates.

All GPU devices produced from 2013 onward support dynamic parallelism as described in this study, and thus allow developers to employ this programming paradigm to overcome various programming problems. In terms of spiking neural network simulations, dynamic parallelism substantially accelerates the massive neural computations, by implementing the spike-triggered calculations at each synaptic updating step. In previous parallel SNN implementations, this step was considered to be the bottleneck of the simulations, because the developed algorithms kept running obsolete calculations for spike propagation, even when the presynaptic neuron did not elicit any spike. Especially, the simulations of densely connected neurons operating under sparse spiking regimes (like observed experimentally in the cerebral cortex, or when simulating the neural dynamics at a high temporal resolution) benefit from the considerable speed up via dynamic parallelism. We therefore foresee that spike propagation will no longer be the major determinant of simulation duration of large-scale dynamic neural networks.

The premise of parallel computing is: parallelization accelerates computations. However, parallelization is only possible if the same exact computations are performed again and again on different data points; and these computations are not dependent on each others results. Modern GPU's can run millions of threads in parallel, therefore millions of neural state update and synaptic update can be parallelized. However, the computations can be parallelized only if the calculations are exactly the same, even if with different parameters. Therefore, N- and S-algorithms require to finish all neural state updates to start synaptic propagation. If the neural network architecture requires many small sets of different neuron types, whose behaviors are defined by different equations, GPU utilization would decrease. That would mean, not many calculations are done in parallel and many processors are waiting to be assigned to a calculation. Such scenario would not optimize throughput, thus the architecture of the network is also a consideration for GPU. For full utilization of GPU in calculations, the number of calculations running in parallel should cover the number of threads started at a parallel block.

Acknowledgments

This work was supported by the EU FP7 Marie Curie program, ITN project NETT (nr. 289146, BK), and by the EU Horizon 2020 program, ERC Advanced Grant, project ORIENT (nr. 693400, AJVO, BK). The Tesla K40 used for this research was donated by the NVIDIA Corporation.

Biographies

A. Bahadir Kasap received his B.Sc. degree in Physics from Koc University, Istanbul, Turkey in 2009, and his M.Sc.degree in Computational Sciences in Engineering from the Technical University of Braunschweig, Germany in 2012. Since then, he is a Ph.D. Candidate in computational neuroscience at the Donders Institute for Brain, Cognition and Behavior, Radboud University in Nijmegen, the Netherlands. His research focuses on command generation for eye movements in a spiking neural network model of the midbrain superior colliculus. He recently started working as a Data Scientist at Sogeti.

John van Opstal is professor of biophysics at the Donders Institute for Brain, Cognition and Behavior, Radboud University in Nijmegen, the Netherlands. His research interests lie in sound processing and localization, plasticity in hearing, multisensory integration, and the role of the superior colliculus in saccadic eye-head gaze shifts. He acquired a EU-FP7-IDP grant for HealthPAC (2013) and a personal ERC Advanced Grant ORIENT (2016). He published his book "The Auditory System and Human Sound-Localization Behavior" in 2016.

References

[1]. Owens J, Houston M. GPU computing. Proc IEEE. 2008; 96(5):879–899.

[2]. Baladron Pezoa J, Fasoli D, Faugeras O. Three applications of GPU computing in neuroscience. Comput Sci Eng. 2012; 14(3):40–47.

[3]. Fidjeland A, Shanahan M. Accelerated simulation of spiking neural networks using GPUs. Proceedings of the 2010 International Joint Conference on Neural Networks (IJCNN); 2010. 18–23.

[4]. Hoang RV, Tanna D, Jayet Bray LC, Dascalu SM, Harris FC Jr. A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. Front Neuroinform. 2013 Oct. 7:19. [PubMed: 24106475]

[5]. Yavuz E, Turner J, Nowotny T. GeNN: a code generation framework for accelerated brain simulations. Sci Rep. 2016; 6(November 2015):18854. [PubMed: 26740369]

[6]. Brette R, Goodman DFM. Simulating spiking neural networks on GPU. Network. 2012; 23(4): 167–182. [PubMed: 23067314]

[7]. Zenke F, Gerstner W. Limits to high-speed simulations of spiking neural networks using general-purpose computers. Front Neuroinform. 2014 Sep.8:76. [PubMed: 25309418]

[8]. Slażyński L, Bohte S. Streaming parallel GPU acceleration of large-scale filter-based spiking neural networks. Network. 2012 Dec.23:183–211. [PubMed: 23098420]

[9]. Nageswaran JM, Dutt N, Krichmar JL, Nicolau A, Veidenbaum AV. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. Neural Networks. 2009; 22(5–6):791–800. [PubMed: 19615853]

[10]. Thibeault CM, Minkovich K, O'Brien MJ, Harris FC, Srinivasa N. Efficiently passing messages in distributed spiking neural network simulation. Front Comput Neurosci. 2013 Jun.7:77. [PubMed: 23772213]

[11]. NVIDIA Corp. Dynamic parallelism in CUDA, TechBrief. 2012:1–3. [accessed 12-May-2017] Available online; https://goo.gl/KhEhve.

[12]. Izhikevich EM. Simple model of spiking neurons. IEEE Trans Neural Netw. 2003; 14(6):1569–1572. [PubMed: 18244602]

[13]. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. AMC Queue. 2008 Apr.6:40–53.

[14]. Mutch J, Knoblich U, Poggio T. CNS: A GPU-Based Framework for Simulating Cortically-Organized Networks. Computer Science and Artificial Intelligence Lab; 2010. 2010-02-26

[15]. Fidjeland AK, Roesch EB, Shanahan MP, Luk W. NeMo: a platform for neural modelling of spiking neurons using GPUs. Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors; 2009. 137–144.

[16]. Hodgkin AL, Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. J Neurophysiol. 1952; 117(4):500–544.

[17]. Gugala K, Figas A, Jurkowlaniec A, Rybarczyk A. Parallel simulation of stochastic denritic neurons using NVidia GPUs with CUDA C. Proceedings of the 18th International Conference on Mixed Design of Integrated Circuits and Systems; 2011. 614–617.

[18]. Garaas TW, Marino F, Duzcu H, Pomplun M. A design for real-time neural modeling on the GPU incorporating dendritic computation. Proceedings of the 5th International Workshop on Artificial

Neural Networks and Intelligent Information Processing; SciTePress - Science and and Technology Publications; 2009. 69–78.

[19]. Yudanov D, Shaaban M, Melton R, Reznik L. GPU-based simulation of spiking neural networks with real-time performance & high accuracy. Proceedings of the 2010 International Joint Conference on Neural Networks; 2010. 1–8.

[20]. Goossens HHLM, Van Opstal AJ. Optimal control of saccades by spatial-temporal activity patterns in the monkey superior colliculus. PLoS Comput Biol. 2012; 8(5):e1002508. [PubMed: 22615548]

[21]. Kasap B, Van Opstal AJ. A spiking neural network model of the midbrain superior colliculus that generates saccadic motor commands. Biol Cybern. 2017; 111(3):249–268. DOI: 10.1007/s00422-017-0719-9 [PubMed: 28528360]

[22]. Schliebs S, Kasabov N. Evolving spiking neural network-a survey. Evol Syst. 2013; 4(2):87–98.

[23]. Collobert R, Kavukcuoglu K, Farabet C. Torch7: A matlab-like environment for machine learning. BigLearn: NIPS Workshop; 2011.

[24]. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, et al. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015 Software available from tensorflow.org.

[25]. Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cuDNN: Efficient Primitives for Deep Learning. 2014:1–9.

[26]. Beyeler M, Carlson KD, Chou TS, Dutt N, Krichmar JL. CARLsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks. Proceedings of the International Joint Conference on Neural Networks; 2015.

[27]. Nowotny T. Flexible neuronal network simulation framework using code generation for NVidia CUDA. BMC Neurosci. 2011; 12(Suppl 1):P239.

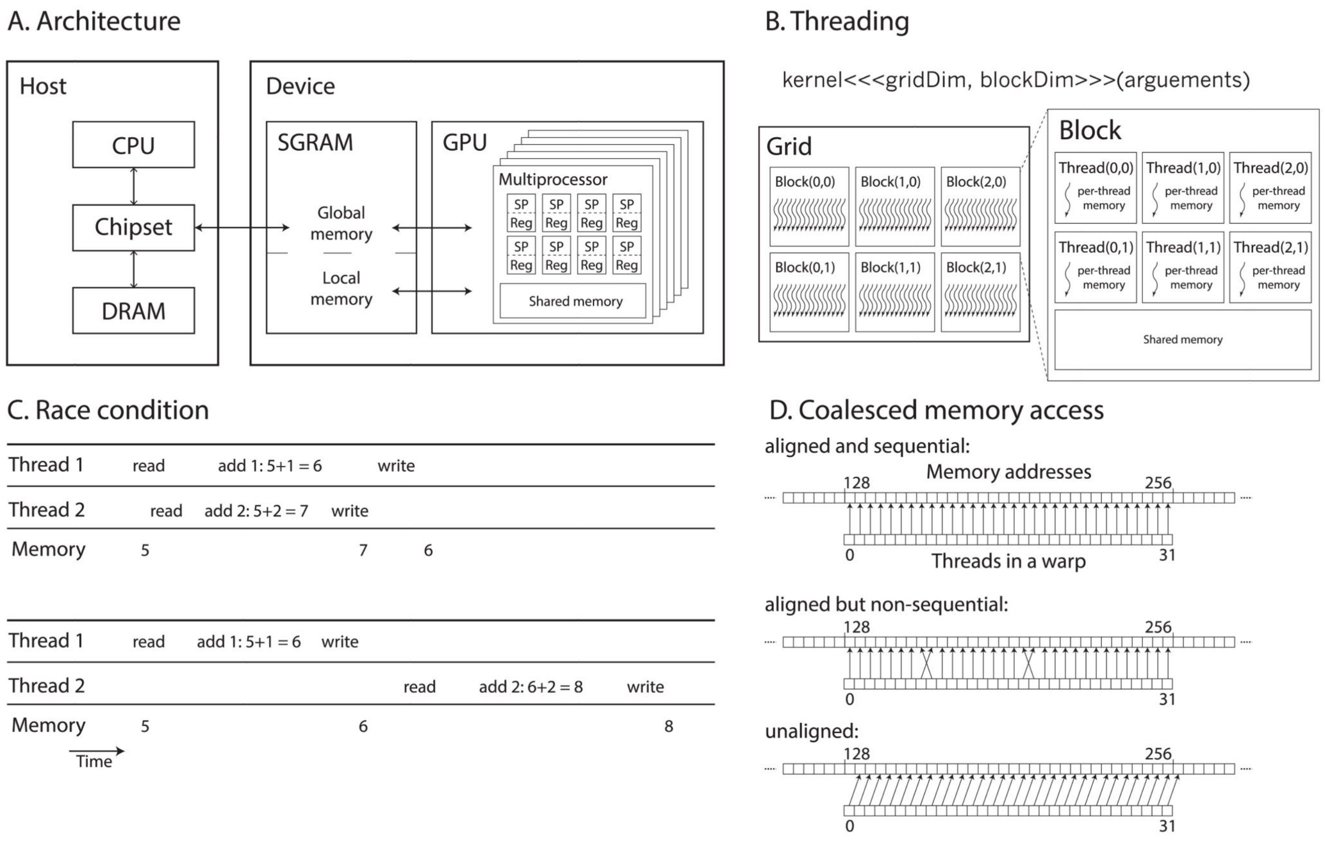

Fig. 1.

(A) Schematic of current GPU hardware architecture, and the different memory classes. (B) Grid structure of a parallel application launch. (C) An example race condition. (D) Three different forms of coalesced memory access.

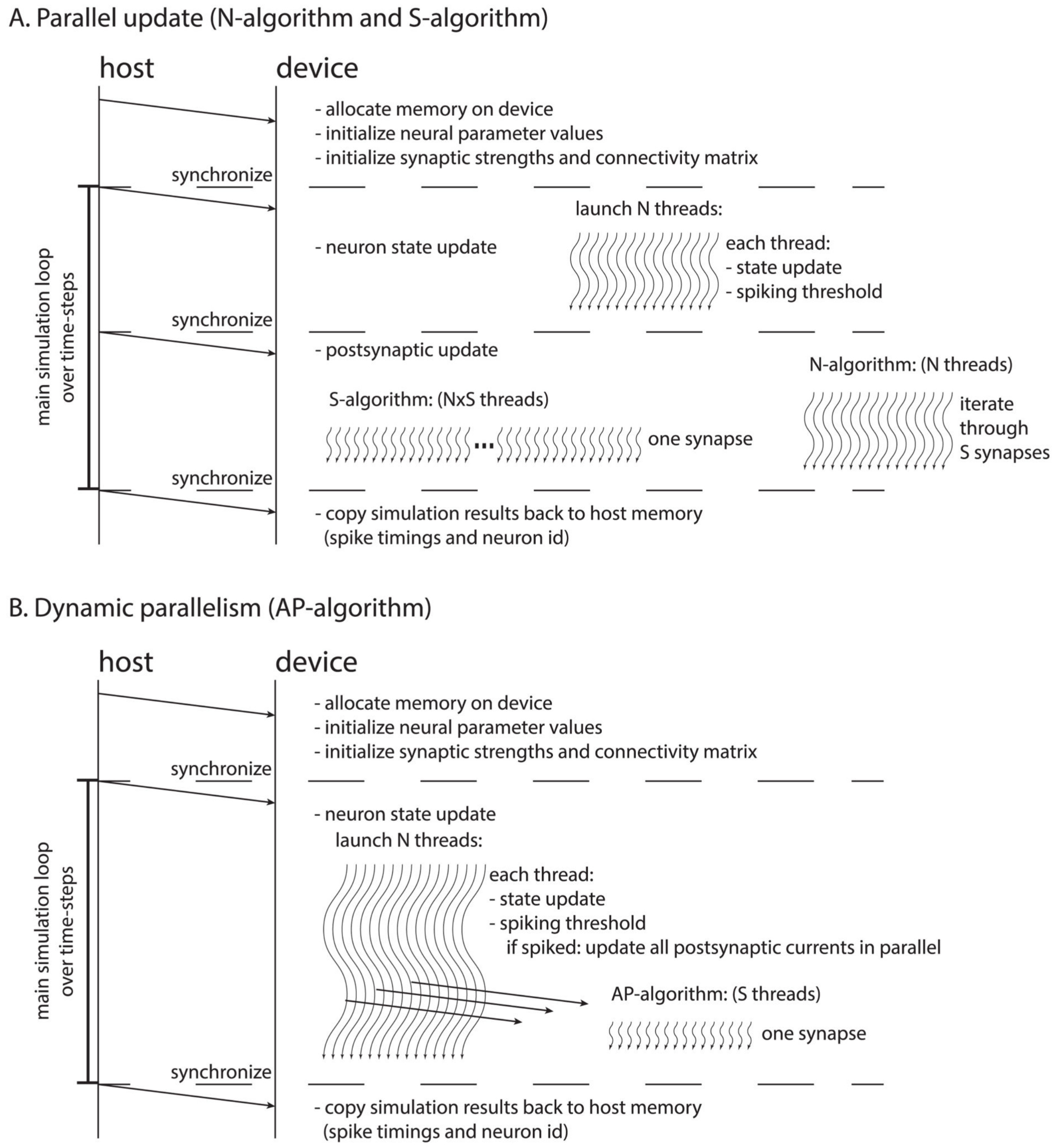
A. Parallel update (N-algorithm and S-algorithm)
host
device
- allocate memory on device
- initialize neural parameter values
- initialize synaptic strengths and connectivity matrix
synchronize
main simulation loop over time-steps
launch N threads:
- neuron state update
each thread:
- state update
- spiking threshold
synchronize
- postsynaptic update
N-algorithm: (N threads)
S-algorithm: (NxS threads)
one synapse
iterate through S synapses
synchronize
- copy simulation results back to host memory (spike timings and neuron id)
B. Dynamic parallelism (AP-algorithm)
host
device
- allocate memory on device
- initialize neural parameter values
- initialize synaptic strengths and connectivity matrix
synchronize
main simulation loop over time-steps
- neuron state update
launch N threads:
each thread:
- state update
- spiking threshold
if spiked: update all postsynaptic currents in parallel
AP-algorithm: (S threads)
one synapse
synchronize
- copy simulation results back to host memory (spike timings and neuron id)

Fig. 2.

Overview of different algorithms to update state variables and current input at a time-step for *N* neurons and *S* synapses per neuron. The main simulation loop determines the duration of the simulation. Boxes summarize what each thread calculates in parallel for different algorithms. For all simulations, states of *N* neurons are updated in parallel. (A) Parallelization across neurons (N-algorithm), and parallelization across synapses (S-algorithm). (B) Spike-triggered parallelization (AP-algorithm).

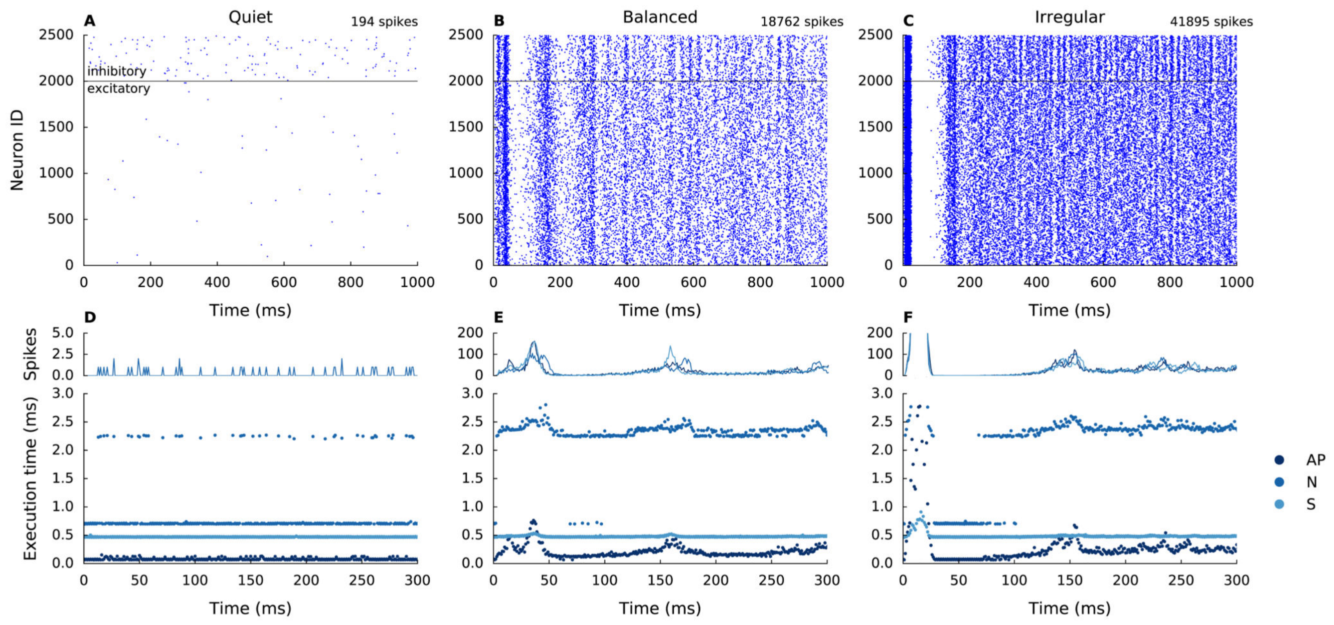

Fig. 3.
(A–C) Example raster plots and (D–F) algorithm performance for a network of N=2500 neurons, each with S=1000 synapses, simulated under three firing regimes during one second: (A) Quiet (194 spikes in the network), (B) balanced (18762 spikes), and (C) irregular firing (41895 spikes). (D–F) The total number of spikes per time-step for the different regimes and algorithms is displayed in the top panels. Results are shown for the first 300 ms of the simulations under (D) quiet, (E) balanced, and (F) irregular firing in the network. Panel D only shows the spike counts for the S-algorithm, for clarity. A comparison of the measured execution times of each time-step for the different algorithms is shown by lines in different shades of blue in the bottom panels.

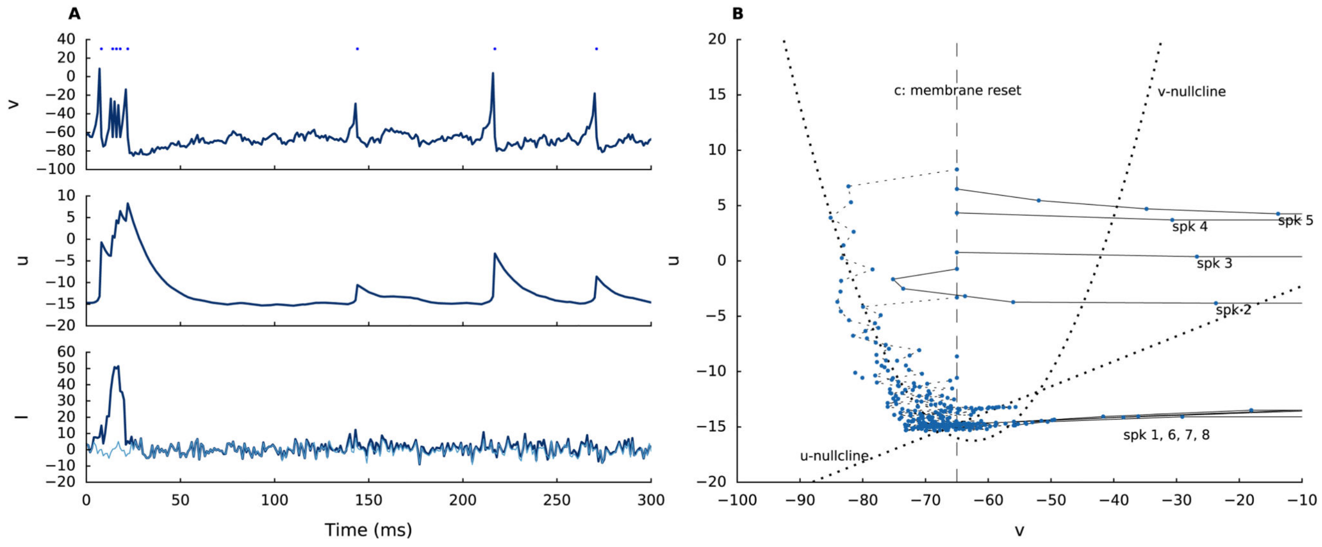

Fig. 4.

Temporal dynamics of a single neuron in the irregular firing regime. (A) The temporal dynamics of the membrane potential (top), $v(t)$, the recovery variable (center), $u(t)$, and the current input (bottom), $I(t)$, shown separately for the stochastic input (light), and the total current input (dark), which includes the synaptic pulses in the network (Eq. (4)). (B) Phase-plane analysis ($u(t)$ vs. $v(t)$) of this neuron during the first 300 ms shows the relative evolution of the state variables of the neuron. Blue points show the ($u(t)$, $v(t)$) values at each time-step. u- and v- nullclines are indicated by the dotted curves (for $I = 0$); the vertical dashed line depicts the membrane reset potential (-65 mV); spikes are shown as thin lines, and are labeled in their order of occurrence. During the initial rise of the input current (between 5–25 ms), u increases rapidly during repetitive firing, and the neuron elicits a short high-frequency burst of 5 spikes. The (u, v) trajectory on the phase plane returns back to the systems stable point (the resting value of both variables, first intersection of the nullclines) after the 5th spike, and the following spikes (6, 7, and 8) occur around this nearly stable u value.

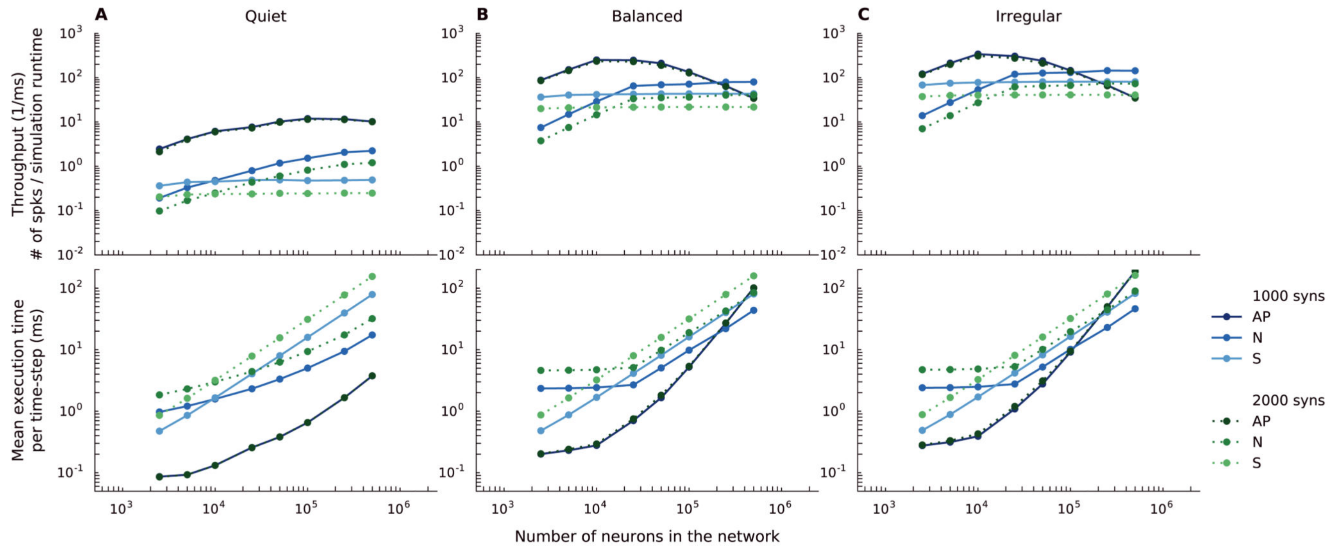

Fig. 5.

Mean execution time per time-step (bottom) and throughput (top) of the SNN simulations, as function of the number of neurons (on logarithmic scales), and either S=1000 (blue), or S=2000 (green) randomly assigned synapses/neuron, for the different parallelization strategies (see legend) under quiet (A), balanced (B) and irregular (C) firing regimes. For networks up to $2.5 \cdot 10^5$ neurons (and larger *S*), the PA-algorithm outperforms the N- and S-algorithms for all conditions, as its throughput is higher, and the mean execution time shorter. Note also that, in contrast to the N- and S-algorithms, AP performance is insensitive to *S*, but it depends more strongly on the firing regime. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

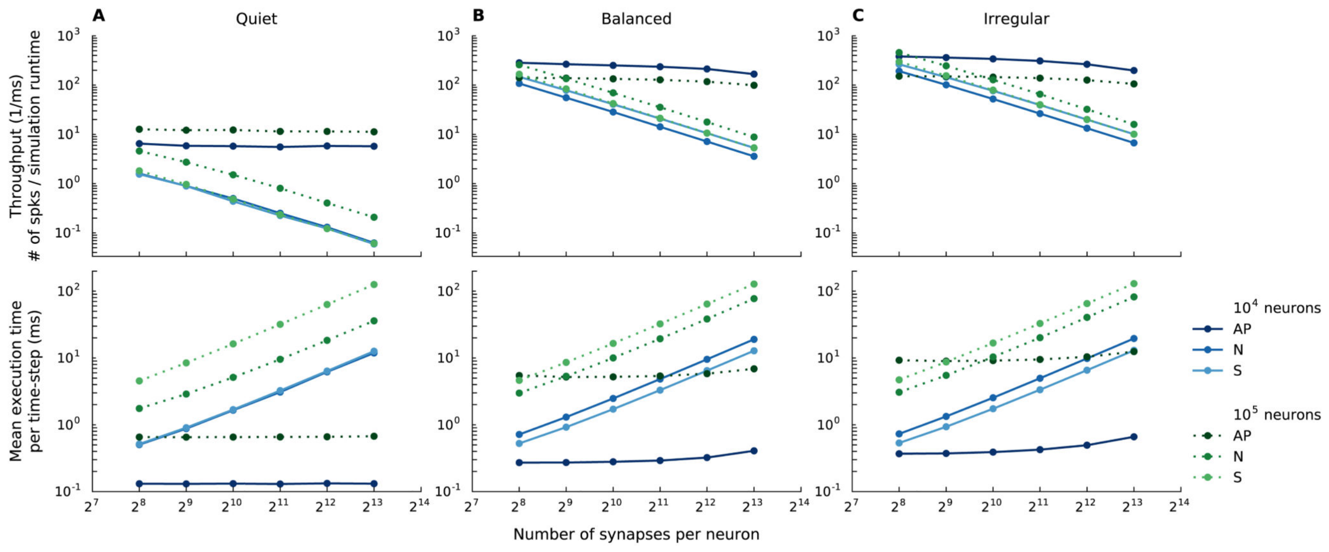

Fig. 6.

Mean runtime per time-step and throughput of the simulations, as function of the number of synapses/neuron (logarithmic scale) in a network with 10^4 (blue) and 10^5 (green) neurons, for the different parallelization strategies (see legend) under quiet (A), balanced (B) and irregular (C) firing regimes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

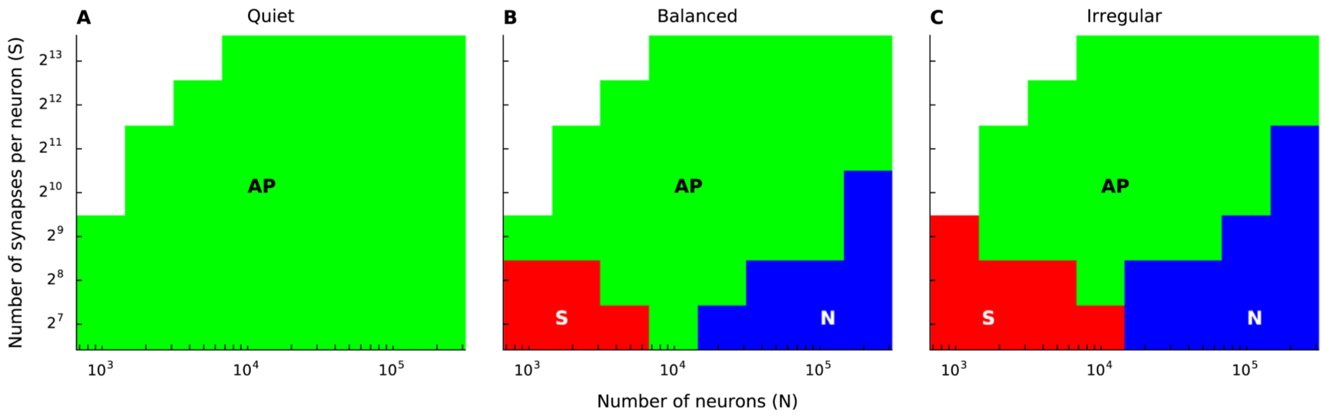

Fig. 7.

Comparative performance of the three algorithms for the three different firing regimes (A: Quiet, B: Balanced, C: Irregular), as function of the number of neurons (N) and the number of random synapses per neuron (S). In each bin, the winning algorithm has been indicated by color (AP: green, S: red, N: blue). Note that the AP algorithm outperforms the other two algorithms, especially when the spike counts are low (A): it is the fastest algorithm, irrespective of network size in the quiet regime, but also for higher spiking activity, when the number of synapses per neuron is high (B and C). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 1

Overview of parameters to set up a heterogeneous network with distinct neurons. r_i and q_i are random numbers drawn from a uniform distribution on [0, 1].

	Excitatory neurons	**Inhibitory neurons**
a_i	0.02	0.02+0.08 r_i
b_i	0.2	0.25-0.05 r_i
c_i	−65+15 r_i^2	−65
d_i	8–6 r_i^2	2
I_i	2.5 q_i (quiet)	−1.0 q_i (quiet)
	5.0 q_i (balanced)	−2.0 q_i (balanced)
	7.5 q_i (irregular)	−3.0 q_i (irregular)