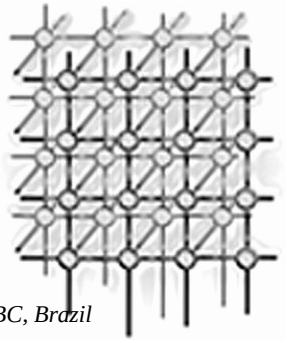
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–6 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

A Multi-GPU Algorithm for Large-scale Neuronal Networks



Raphael Y. de Camargo[*,†,‡], Luiz Rozante[‡], and Siang W. Song[‡,§]

[‡]*Center for Mathematics, Computation and Cognition, Universidade Federal do ABC, Brazil*
[§]*Department of Computer Science, Universidade de São Paulo, Brazil*

SUMMARY

Large-scale simulations of parts of the brain using detailed neuronal models to improve our understanding of brain functions are becoming a reality with the usage of supercomputers and large clusters. However, the high acquisition and maintenance cost of these computers, including the physical space, air conditioning, and electrical power, limits the number the scientists that can perform this kind of simulation. Modern commodity graphical cards, based on the CUDA platform, contain graphical processing units (GPUs) composed by hundreds of processors that can simultaneously execute thousands of threads and thus constitute a low-cost solution for many high-performance computing applications.

In this work, we present a CUDA algorithm that enables the execution, on multiple GPUs, of simulations of large-scale networks composed of biologically realistic Hodgkin-Huxley neurons. The algorithm represents each neuron as a CUDA thread, which solves the set of coupled differential equations that model each neuron. Communication among neurons located in different GPUs is coordinated by the CPU. We obtained speedups of 40 for the simulation of 200k neurons that received random external input and speedups of 9 for a network with 200k neurons and 20M neuronal connections, in a single computer with 2 graphic boards with 2 GPUs each, when compared with a modern quad-core CPU.

KEY WORDS: GPU computing, CUDA, simulation, neural networks, Hodgkin-Huxley model

1. INTRODUCTION

To improve our understanding of brain functions, such as memory [1], vision [2], cortical processing [3, 4], and mental illnesses [5], scientists perform large-scale simulations of parts of the brain using detailed neuronal and connectivity models. In realistic simulations, each neuron is modeled by a set of coupled differential equations (from a couple to thousands per neuron), that describe the

[*]Correspondence to: Raphael Y. de Camargo, Univ. Federal do ABC, R. Santa Adélia, 166. Santo André/SP, Brazil, 09210-170
[†]E-mail: raphael.camargo@ufabc.edu.br
Contract/grant sponsor: Brazilian National Research Council (CNPq); contract/grant number: 550895/2007-8, 474714/2009-8, 301652/2009-0
Contract/grant sponsor: CAPES ; contract/grant number: PVNS (National Senior Visiting Professor Program)

Received
Revised

dynamics of the neuron membrane and ionic channels [6, 7]. Neurons communicate through synaptic connections, described by their source and target neurons, communication propagation delay and synaptic weight. A simulation can contain millions of neurons and billions of synaptic connections, generating a high demand of computing power [8]. Large-scale simulations are currently performed on supercomputers [9, 3], such as the IBM BlueGene, and large clusters [10, 11, 4]. The acquisition and maintenance cost of these computers, including the physical space, air conditioning and electrical power to maintain those computers, is prohibitively high for most institutions.

Modern GPUs, based on the CUDA platform [12, 13], have hundreds of simple processors that, when used in parallel, can sustain high computing power. Due to the low cost of GPU boards and small space requirements, their usage constitutes an excellent alternative in the area of high-performance computing. GPUs are optimized for SIMT (Single-Instruction Multiple-Thread) floating-point operations, where a large number of threads execute a single instruction, such as in the numerical integration of a large number of differential equations. The CUDA platform has already been used for a wide variety of applications, such as simulation of stochastic systems of chemical reactions [14], molecular dynamics [15], electrostatic potentials [16] and fluid flows [17].

In the area of neural networks, Bernhard *et al.* [18] simulated networks of integrate-and-fire neurons, which are very simple neuron models represented by a single differential equation. These implementations are prior to CUDA, which means that the simulation elements were mapped in textures and the operations over the elements in geometrical operations. Nageswarana *et al.* [19] implemented a simulator for large-scale spiking neural networks, with neurons based on the Izhikevich's simplified spiking neuron model [20], which is more realistic than the integrate-and-fire neurons and can generate some realistic behaviors. They designed an efficient algorithm for spike processing and delivering that work for their simplified communication model and on a single GPU. In contrast, the detailed neuronal models, which we use in our work, include information on cell morphology and ionic and synaptic channels, resulting in dozens of state variables and differential equations per neuron. The algorithms to solve the detailed models are more complex and very different from the algorithms for simpler models. To the best of our knowledge, there are no studies about the simulation in GPUs of large-scale neuronal networks that use detailed neuronal models. This paper aims to fill this gap.

We present a CUDA algorithm that enables the execution, on multiple GPUs, of simulations of large-scale networks composed of biologically realistic Hodgkin-Huxley neurons[†]. Each neuron is modeled as a set of coupled differential equations and dozens of state variables. We assign a CUDA thread per neuron and we launch thousands of threads per GPU that perform the numerical integration of the differential equations in parallel on multiple GPUs. We use the CPU to coordinate the communication among neurons executed on different GPUs.

We implemented and performed a detailed experimental evaluation of the algorithm, including the analysis of simulation accuracy, speed-up compared to CPUs, scalability analysis and profiling of the execution time. We show that it is possible to perform simulations of networks with over 200k biologically realistic neurons and 20M synaptic connections using a single computer, with 2 graphic boards with 2 GPUs each, with the same performance of a small conventional cluster.

[†] Simulator source code and experimental setup available at `http://ncsc.ufabc.edu.br/~rcamargo/neuralcuda`.

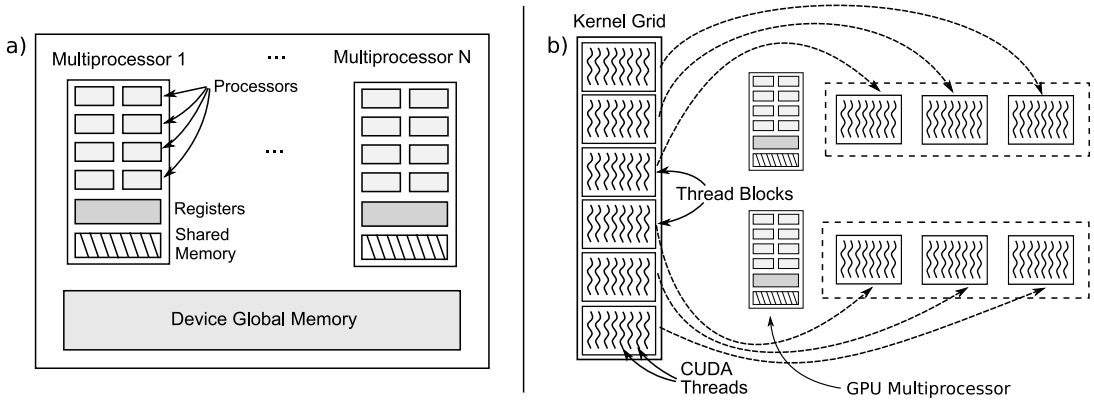


Figure 1. The CUDA platform. a) Architecture of a modern GPU, containing a large global memory and a set of multiprocessors, each one with an array of floating-point processors, a small shared memory and a large number of registers. b) Hierarchical organization of CUDA threads in thread blocks and in kernel grids, where each thread block is assigned to a single multiprocessor.

2. CUDA PLATFORM

Modern graphic boards have powerful GPUs (Graphics Processing Unit) composed of hundreds of simple processors for floating-point operations, enabling the parallel processing of a large number of instructions [12]. Figure 1 shows the GT200 architecture, which is organized as a set of multiprocessors, each composed of 8 processors, a large number of registers, and a small high-speed shared memory.

The CUDA architecture [13] supports an extension of the C programming language, where programmers can define special functions, called *kernels*, which are executed in the GPU, while the remaining of the CUDA programs are executed in the CPU. For each kernel execution, the user must define the number of threads to launch and divide the threads in blocks, forming a grid of blocks. In CUDA, each kernel block is executed in a single multiprocessor, which execute the kernel threads of each received block in parallel, as shown in Figure 1.

To use all the n multiprocessors from a GPU, it is necessary to create at least n blocks. Moreover, each multiprocessor simultaneously executes groups (called *warps*) of w threads from a single block, and several warps should be present on each GPU for efficient usage of its processors. For example, NVIDIA's GTX 295 boards have 1982MB of global memory and 2 GPUs, each one with 240 processors divided among $n = 30$ multiprocessors, and each one with 8192 registers and 16kB of shared memory. If the warp size w is 32, we would need $30 * 32 * 4 = 3840$ threads per GPU for efficient execution, supposing we need 4 warps per block.

The main challenge when implementing efficient CUDA programs is coding the application in a number of threads large enough to keep all the GPU processors occupied. Each thread, however, should also keep most of the state variable that it uses in the small amount of shared memory available per

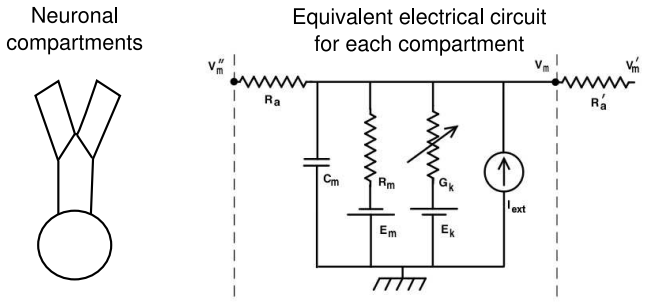


Figure 2. Model of a single neuron as a set of isopotential compartments, with each compartment represented by an electrical circuit.

multiprocessor, since the global memory access latency is very high. With more threads per kernel block, a smaller fraction of the thread state variables will fit in the shared memory, and thus it is necessary to find a tradeoff between them.

3. SIMULATION OF DETAILED NEURONAL MODELS

Neurons are specialized cells that have a polarized membrane that maintains a potential difference of about 60mV between the internal and external mediums. Information processing occurs through changes in this membrane potential. To enable the efficient simulation of the neuron dynamics, we model neurons as a set of isopotential compartments connected by a radial resistance [6, 7]. Each compartment functions as an electrical circuit, with the cell membrane represented by capacitors and ionic channels by resistances, as shown in Figure 2.

The membrane potential $V_m(t)$, at time t, is determined by integrating a set of differential equations, with each equation representing a neuronal compartment m, shown in Figure 2. We must integrate a separate set of differential equations for each neuron, since each one has different values for its state variables, such as V_m. The equation for each compartment has the form:

$$C_m \frac{dV_m(t)}{dt} = \frac{E_m - V_m(t)}{R_m} + \frac{V_m' - V_m(t)}{R_a'} + \frac{V_m'' - V_m(t)}{R_a} + I_{ion}(t) + I_{ext}(t) \qquad (1)$$

where the constant E_m represents the membrane reverse potential, C_m the membrane capacitance, R_m the membrane resistance, and R_a the axial resistance. V_m', V_m'' and R_a' are the corresponding values for the neighbor compartments.

The variable $I_{ext}(t)$ is the external current applied in the neuron and $I_{ion}(t)$ is the current that pass through ionic channels present in the membrane. The current I_{ion} in each compartment is given by:

$$I_{ion}(t) = \sum_i (E_i - V_m(t))G_i(t)$$

where i represents the ionic channel from each compartment, $G_i(t)$ the conductance of the channel at time t and E_i the reverse potential for the ions that pass through channel i.

Active channels. The voltage dependent active channels are responsible for spike generation [6, 7], which occurs when the membrane potential reaches a threshold, and is the mechanism by which neurons communicate. Active channels are modeled as a set of gates, that can permit or block the passage of ions, with independent dynamics of opening and closing, as proposed by Hodgkin-Huxley [21]. For example, we can model sodium (Na) channels as having two gates, m and h, that control the flow of ions through the channel.

We represent the set of Na channels of each compartment as a single channel, with the conductance of the channel at time t given by $G_{Na}(t) = g_{max_{Na}} * m^3 * h$, where $g_{max_{Na}}$ represents the maximum conductance of the channel. The gate variables m and h assume values from 0 to 1.0, representing the percentage of gates that are open at time t. The dynamics of each gate is given by an equation of type:

$$\frac{dm(t)}{dt} = \alpha_m(V)(1 - m(t)) - \beta_m(V)m(t) \tag{2}$$

where $\alpha_m(V)$ and $\beta_m(V)$ are the rate of opening and closing of the gate m, and their values are dependent on the membrane potential V. There are similar equations for gate h, with different functions $\alpha_h(V)$ and $\beta_h(V)$. Functions $\alpha(V)$ and $\beta(V)$ are the main determinants of the active channel activity and different models have distinct functions.

Cells can have others types of channels, such as potassium (K) channels, with a single gate type n and conductance given by $G_K(t) = g_{max_K} * n^4$.

Synaptic channels. They are the main communication mechanism in neuronal networks [7] and are activated by the release of neurotransmitters from a presynaptic neuron j in a synaptic channel i of a postsynaptic neuron, which are triggered by spikes generated at neuron j. The behavior of the neuronal network is determined by the pattern of connections among the neurons and the synaptic weights w of these connections, which determine the strength of interactions. Depending on the connection pattern, networks can act as pattern recognition networks in the visual system [2], control visual attention, or enable the storage of long-term memories [1].

The conductance $G_i(t)$ of each synaptic channel i at time t is given by:

$$G_i(t) = \sum_{spk} g_{i_{max}} w_{ji} \frac{t - t_{spk}}{\tau} exp(1 - \frac{t - t_{spk}}{\tau}) \tag{3}$$

where spk represents each delivered spike, $g_{i_{max}}$ the maximum conductance of the channel, w_{ji} the synaptic weight for spikes from source neuron j, t_{spk} the delivery time of each spike and τ the channel time constant, which defines the speed of the activation and inactivation of the synaptic channel. After a period of $4 * \tau$, the contribution of a spike in the postsynaptic cell can be considered negligible.

4. THE SIMULATION ALGORITHM

The simulation has two main parts, which are: (1) integration of the set of differential equations representing the k compartments of each neuron; and (2) spike processing, where the algorithm verifies

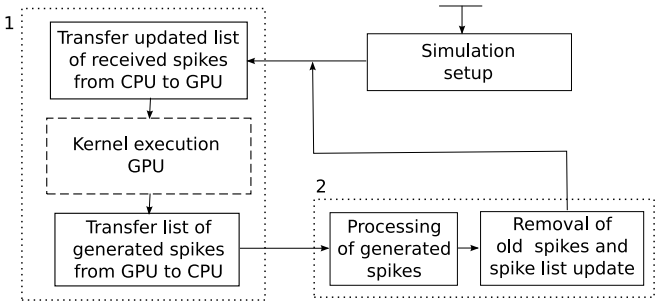


Figure 3. The simulation algorithm. It is composed of the simulation setup and a main loop with two parts: (1) integration of the set of differential equations of each neuron; and (2) spike processing.

the spikes generated at each neuron and delivers it to the neurons to which it connects. The first part is the most computationally demanding step of the simulation and which we implemented as a CUDA *kernel* for execution in the GPU. The second part involves the messages exchanged among neurons located in different GPUs and, consequently, we used the CPU to perform the spike processing.

The simulation algorithm is divided in steps, shown in Figure 3. It starts with the simulation setup, which configures the neurons, allocates memory in the device (graphic board) and transfers the neuron simulation data from the host (computer) main memory to the device memory. The bulk of the algorithm consists of the parts 1 and 2, which are executed repeatedly until the simulation finishes. Our algorithm permits the usage of multiple GPUs, in which case a different CPU thread is launched for every GPU used.

In part 1, the simulator transfers information about delivered spikes to the GPU, launches the CUDA kernel, which solves the differential equations, and, finally, transfers the list of generated spikes from the device to host memory. In part 2, the algorithm checks the spikes generated by each neuron and sends the spikes to each neuron that it connects, and then synchronizes the CPU threads to guarantee that they all finished the spike delivery. Finally, it processes the list of spikes received at each neuron, removing old received spikes and organizing the spikes for transferring to the device memory.

We decided to perform a sequence of n integration steps during each CUDA kernel execution. This does not cause any effect in the simulation results, since in biological neuronal networks there are communication delays between spike generation and post-synaptic activation. For a communication delay of $10ms$ and $\delta t = 0.1ms$, we can safely choose $n = 100$. The execution of n integration steps per kernel call has two performance advantages: (1) the overhead of each kernel call is too high, due to the process of switching the execution to the GPU and the repopulation of the shared memory of all multiprocessors; and (2) the CPU threads must synchronize during spike processing, to guarantee that the spikes will be delivered at the correct time. Processing n integration steps per kernel call reduces the number of kernel calls and synchronizations by a factor of n.

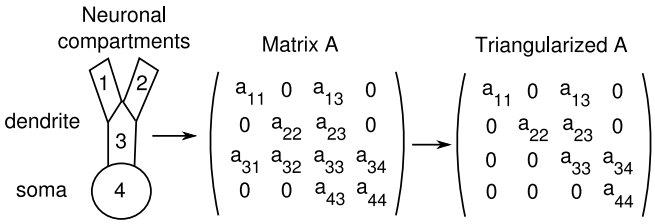


Figure 4. Triangularizable matrix A generated using the Hines method. Compartment c is assigned to line c, which contains non-zero elements only in the columns c' that represent the links with other compartments c'.

4.1. CUDA Kernel

To solve the system of differential equations of each neuron, we used the method described by Hines [22]. In this method, the equations are coded as a linear system of the type $A * V = B$, where A is a $k \times k$ sparse matrix, with its rows containing the voltage-dependent coefficients from each compartment, V is a vector of size k containing the membrane potential $V_m(t)$ on each compartment, and B is a vector of size k containing the potential independent values. In each integration step the linear system is solved and the simulation advances a time interval δt. Since matrix A is sparse, we can represent the matrix as an array of size $O(k)$.

If matrix A is triangularizable, we can solve the system by performing the triangularization followed by back-substitution, where we evaluate first the value of V_k, which is the potential of the last compartment, followed by V_{k-1}, and so on. A triangularizable matrix is produced by numbering the compartments starting at the most distant one and finishing at the soma [22]. Figure 4 shows a neuronal model with 4 compartments and the corresponding A matrix before and after the triangularization.

4.1.1. Kernel algorithm

We map each neuron as a single CUDA thread, with each thread performing all the steps of the Hines method for its corresponding neuron, since the triangularization and back-substitution must be performed sequentially for each neuron. Figure 5 shows the simulation steps for each kernel thread. Step 1 transfers the heavily used data from global memory to the shared memory, reducing the memory access time. In steps 2 to 6, the kernel performs the numerical integration of the neuron equations, repeating these steps n times. The kernel finishes in step 7, where the data modified during the kernel execution, such as the membrane potential and active channel gate states are written back to global memory. We describe steps 2 to 6 in detail below.

Active channels. To determine the current passing through each active channel in step 2, it is necessary to evaluate the state of each gate from the channel, which is done by integrating Equation 2. Since the functions α and β are dependent on V, they must be evaluated in every integration step. From

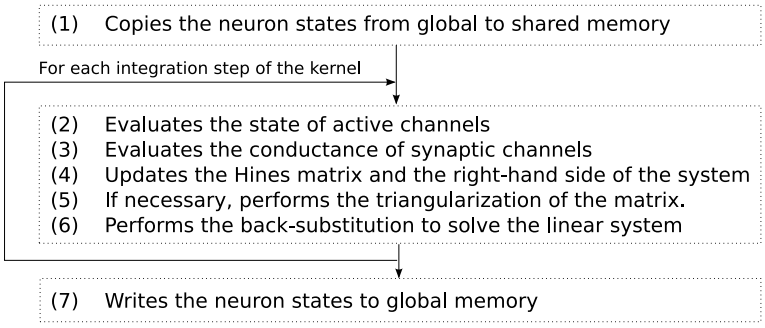


Figure 5. Simulation steps of the CUDA kernel. This algorithm is executed in parallel by each kernel thread.

the percentage of gates open for each type of channel it is possible to determine the conductance of each active channel and the current passing through it.

Active channels are responsible for spike generation, which are triggered when the membrane potential exceeds a sharp threshold. During spike generation, the channels conductances change very rapidly and the values of the currents passing through each active channel are used to determine the potential on each cell compartment. The presence of the active channels generates a system of stiff differential equations, which is the limiting factor for increasing the step size. To enable the usage of integration steps of moderate size, we determine the values of the channel gates (Equation 2) and the currents of the active channels at the midpoint of each time step, that is, $t + \delta t/2$, which increases the precision of the integration.

Synaptic channels. In step 3 we evaluate the current in the synaptic channels, which are activated by spikes generated in the presynaptic neurons. The simulator evaluates Equation 3 for every spike delivered to each synapse in the neuron. For each neuron, we keep in the main memory an array containing the time of each generated spike and the corresponding synaptic weight. But since each neuron can receive spikes from thousands of neurons, it is not possible to transfer the complete array to the shared memory, so the spike times are obtained from the global memory at every integration step.

The global memory access latency compromises the kernel performance, but we can reduce this problem by running a higher number of threads per block, for example, 128 threads. In this case, while some threads are waiting for the spike information from the global memory, others are evaluating their synaptic channel conductance from spikes obtained previously. Moreover, in each global memory access, we can fetch information about multiple spikes.

Solving the linear system. The kernel solves the linear system of equations representing the cell compartments in steps 4 to 6. We perform implicit integration in the system of differential equations that represent the cell compartments (Equation 1), since it allows the usage of larger integration steps.

Step 4 is straightforward and consists of updating the right-hand side (vector B) of the system and the matrix A. Next, we triangularize the updated matrix A (if needed). When there are active channels only in the cell soma, only the coefficient of the last compartment is modified and, consequently, there is no need to triangularize matrix A in every step. When the channels are located in other compartments, the triangularization in every step is required, since the coefficients of other compartments are also changed. Our simulator takes advantage of this and triangularizes the matrix A only when necessary. After the triangularization, the kernel finishes the integration step by performing a back-substitution, where we first evaluate the value of the last compartment (soma), which we will call compartment k, then the value of compartment $k-1$ and so on, until we evaluate all the potentials at time $t+\delta t$.

4.1.2. Kernel algorithm implementation

The performance of CUDA applications is determined by the fraction of GPU processors active at each moment. This requires the usage of a large number of threads and that threads have immediate access to the data they need, which is accomplished by putting the state variables that each thread needs in the shared memory. Determining the number of threads (neurons) per kernel block is an important parameter. On the one hand, more neurons mean the possibility of higher parallelism, since more threads can be executed by each multiprocessor. On the other hand, the shared memory will hold only a smaller part of the neuron state, requiring more accesses to the high-latency global memory.

We need to keep separate storage space for the state variables of each neuron, such as the V_m on each compartment and ionic and synaptic channels states. Since the shared memory can hold only part of these variables, we selected the ones used multiple times on each integration step and whose state must be kept across the integration steps, such as the V_m and the channel gates state. This enables a higher number of neurons per block, compensating the latency caused by accesses to the global memory, since there will be more threads ready for execution. We used between 32 and 196 threads per block, depending on the number of compartments per neuron and the precision of floating point numbers.

To perform load-balancing, we define a selection of neuron types and allocate each simulated neuron to one of these types. Load-balancing is obtained by distributing evenly the blocks among the GPUs. This grouping also promotes a reduction in shared memory usage, since we can share static information that is equal for all neurons of its type, such as morphological and membrane property information, which are consolidated in the Hines matrix.

There are several optimizations that can be applied to CUDA applications, such as coalescing global memory accesses and preventing access conflicts in the banks of the multiprocessors shared memory. Such optimizations often bring important performance enhancements [13]. We applied these techniques in our code, specially in data that are accessed often, such as the membrane potential on each compartment, the current in the active channels and the state of the active channel gates. Although they brought some important gains in performance, these fine grain optimizations are not the focus of this paper and will not be discussed here.

4.2. Neuronal communications

After finishing the kernel execution, the next step of the simulation algorithm, shown in Figure 3, is to process the generated spikes. For each neuron, the simulator gets the list of generated spikes and delivers the generated spikes to all neurons to which it connects. The connectivity of each individual

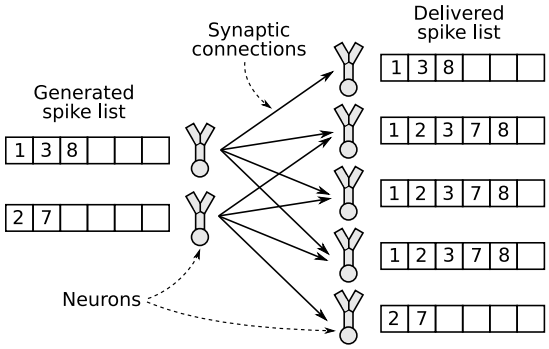


Figure 6. Spike processing and delivery, where the spikes generated by the pre-synaptic neurons during the last kernel execution are delivered to all post-synaptic neurons to which they connect.

neuron is defined independently, including the number of synapses, their weights, and axonal delay. This allows the simulation algorithm to have the flexibility required for the majority of large-scale simulations developed recently [3, 1, 4, 2].

The algorithm searches for spikes in all neurons of the simulation and delivers the spikes to all its post-synaptic neurons, as shown in Figure 6. This generates a huge number of delivered spikes, making the spike processing step of the simulation time and memory consuming. For instance, if there are 100k neurons, each connected to other 1000 neurons, and the mean number of generates spikes per neuron per kernel call is 2, there will be 200 million delivered spikes after each kernel call. Each synaptic channel from each neuron has a delivered spike list that contains all the spikes that contains the spike time and synaptic weight for each spike delivered to the synaptic channel. The delivered spike list is copied to the GPU before starting the CUDA kernel execution for the next n steps, and is transversed on every integration step to determine the current on each synaptic channel. The spike list resides in the high-latency device global memory and, consequently, reducing the spike list size improves both performance and memory usage.

We used two strategies to reduce the number of entries in the delivered spike list. The first was to implement this list as a hash map, which maps the spike time with the connection weight. If two or more spikes generated at the same time are delivered to the synapse, they are merged in a single entry by summing the weights of each connection, thus reducing the number of spikes to process in each integration step. The second strategy was to remove the spikes generated earlier than 4τ time units from the current time, where τ is the time constant of the synaptic channel, since these spikes would have negligible impact on the neuron synaptic currents. After delivering the spikes, we traverse the hash map of each synapse and eliminate the old spikes. However, even with these optimizations, spike delivery causes most of the memory consumption at the host machine and is the limiting factor for increasing the simulation size.

Another option would be to perform spike processing and delivery in the GPU. Performing spike processing completely in the GPU is not possible, since this would involve communication between

threads of different blocks running on different GPUs. However, we could perform most of this task in the GPU, leaving to the CPU only the task of dispatching the messages to the correct GPUs before the next kernel launch. This possibility is indicated by Nageswarana *et al.* [19], which developed an algorithm for spike processing and delivery in CUDA. The algorithm works only for single GPUs and when using much simpler neuronal and synaptic models, where the spike time and synaptic weight information are used only once per generated spike. But it is not clear that an efficient algorithm can be developed for simulations using multiple GPUs, which would require the synchronization among threads of different GPUs and transfer of large amounts of information among the GPUs, and detailed neuronal and synaptic models, where spike information is used for several integration steps. Since the simulation of neurons is the most computationally demanding part of the complete simulation, we decided to focus this work on the efficient simulation of detailed neuronal models, performing the spike processing and delivery in the CPU.

4.3. Time complexity

To determine the time complexity of the kernel, we evaluate the time necessary to solve the linear system of differential equations and the currents in the active and synaptic channels. The Hines matrix is sparse and can be implemented as a linear array of size $O(nComp)$, where $nComp$ is the number of compartments. Consequently, the time spent solving the linear system is $O(nComp)$ per neuron at each integration step. The time to evaluate the current of the active ionic channels is dependent only on the number of active channels $nActive$. Finally, to determine the time spent evaluating the synaptic currents, we need to determine the number of active spikes per synaptic channel, where each spike remains active during at least an entire kernel execution. Defining $nConn$ as the average number of connections per neuron, $spkRate$ as the mean spike rate of the neurons, and $kSteps$ as the number of steps per kernel call, results in $O(nConn * spkRate * kSteps)$ spikes actives at each synapse.

The complete simulation has $nSteps$ integration steps and $nNeurons$ neurons. The total processing time of the kernel is $O(nSteps * nNeurons * (nComp + nActive + nConn * spkRate * kSteps))$, which indicates that the synaptic processing is the dominant factor in kernel processing.

Spike processing and delivery is executed in the CPU after each kernel call, and it needs to deliver each generated spike to every post-synaptic neuron. It is called $O(\frac{nSteps}{kSteps})$ times and the number of active spikes per neuron is $O(nConn * spkRate * kSteps)$. Consequently, the total processing time of the spike processing and delivery in the CPU is $O(nSteps * nNeurons * nConn * spkRate)$, which is lower than the kernel complexity by a factor of $kSteps$.

5. EXPERIMENTS

We evaluate the simulator to determine the performance gains obtained with the usage of GPUs in comparison with CPUs and to check the precision of the obtained results. We performed the experiments using a computer with a 2.66GHz Intel Core i7 920 processor, 6 GB of RAM memory and 2 NVIDIA GTX 295 graphic boards, with 2 GPUs and 1892 MB of memory on each board. We used a 64 bits Ubuntu 9.04 operating system, CUDA version 2.3 and graphic drivers version 190.18. We used the g++ compiler, configured to generate optimized code with the option -O3.

Besides the GPU implementation, we also implemented the simulator using the CPU, with the objective of comparing the performance gains obtained with the usage of GPUs and the differences in the simulation results due to precision differences. For both versions, we can configure, at compile time, the simulation to run with double precision or single precision floating point numbers. We used C++ to implement the simulation in the CPU. It shares most of the code with the GPU version, except for the kernel, which in the CPU version was coded in a class called `HinesMatrix` that solves the linear system. This class performs the same simulation steps of the GPU kernel, but we assign to each CPU core a fraction of all the neurons, and each core performs the steps for each neuron of its fraction. One important difference is that the CPU does not have a shared memory controlled by the application, but a large cache memory that is controlled by the CPU, resulting in a much simpler code. Neurons are statically divided equally among the cores during the simulation startup, resulting in a balanced distribution of load among the cores.

In all simulations we used a network with two types of neurons (pyramidal and inhibitory). Each neuron contains the 2 types of active ionic channels (Na and K) described in Section 3. Pyramidal neurons have 2 types of synaptic channels, excitatory (AMPA) and inhibitory (GABA) channels, while inhibitory neurons have only excitatory synaptic channels. Each pyramidal neuron is connected randomly to N other pyramidal cells and N other inhibitory cells through excitatory AMPA synapses, and each inhibitory cell is connected to a single pyramidal cell through a inhibitory synapse. These neuron types and network architecture resemble the existing models of the cerebral cortex [3, 5, 2]. We use the same number of pyramidal and inhibitory cells and the pyramidal cells receive random external synaptic input.

5.1. Simulation precision

Current GPU architectures have higher performance when using single precision floating point representations, since it has more processors dedicated to single precision (*float*) numbers than to double precision (*double*) ones. Moreover, the amount of shared memory in each multiprocessor is very limited (16 kB), and double precision numbers use twice the memory [13]. The next generations of GPU cards will improve double precision performance considerably [23], but it is not clear if the difference in performance will be eliminated in the near future.

We compared the differences in performance and precision when executing the simulation using the *double* and *float* data types. We used a network with 100k neurons with 4 compartment and two connectivity patterns: the first with no connections and the second with 100 random connections per neuron, for a total of about 10M connections. We used a simulation time[†] of 10s, during which the neurons received random synaptic input and generated an average of 21 spikes per second per neuron for the network with no connections and 44 for the connected one. We executed 5 series of simulations, varying for each simulation series the connections between the neurons and the input spikes.

Execution time penalties. We evaluated the execution overhead of using double precision numbers in the GPU. When we consider only the execution time spent in the kernel processing, we have a

[†]We use the term *execution time* to denote the time spent to execute the simulation and *simulation time* to denote the elapsed time in the simulated neuronal network.

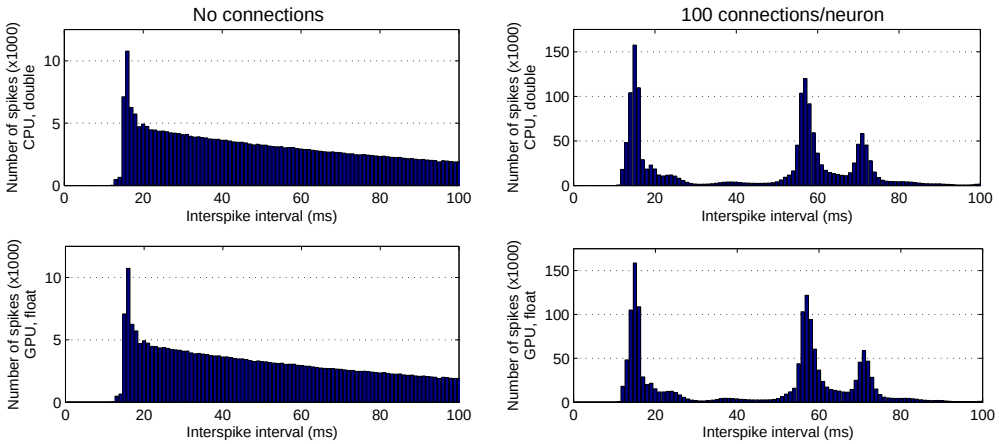


Figure 7. Comparison of the distribution of interspike intervals when using different precisions in the GPU and CPU versions. We considered the scenarios with no connections and 100 connections per neuron.

performance penalty of 43.8% for the simulation of isolated neurons and 49.8% for the simulations with 100 connections per neuron, when using double precision numbers instead of single precision ones. When we consider the total execution time, these penalties fall to 29.9% and 16.5% respectively. As expected, these results show that the performance penalties in the GPU are considerable and we should use single precision numbers if they produce correct results in the simulation. As a comparison, for the simulations using the CPU, the usage of double-precision numbers incurred an overhead that was always below 3%.

Precision errors. To evaluate possible differences in simulation results, we used the distribution of inter-spike intervals, which summarizes the neuronal and network dynamics. Figure 7 shows the mean distribution of inter-spike intervals when considering the spikes of all neurons in the network. The topmost graphics shows the distribution of inter-spike intervals for the simulation in the CPU with double precision numbers and the bottommost graphics shows the simulation in the GPU with single precision. The left-hand side graphics show the simulation with no connections and the right-hand side ones the simulations with 100 connections per neurons.

With no connections, there is a larger number of inter-spike intervals starting at about 15ms and this value slowly decreases as we increase the interval size. These values correspond to the time between the input stimulus in the neurons, which is distributed in a similar fashion. In the connected network, there are three peaks at the intervals 15ms, 57ms, and 72ms, which appear due to the network architecture, where pyramidal neurons are connected to inhibitory ones which inhibits other pyramidal neurons. This metrics is very useful since it summarizes the dynamics of the network. Comparing the results of the simulation using the CPU and double precision with that of the GPU with single precision, we can

see that the differences are very small, which shows that the networks have no significant differences in their dynamics.

There are scenarios where the usage of double precision numbers would be important. For instance, one can consider a network containing a few detailed neuronal models composed of hundreds of compartments and dozens of ionic channel types. In these simulations, the exact response of each neuron can be as important as the dynamics of the neuronal population. But this kind of model is not suitable for simulation in GPUs, since it cannot be easily partitioned among kernel threads.

Since single precision numbers provide sufficient precision in the simulations that are suitable for execution in GPUs and performance is the main issue, we use *float* as the default floating point type in our simulator.

5.2. Distribution of the execution times

We evaluated the distribution of execution time in the different parts of the simulation. The objective was to determine which tasks contribute most to the total execution time, thereby allowing a better understanding of the gains in the execution time obtained by the usage of GPUs in the simulation. In this experiment and in the next ones we use single precision numbers for both CPU and GPU simulations.

We used simulations with 20k and 200k neurons and connectivities of 0, 100 and 1000 random connections per neuron. We also varied the synaptic weights, enabling the pyramidal neurons to operate with 2 mean firing rates: 16 spikes/second (low) and 60 spikes/second (high). These rates correspond to those found in the cerebral cortex [3]. In the 200k neuron simulations, we did not consider the network with 1000 connections per neuron since 6GB of RAM memory in the computer was not enough for spike processing.

We compared the execution times for the same simulation using the GPU and CPU, with the difference that the CPU does not need to perform some of the steps the GPU simulation performs, such as transferring data to and from the device memory. For this experiment, we divided the simulation in 3 sections:

- *HinesKernel:* the time spent in the simulation of each neuron, including the integration of the differential equations of the compartments and active channels, and evaluation of the synaptic channel currents. In the GPU version, this represents the time spent inside the GPU kernel;
- *ConnRead:* the time used to process the generated spikes and deliver them to the target neurons;
- *ConnWrite:* the time spent to process the received spikes on each neuron, including the removal of old spikes and, in the GPU version, to prepare this information for transferring to the device memory.

The topmost graphs in Figure 8, plotted in logarithmic scale, show the execution time in each of the steps for the network with 20k neurons (left-hand side graph) and 200k neurons (right-hand side graph). We can see that the total execution time increases rapidly as we increase the number of connections and the spike rate. The increase occurs not only in the spike processing steps, but also in the the simulation of each neuron (HinesKernel), since in each integration step it is necessary to evaluate the synaptic activity on each neuron. Consequently, the main factor that determines the execution time is the number of spikes delivered to neurons.

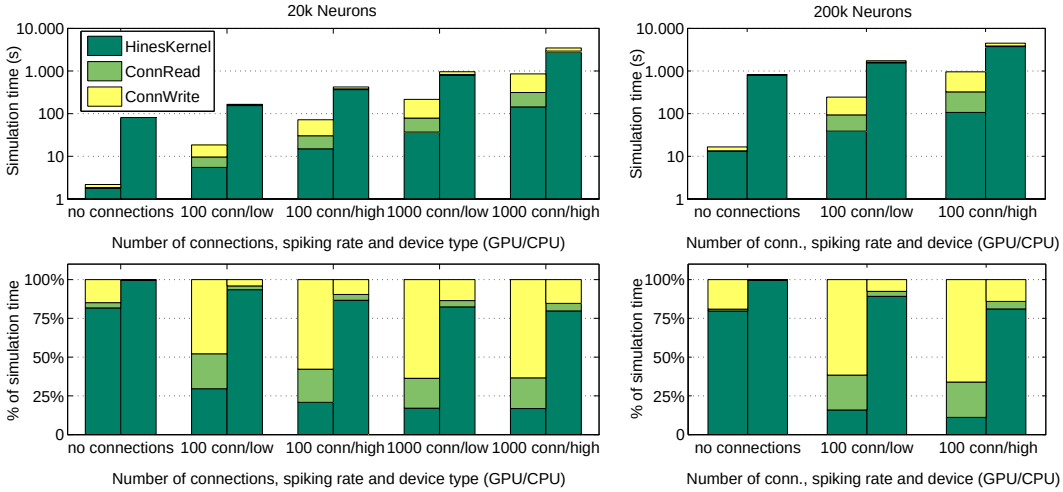


Figure 8. Distribution of execution times in each part of the simulation. The topmost graphs, plotted in logarithmic scale, show the execution times in seconds. The bottommost graphs show the percentage of the total time.

The bottommost graphs show the percentage of the execution time spent in each of the steps. With no connections, nearly all the execution time is spent in the HinesKernel section for the CPU and about 80% for the GPU version. For networks with 100 and 1000 random connections per neuron we can see that most of the GPU execution time is spent in spike processing and for the CPU most of the time is spent in the kernel processing. Actually, the time in the communication steps (ConnRead and ConnWrite) is the same in the CPU and GPU versions, but since the kernel is executed faster in the GPU, the relative contribution of the former in the total execution time is higher in this case. However, in both cases, as we increase the number of generated and delivered spikes, the relative contribution of the spike processing part also increases.

This experiment confirms that solving the differential equations of the neurons (HinesKernel) is the most demanding step, requiring at least 80% of the total execution time in the CPU simulations. Our CUDA implementation of the the HinesKernel step resulted in impressive speedups, between 20 and 60. As result, the contribution of this step in the GPU simulation becomes as low as 11.1% of the total execution time, in the simulation with 200k neurons and high firing rates. Consequently, spike processing is responsible for up to 88.9% of the execution time in the GPU simulation and to obtain higher speedups our next step will be to develop multi-GPU algorithms to accelerate spike processing.

We also evaluated the performance improvements obtained by performing multiple integration steps in each kernel call. Figure 9 shows the time spent in the kernel and communication parts of the simulation for different numbers of steps. The left-hand side graph shows the results for the simulation with 10k neurons and no connections between neurons. The kernel execution time increased only 3.6% when the number of steps was reduced from 100 to 50, but it increased 26.5% for 10 steps, and 267.7%

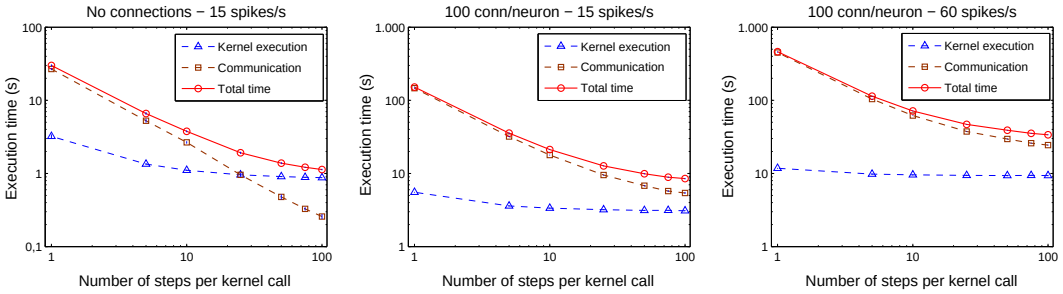


Figure 9. Effect of changing the number of integration steps on each kernel launch. The graphs, plotted in log-log scale, show the execution time of the kernel part, communication part, and complete simulation.

for a single step. The right-hand side graph shows the results for the simulation with 100 connections per neuron and a mean spike rate of 60 spikes per second, where the kernel overhead was negligible for 50 steps, 2.3% for 10 steps and 25.7% for a single step. The increase in the kernel execution time was expected, since there is an overhead for launching the kernel and transferring the state of the neurons from the device global memory to the shared memory. In the scenario with the synaptic connections the increase was smaller because more time is spent processing each integration step.

But in all scenarios of Figure 9 the time of the communication phase increased much faster than the kernel processing, since they must be performed before each kernel call. In simulation with 100 connections per neuron and 60 spikes/s, the increase in the total execution time was 15.4% for 50 steps and 111.9% for 10 steps. For the complete simulation, the overhead of performing 50 steps per kernel launch can be acceptable, but for 10 steps the overhead is too high.

5.3. Performance gain with the usage of GPUs

We evaluated the performance gains obtained with the usage of GPUs and the speedup obtained as we increase the number of GPUs. The simulation configuration is similar to the ones in Section 5.1, except for the number of neurons (1k, 10k, 100k, and 200k) and neuronal compartments (4, 8, 12, and 16). We compared the case with no synaptic connections and with 100 random connections per neuron, with pyramidal spikes rates of 9 spikes/s and 10 spikes/s, respectively. We simulated up to 200k neurons, which due to the amount of memory available in the graphic boards, is the maximum number of neurons that can be simulated in a single GPU. In the CPU simulation, we used all the 4 cores of the processor, by launching 4 threads, each one responsible for one quarter of the neurons.

We measured the speedup of the GPU simulation, in comparison with the one using the CPU, when we varied the number of compartments per neuron. Figure 10 shows that as we increase the number of compartments in a simulation with 100k neurons and no connections, the obtained speedup decreases. This occurs because the amount of state variables per neuron increases linearly with the number of compartments per neuron, requiring a reduction in the number of neurons per kernel block. Many

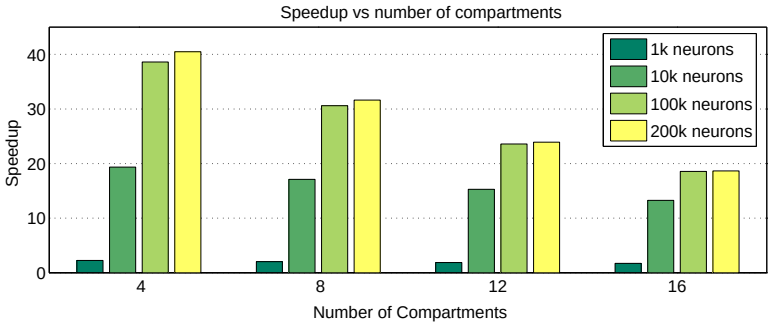


Figure 10. Speedup obtained with the usage of 4 GPUs for simulations using neurons with different complexities, compared with the usage of 4 CPU cores.

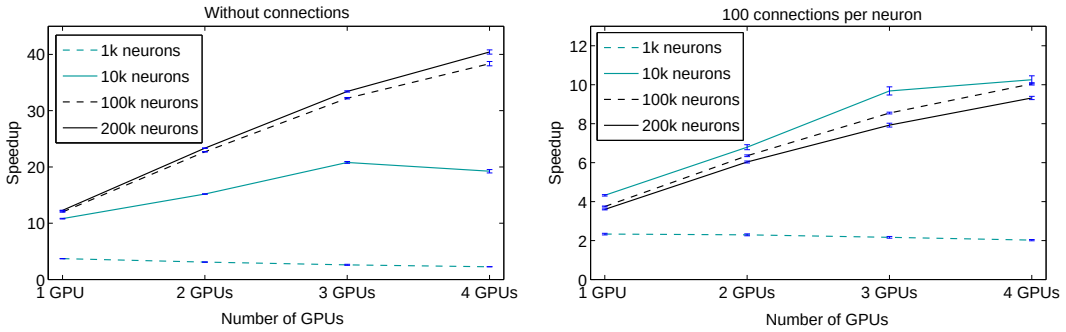


Figure 11. Speedup obtained using different number of GPUs compared with the usage of 4 CPU cores.

realistic simulations use at most 8 compartments per neuron, making this reduction in performance less important. We do not show the results for the simulation with 100 connections per neuron because the number of compartments per neuron influences its dynamics and, consequently, the number of generated spikes per neuron, which would cause distortions in the execution time of each simulation.

We also measured the speedup as we increase the number of GPUs for simulations with different number of neurons and connections per neuron. The left-hand side graphic in Figure 11 shows the mean speedup and the standard deviation, measured over 8 executions of the simulation with no connections. The speedup is higher when simulating larger number of neurons, specially when using multiple cores, since there will be more threads to maintain the GPU processors occupied. For instance, when using 10k neurons, it is better to use only 3 GPUs and when using 1k neurons, it is better to use only one. But when simulation has at least 100k neurons, the speedup with the number of GPUs increased

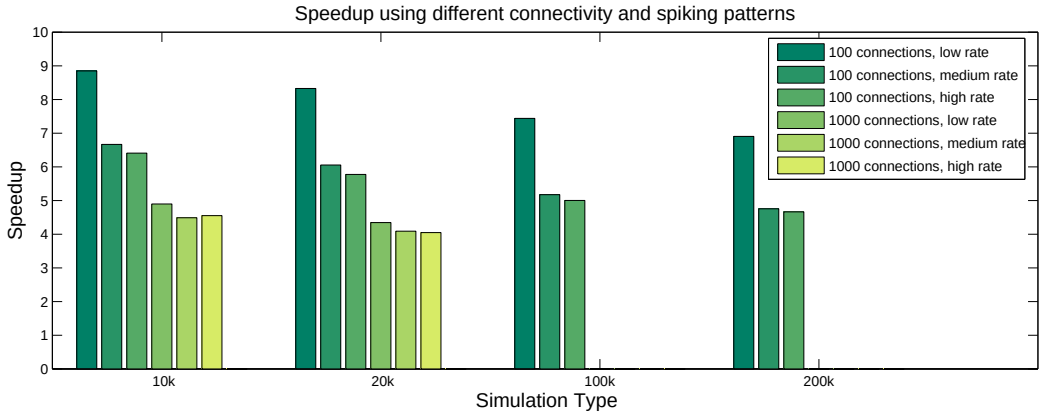


Figure 12. Speedup obtained when executing simulations using 4 GPUs, for networks containing 10k, 20k, 100k and 200k neurons, using 100 and 1000 connections per neuron and mean firing rates of about 16 spikes/s (low), 45 spikes/s (medium), and 60 spikes/s (high).

almost linearly and we obtained speedups of 40 for 4 GPUs. Although neuronal networks always have connections between neurons, this result shows that GPUs are effective for solving the linear systems of differential equations and the ionic channel equations necessary to simulate individual neurons.

In the case with 100 connections per neuron, the gains are still considerable. When using 4 GPUs, we obtained a speedup of 10 for the simulation of 100k neurons and 10M synaptic connections, and 9 for the simulation with 200k neurons and 20M connections. The speedups are lower because of the time spent in spike processing and delivery, which is performed in the CPU. Another difference is that the speedup is lower when we increase the number of neurons and, consequently, the number of synaptic connections. Nevertheless, for the evaluated scenarios, a speedup of 9 is excellent, allowing a scientist to substitute a cluster with 9 machines by a single one.

5.4. Scalability with the number of neurons and connections

We evaluated the changes in the execution time as we change neuronal network properties, such as the number of neurons, connections per neuron, and spike rate. We performed the simulations using 4 GPUs, comparing the result with the simulation in the CPU when using its 4 cores. The graph in Figure 12 shows the speedup obtained as we varied the number of neurons (from 10k to 200k), the number of connections per neuron (0, 100 and 1000), and the mean spike rate of the pyramidal neurons (low, with about 16 spikes/s, medium, with about 45 spikes/s, and high, with about 60 spikes/s).

The obtained speedup varied between 9 and 4, depending on the spike rate and number of connections. When we consider the same number of connections per neuron and mean spike rate, the speedup decreases as we increase the number of neurons, due to the increase in the number of dispatched spikes. For the same reason, simulations with more synaptic connections per neuron and

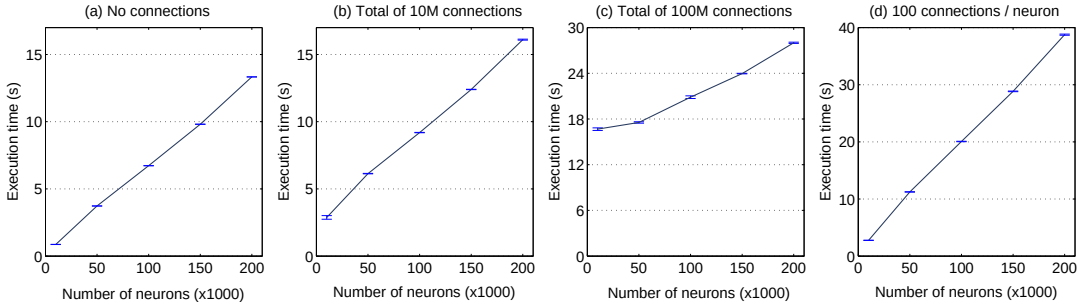


Figure 13. Scalability of the kernel as we increase the number of neurons (from 10k to 200k) in a network without connections (a), with fixed number of connections (b, c) and with 100 connections per neuron (d).

higher spike rates, have a lower speedup. For the simulations with 100k and 200k neurons we only considered the scenario with 100 connections per neuron, where we obtained speedups between 5 and 7, depending on the firing rate. The results were different from the results obtained in Section 5.3 due to the different firing rates used. Finally, with 100k and 200k neurons we could not simulate the case with 1000 connections per neuron, since the main memory in the computer was not enough to accommodate the complete simulation.

We also evaluated the scalability of the kernel as we increase the number of neurons from 10k to 200k, with a mean spike rate of 16 spikes per second. Figure 13 shows that in all scenarios the increase in the kernel execution time with the number of neurons was linear. For the simulations with no connections (a), with a total of 10M connections (b), and with 100M connections (c), the slope of the line is similar, with the execution times differing by a fixed amount, which is the time used to process the spikes in the synapses of the target neurons. With 100 connections per neuron (d), the slope is much higher, which is expected, since in addition to more neurons, there are also more spikes to process. We can conclude from these results that the kernel execution time is linearly dependent on the number of neurons and on the total number of connections, as we estimated in Section 4.3.

Another interesting point is that the kernel time increases more slowly than the problem size. When we increase the number of neurons by a factor of 4 (from 50k to 200k), we obtained an increase in execution time by a factor of 3.57 in (a), 2.62 in (b) and 1.59 in (c). This indicates that execution time is not limited by memory bandwidth or amount of GPU processors. It is limited by memory latency, since with more neurons the GPUs can overlap a larger number of memory requests, which is more evident in the scenario with more connections and, consequently, more spikes to process.

6. CONCLUSIONS

The proposed algorithm and its implementation enable the simulation of large-scale neuronal networks composed of 200k detailed neuronal models and 20M connections in a single computer with 2

commodity graphic boards. Our implementation permits a considerable degree of flexibility when constructing neuronal models, which can have different number of compartments and distribution of ionic channels. Moreover, the network can have an arbitrary connection pattern, allowing the execution of many types of large-scale simulations.

The simulation of the detailed neuronal models, without spike dispatching, was up to 40 times faster when using GPUs, compared with the execution using 4 CPU cores, showing that GPUs can be used in detailed and complex simulations. For the complete simulation, with spike dispatching, we obtained speedups between 5 and 10, which means that a single machine equipped with current generation GPUs can substitute a small cluster containing from 5 to 10 conventional machines.

Spike dispatching is currently managed by the CPU, since it is necessary to exchange messages between neurons executing on different GPUs. Our next step is to explore the feasibility of transferring part of the spike dispatching processing to the GPUs, reducing the performance bottleneck. The algorithm will need to circumvent the lack of efficient synchronization primitives between thread blocks in CUDA and the information exchange between neurons executing on different GPUs.

Next, we will develop algorithms that allow the simulation execution on multiple machines, which will bring other challenges, due to the high network latency. Efficient load-distribution and placement algorithm will be essential to enable the scalability of the simulation with the number of machines. Finally, we will develop tools that allow users to specify the neurons properties and connectivity using well known description languages, which will enable scientists with no or little knowledge of parallel programming to use the processing power available at budget graphics cards.

REFERENCES

1. Rolls ET. *Memory, Attention, and Decision-Making: A Unifying Computational Neuroscience Approach*. Oxford Univesity Press, 2007.
2. Rangan AV, Tao L, Kovačič G, Cai D. Large-scale computational modeling of the primary visual cortex. *Coherent Behavior in Neuronal Networks*, vol. 3. Springer Series in Computational Neuroscience, 2009; 263–296.
3. Djurfeldt M, Lundqvist M, Johansson C, Rehn M, Ekeberg O, Lansner A. Brain-scale simulation of the neocortex on the IBM Blue Gene/L supercomputer. *IBM Journal of Research and Development* January 2008; **52**(1/2):31–41.
4. Izhikevich EM, Edelman GM. Large-scale model of mammalian thalamocortical systems. *Proceedings of the National Academy of Sciences* March 2008; **105**(9):3593—3598.
5. Rolls ET, Loh M, Deco G, Winterer G. Computational models of schizophrenia and dopamine modulation in the prefrontal cortex. *Nature Reviews Neuroscience* September 2008; **9**:696–709.
6. Bower JM, Beeman D. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. Second edn., Springer-Verlag, 1998.
7. Koch C, Segev I ((eds.)). *Methods in Neuronal Modeling: From Ions to Networks*. 2nd edn., MIT Press, 1999.
8. Hines M, Carnevale N. Translating network models to parallel hardware in NEURON. *Journal of Neuroscience Methods* April 2008; **169**(2):425–455.
9. Markram H. The blue brain project. *Nature Reviews Neuroscience* February 2006; **7**:153–160.
10. Migliore M, Cannia C, Lytton W, Markram H, Hines M. Parallel network simulations with NEURON. *Journal of Computational Neuroscience* 2006; **21**:119–129.
11. Plesser HE, Eppler JM, Morrison A, Diesmann M, Gewaltig MO. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. *Euro-Par 2007 Parallel Processing, Lecture Notes in Computer Science*, vol. 4641. Springer Berlin / Heidelberg, 2007; 672–681.
12. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC. GPU computing. *Proceedings of the IEEE* 2008; **96**(5):879–899.
13. NVIDIA Corporation. *CUDA 2.1 Programming Guide* 2009.
14. Li H, Petzold L. Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the GPU. *International Journal of High Performance Applications* published online on June 16, 2009; .

15. Anderson JA, Lorenz CD, Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics* 2008; **227**:5342—5359.
16. Hardy DJ, Stone JE, Schulten K. Multilevel summation of electrostatic potentials using graphics processing units. *Parallel Computing* 2009; **35**:164—177.
17. Bernaschi M, Fatica M, Melchionna S, Succi S, Kaxiras E. A flexible high-performance lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. *Concurrency and Computation: Practice & Experience* 2010; **22**(1):1–14.
18. Bernhard F, Keriven R. Spiking neurons on GPUs. *ICCS'06: Proc. of the Int. Conference on Computational Science, Lecture Notes in Computer Science*, vol. 3994, Springer Berlin, 2006; 236–243.
19. Nageswarana JM, Dutta N, Krichmarb JL, Nicolaua A, Veidenbaum AV. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks* August 2009; **22**(5–6):791–800.
20. Izhikevich EM. Simple model of spiking neurons. *IEEE Transactions on Neural Networks* 2003; **14**:1569–1572.
21. Hodgkin AL, Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology* August 1952; **117**(4):500–544.
22. Hines M. Efficient computation of branched nerve equations. *International Journal Biomedical Computation* January 1984; **15**(1):69–76.
23. NVIDIA Corporation. *Whitepaper – nVidia's Next Generation CUDA Compute Architecture: Fermi* 2009.