Neural Networks 22 (2009) 791–800

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

2009 Special Issue

A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors☆

Jayram Moorkanikara Nageswaran[a,*], Nikil Dutt[a], Jeffrey L. Krichmar[b], Alex Nicolau[a], Alexander V. Veidenbaum[a]

[a] Department of Computer Science, University of California Irvine, Irvine, CA 92697-3435, United States

[b] Department of Cognitive Sciences, 3151 Social Science Plaza, University of California Irvine, Irvine, CA 92697-5100, United States

ARTICLE INFO

Article history:
Received 6 May 2009
Received in revised form 4 June 2009
Accepted 25 June 2009

Keywords:
Izhikevich spiking neuron
CUDA
Graphics processor
STDP
Data parallelism

ABSTRACT

Neural network simulators that take into account the spiking behavior of neurons are useful for studying brain mechanisms and for various neural engineering applications. Spiking Neural Network (SNN) simulators have been traditionally simulated on large-scale clusters, super-computers, or on dedicated hardware architectures. Alternatively, Compute Unified Device Architecture (CUDA) Graphics Processing Units (GPUs) can provide a low-cost, programmable, and high-performance computing platform for simulation of SNNs. In this paper we demonstrate an efficient, biologically realistic, large-scale SNN simulator that runs on a single GPU. The SNN model includes Izhikevich spiking neurons, detailed models of synaptic plasticity and variable axonal delay. We allow user-defined configuration of the GPU-SNN model by means of a high-level programming interface written in C++ but similar to the PyNN programming interface specification. PyNN is a common programming interface developed by the neuronal simulation community to allow a single script to run on various simulators. The GPU implementation (on NVIDIA GTX-280 with 1 GB of memory) is up to 26 times faster than a CPU version for the simulation of 100K neurons with 50 Million synaptic connections, firing at an average rate of 7 Hz. For simulation of 10 Million synaptic connections and 100K neurons, the GPU SNN model is only 1.5 times slower than real-time. Further, we present a collection of new techniques related to parallelism extraction, mapping of irregular communication, and network representation for effective simulation of SNNs on GPUs. The fidelity of the simulation results was validated on CPU simulations using firing rate, synaptic weight distribution, and inter-spike interval analysis. Our simulator is publicly available to the modeling community so that researchers will have easy access to large-scale SNN simulations.

Published by Elsevier Ltd

1. Introduction

Spiking neural network (SNN) models are emerging as a plausible paradigm for characterizing neural dynamics in the cerebral cortex (Brette et al., 2007; Maass & Bishop, 2001). Unlike firing rate-based models, SNN models incorporate the precise time structure of spike trains leading to many interesting properties such as temporal binding due to synchronized firing, and feed-forward propagation of spike pools as in synfire chains (Abeles, 1991). SNN models augmented with biologically accurate learning mechanisms such as competitive Hebbian learning (Song, Miller & Abbott, 2000) and axonal transmission delay have shown impressive learning, memory and adaptation capacities (Izhikevich, 2006; Song & Abbott, 2001). SNNs perform an event driven data processing to spike based events leading to faster system response (Thorpe, Delorme & Rullen, 2001). The SNN models have high biological fidelity, and can model many characteristics of brain architecture (Ananthanarayanan & Modha, 2007).

For understanding different dynamics in the SNNs, and to use it in real-time applications such as in robotics, it is essential to have a large-scale network model that operates in almost near real-time. Conventional processors do not have enough parallelism and memory bandwidth for real-time simulation of SNNs. Modern parallel architectures (such as clusters, super-computers, or high-performance processors) promise powerful alternatives for speeding spiking network simulation, but incur

☆ An abbreviated version of this article appeared in [Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., & Veidenbaum, A, (2009) Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. In *International conference on neural networks*], published under IEEE copyright. The source code for the GPU-SNN simulator can be downloaded at http://www.ics.uci.edu/~jmoorkan/project.

* Corresponding author. Tel.: +1 949 331 4006; fax: +1 949 824 4056.
E-mail addresses: jmoorkan@uci.edu (J.M. Nageswaran), dutt@uci.edu (N. Dutt), jeff.krichmar@uci.edu (J.L. Krichmar), nicolau@ics.uci.edu (A. Nicolau), Alex.Veidenbaum@uci.edu (A.V. Veidenbaum).

0893-6080/$ – see front matter. Published by Elsevier Ltd
doi:10.1016/j.neunet.2009.06.028

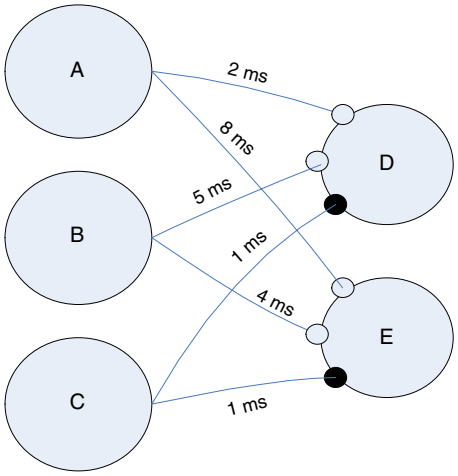


Fig. 1. A simplified illustration of the cortical network. A, B, C, D, E represent different neurons. The axonal delays (in ms) are annotated on the connections between neurons. Synaptic connections of neurons are represented as small circles and from an inhibitory neurons by solid small circles.

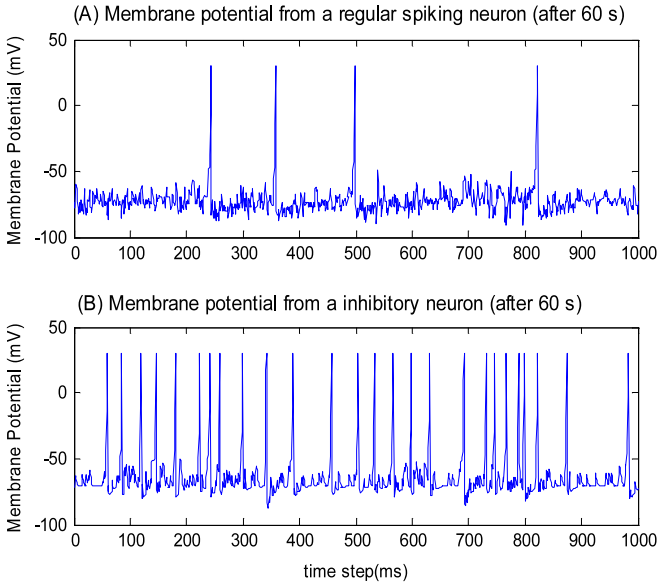


Fig. 2. Membrane potential produced by regular spiking and fast spiking neuron using the Izhikevich model.

high cost for purchase, maintenance and has size limitations. Graphics Processing Units (GPUs) have emerged as a powerful and cheap computational platform for the acceleration of diverse applications. Some of the recently developed GPUs include IBM CELL, NVIDIA CUDA, and ATI Stream Processor (Fatahalian & Houston, 2008). The Compute Unified Device Architecture (CUDA) from NVIDIA allows programmers to more easily harness the parallel processing capability of GPUs with standard C code. Some characteristics of the CUDA GPUs that make them suitable for simulating SNNs are: (1) extreme multithreading with thousands of threads running concurrently, (2) hardware mechanisms that allow automatic context switching between threads, minimizing idle time, and (3) specialized functional units that perform compute-intensive mathematical calculations (e.g., trigonometric functions) in hardware. The above characteristics allow parallel simulation of hundreds of thousands of neurons as lightweight threads on a GPU. One limitation of GPUs for simulating SNNs is the available memory bandwidth. Many biologically realistic SNN models tend to be memory-bounded, with a very low ratio of computation to communication; hence the overall performance is restricted by the maximum bandwidth achievable by the GPUs rather than the peak floating point operations.

In this article, we present strategies for efficient simulation of biologically realistic large-scale SNN models by incorporating Izhikevich neuron models (Izhikevich, 2003) on the NVIDIA GPU platform. The main challenges in simulating SNNs using GPUs are: (i) effective parallelism to optimize the GPU resources (processors, shared memory and memory bandwidth), (ii) effective handling of large fan-in/fan-out connections to neurons, and (iii) efficient usage of limited GPU memory for simulating large networks (more than 10^5 neurons and 10^7 synaptic connections). The main objective of this paper is to show the implementation of biologically realistic SNNs using CUDA GPUs, and various optimizations to achieve high simulation performance. In addition, we perform fidelity analysis of our GPU simulations (using measures such as firing rate, inter-spike-intervals, and synaptic weight distribution) to ensure that the GPU simulation results match the CPU simulations. Even though the focus of this work is on single GPU performance, we believe an approach that combines GPU computing and cluster computing capabilities can provide a cost-effective simulation platform for large-scale simulation of up to 50 million neurons (i.e., approaching 'rat-scale' cortical models).

In the rest of this paper we outline the Izhikevich simulation model in Section 2, present related work in Section 3, describe the GPU architecture in Section 4, present strategies for efficient mapping of SNNs to GPUs in Section 5, describe the PyNN like API for simulating large-scale SNN models on GPU in Section 6, and describe experimental results in Section 7.

2. Background

As shown in Fig. 1, the main components for the simulation of large-scale SNNs are: neurons for spike processing, axons and dendrites for spike communication, and synapses for learning and storage. In our simulator, the neuronal dynamics are modeled using the Izhikevich's simple spiking neuron model (Izhikevich, 2003) as it can generate a wide variety of neural responses compared to classical integrate-and-fire (I&F) neurons. At the same time the Izhikevich neuron incurs much less computational cost compared to the Hodgkin–Huxley model (Izhikevich, 2003). Izhikevich neurons are modeled by the following expressions:

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

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

$$\text{if} \quad (v >= +30\ \text{mV}) \quad \text{then } v = c; \quad u = u + d. \tag{3}$$

The variable v denotes the membrane potential of the neuron, u denotes the recovery variable, and I denotes the total input synaptic current to the neuron. The variables a, b, c, d are dimensionless constants taking different values according to the type of simulated neuron. Our SNN networks consist of 80% regular spiking excitatory neurons ($a = 0.02, b = 0.2, c = -65, d = 8$), and 20% fast spiking inhibitory neurons ($a = 0.1, b = 0.2, c = -65, d = 2$). The membrane potential response of these neurons is shown in Fig. 2. The axons in the simulator are modeled as loss-less cables with distance-dependent conduction delay. An example for the axonal conduction delay is shown in Fig. 1. Whenever neuron A fires, neuron D would receive the spike after a 2 ms delay, and neuron E after 8 ms. The axonal conduction delays facilitate the generation of stable, and time locked spatio-temporal neural firing patterns (Izhikevich, 2006). The point of contact between two neurons, termed the synapse, provides a means to adjust the strength of connection between two neurons. In our simulator we have incorporated the long-term memory changes by means of spike-timing dependent plasticity (STDP) (Kepecs, van Rossum, Song & Tegner, 2002). According to the STDP rule, the degree and sign of synaptic modification

is dependent on the exact timings between the firing at the pre-synaptic and post-synaptic side. The STDP mechanism forces the synaptic connections to compete with each other to control the firing of the post-synaptic neuron. This competition followed by potentiation of some synaptic connections and depression of other synaptic connections is one way for generating stable firing patterns in large-scale SNNs (Song et al., 2000).

3. Related work

Most previous works on accelerating SNN simulations mapped large-scale SNNs on distributed computers, or on dedicated hardware architectures (Jahnke, Schnauer, Roth, Mohraz & Klar, 1997; Khan et al., 2008). Some of the earliest work used hyper-cubic parallel computers for modeling SNNs based on I&F neurons (Jahnke et al., 1997; Niebur & Brettle, 1993). Existing SNN simulators such as NEST, PCSIM (see Brette et al. (2007) for more details on spiking neuron simulations) have demonstrated a parallel version that runs on simple clusters (Morrison, Mehring, Geisel, Aertsen & Diesmann, 2005; Plesser, Eppler, Morrison, Diesmann & Gewaltig, 2007). The IBM C2 simulator demonstrated a rat-scale cortical simulation (55 Million neurons with 442 Billion synapses) using a Blue-Gene supercomputer having more than 32K processors (Ananthanarayanan & Modha, 2007). Unfortunately the cost and development time make these approaches impractical for general purpose, large-scale simulations. The neuromorphic community has also built dedicated hardware for simulating SNNs. The Stanford Neurogrid (Merolla, Arthur, Shi & Boahen, 2007) approach simulates one million neurons using a multi-chip array, with each chip simulating 65K neurons. Vogelstein, Mallik, Culurciello, Cauwenberghs and Etienne-Cummings (2007) has demonstrated a multi-chip SNN system using an analog integrate-and-fire neuron chip (with 4800 neurons) and an FPGA for storing the synaptic weights (4 Million synapses). Even though the performance and power efficiency of these dedicated hardware approaches is superior to other techniques, the dedicated hardware approach suffers from limited programmability, and high-cost. SpiNNaker (Khan et al., 2008) deploys an application specific parallel processor interconnected by a network-on-chip communication fabric, resulting in an approach that combines the performance and ease of programmability for realizing SNNs; our GPU approach is general purpose and some of the techniques can be applied directly on the SpiNNaker chip.

To the best of our knowledge, our work is the first to demonstrate a general-purpose approach for simulation of biologically realistic spiking neural networks using the CUDA GPU platform. An abbreviated version of this article appeared in Nageswaran, Dutt, Krichmar, Nicolau and Veidenbaum (2009). Although prior work exists in applying older generation GPUs for simulating spiking neural networks (Bernhard & Keriven, 2006; Nageswaran, Dutt, Wang & Delbruck, 2009), most of these previous approaches use simple integrate-and-fire neurons, and are without biologically realistic neural network features (such as STDP and axonal conduction delay). Adding these features into a SNN simulation is essential for generating various brain dynamics; and these features make the model memory bandwidth intensive. In the remaining sections we analyze the modeling and performance of SNNs mapped on to GPUs.

4. GPU architecture

Fig. 3 shows a simplified view of the CUDA GPU architecture from NVIDIA (NVIDIA, 2008). It contains an array of Streaming Multiprocessors (SMs). Each SM consists of eight Scalar Processors (SPs), a Special Function Unit (SFU), a multi-threaded instruction unit, a 16 KB user-managed shared memory, and 16 KB of cache memory (8 KB constant cache and 8 KB texture cache).

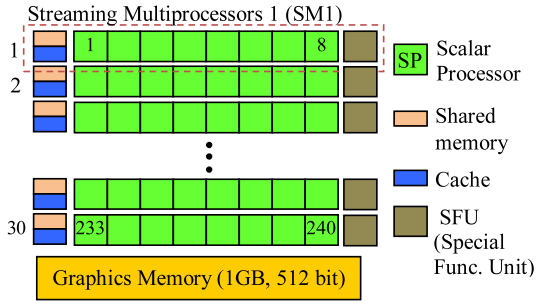


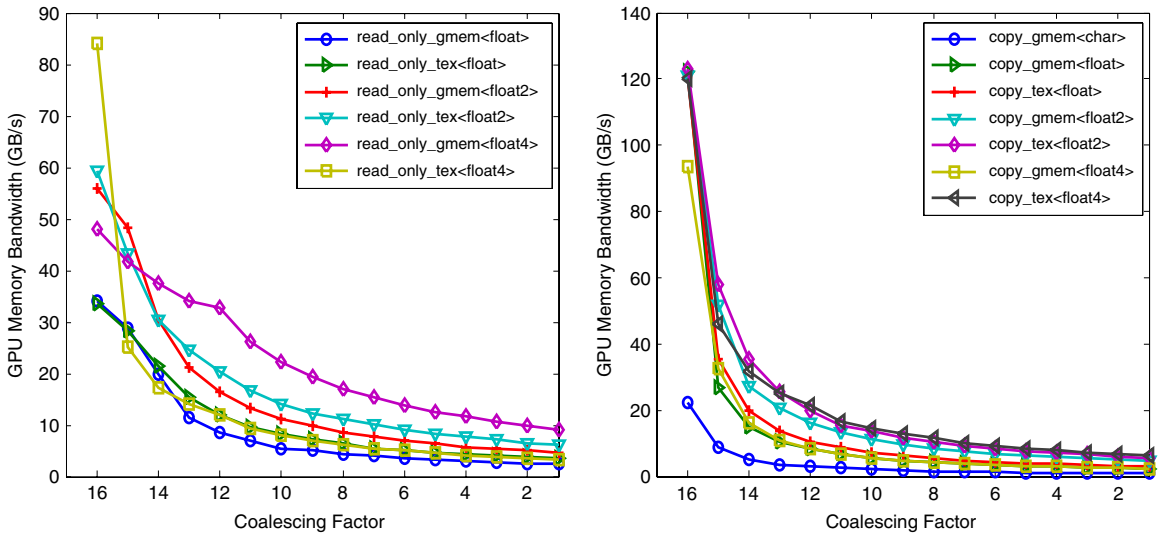
Fig. 3. An illustration of the NVIDIA CUDA architecture with 30 Streaming Multiprocessor (each having 8 Scalar Processor).

In our experiments we used a single NVIDIA GTX280 GPU card that consists of 240 scalar processors grouped into 30 SMs (each operating at 1.2 GHz). The sustained performance of the GTX280 GPU card is approximately 350 GFLOPS. Each SM has a hardware thread scheduler that selects a group of threads (called a *warp*) for execution. If any one of the threads in the warp issues a costly external memory operation, then the thread scheduler automatically switches to a new warp. At any instance of time, the hardware allows a very high number of threads (768 threads per SM in GTX280) to be simultaneously active. By swapping thread groups, the thread scheduler can effectively hide costly memory latency. Each GTX 280 GPU contains a 512-bit DDR3 interface to the graphics memory with a peak theoretical bandwidth of 143 GB/s. In comparison, the standard Pentium chipset with a 64-bit quad-pumped DDR3 interface gives a peak theoretical bandwidth of about 28 GB/s (i.e., 5.1 times slower than a GPU). Like all GPUs, there are many features in CUDA that trade-off generality and ease of programming for achieving very high-processing efficiency in certain circumstances that occur frequently in graphics applications. Fortunately, these same circumstances can be replicated, with some code transformations, in SNN implementations, and thus we can also take advantage of these features in GPUs.

4.1. GPU performance metrics

Now we discuss some of the metrics that influence the performance of SNNs on CUDA GPUs:

- Parallelism: To effectively use the GPU resources, the application needs to be mapped in a data-parallel fashion; each thread should operate on different data. Also, a large number of threads (in the thousands) need to be launched by the application to effectively hide the stalling effects caused while accessing GPU memory.
- Memory bandwidth: To achieve peak memory bandwidth, each processor should perform uniform memory access (e.g., thread0 accesses address0, thread1 accesses address0+4, thread2 accesses address0+8 etc.). If memory accesses are uniform, it is possible to group many memory accesses into a single large memory transaction (termed coalescing operation) achieving high memory bandwidth. In CUDA 1.2 compatible GPUs (and future families) memory coalescing is performed if all SPs within an SM access the same memory segment in any ordering (NVIDIA, 2008). We analyzed the impact of an uncoalesced operation by a simple experiment. In this experiment we measured the memory bandwidth for read-only operation and copy-operation under various operating modes and different values of coalescing factor. The results of our experiments are



(a) Read-only operation. (b) Copy operation.

Fig. 4. Bandwidth analysis of (a) read-only operation and (b) copy operation for various coalescing factors on GTX280 GPU card. `gmem` indicates regular global memory, `tex` indicates regular texture memory. float, float2 and float4 indicate different datatypes used for each memory operation.

shown in Fig. 4. A coalescing factor of 16 indicates that 16 memory operations were grouped into one global memory transaction. A high value of coalescing factor indicates better utilization of GPU memory bandwidth. For each plot we configured the memory transaction to use the regular global memory (*gmem*) or use the GPU texture memory (*tex*). Furthermore, for each case we varied the datatype from simple floats to vector floats like float2 and float4.[1] We can observe from Fig. 4 that the uncoalesced memory operations reduce the memory bandwidth by a factor of 10 compared to fully coalesced operations. Also we can observe from the Fig. 4(a) that for read-only operations, the texture based access can have very high bandwidth if memory accesses are fully coalesced. The SNN simulator presented in this paper tries to maximize coalesced operations by changing the data structure and the data access pattern.

- Memory usage: The memory used by various data structures in the simulator strongly influences the memory bandwidth and the scale of SNN simulations. In our paper, we employ techniques that minimize memory usage by incorporating sparse connectivity and by using the reduced Address-Event-Representation (AER) format for storing firing information (see Sections 5.3 and 5.4). Other compression techniques for eliminating redundancy can be applied to further reduce the memory usage (Morrison et al., 2005).
- Minimize thread divergence: By design, the current CUDA GPUs select a warp of 32 threads, and execute them using a single instruction register. Maximum performance can be achieved if all the threads within the warp execute the same instruction. If different threads within the warp follow different branches (which are termed divergent warps), then this will lead to sub-optimal performance.

It is important to note that the above factors are interrelated, and all four factors need to be optimized together for effective execution of applications on GPUs. For example a technique that improves the memory usage may reduce the overall parallelism resulting in lower effective performance. In the remaining sections of this paper we discuss various approaches to improve the GPU performance metrics for large-scale SNN simulations.

[1] Float2 and float4 indicates vectors of 2 and 4 floats respectively.

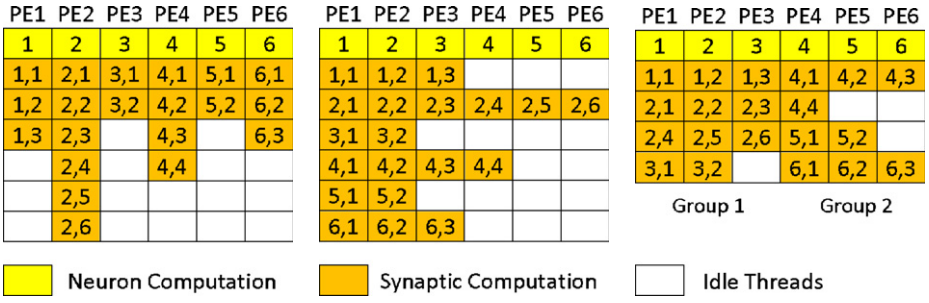
5. GPU mapping

We now present strategies for efficient mapping of SNNs onto GPUs.

5.1. Parallelism analysis

A SNN can be mapped onto an array of processing elements using three different approaches (Jahnke et al., 1997)

1. Neuronal parallelism (N-parallel) (Ananthanarayanan & Modha, 2007; Morrison et al., 2005; Plesser et al., 2007): Each neuron is mapped on a processing element and executed in parallel. The synaptic computation for each neuron is carried out sequentially on its processing element. This approach is most commonly used in CPU based SNN simulators, but the approach is not effective for GPUs as it leads to warp divergence. As an example, consider that neuron 1 (with 100 pre-synaptic connections) is mapped on thread1, and neuron 2 (with 200 pre-synaptic connections) is mapped on thread2. Because all threads within a warp should execute together (using single instruction register), thread1 will be busy waiting for thread2 to finish, leading to poor performance. An illustration of the mapping process based on the N-parallel model is shown in Fig. 5(a) for five processing elements. As we can see from the figure, the synaptic computation is unbalanced between threads because thread 1 takes 4 time units, and thread 2 takes 7 time units, and so on. This unbalanced operation leads to idle threads and reduced performance on the GPU.
2. Synaptic parallelism (S-parallel) (Jahnke et al., 1997; Niebur & Brettle, 1993): For a given neuron each synaptic connection is updated in parallel by different processing element. Thus the synaptic information is distributed over all processing elements. The neuron computation is carried out sequentially. The maximum parallelism is limited by the number of synaptic connections that need to be updated in a given time step and it varies for each neuron. A simple illustration to explain the problem in S-parallel approach is shown in Fig. 5(b). Due to the varying number of synapses per neuron and availability of large number of threads within a block (more than 128 threads), many threads remain idle resulting in decreased performance.



(a) N-parallelism. (b) S-parallelism. (c) NS-parallelism.

Fig. 5. Different kinds of parallelism in SNN simulators. Increasing rows indicate increasing time. Each column indicates a task that is executed in specific processing element (PE). Neuron computation is indicated by the number n, that denotes the neuron identification. Synaptic computation is indicated by the pair n,s, where s denotes the synaptic identification and n denotes the neuron identification.

3. Neuronal-synaptic parallelism (NS-parallel): Uses *both* N-parallel and S-parallel techniques but at different stages in the simulation. We employ the NS-parallel approach since it is a good fit for the GPU architecture. At each time step where the neuron information needs to be updated, the N-parallel strategy is adopted. Thus, every thread within the GPU updates different neuron information in parallel. Whenever a spike is generated, the S-parallel mapping is deployed for updating the synaptic state. S-parallel mapping can be easily applied within SMs at the warp level due to the availability of shared memory and fast synchronization. When performing S-parallel computation, a group of 16 threads (half-warp) or 32 threads (full-warp) coordinate to simulate all synaptic operations for one neuron, and the next group implements the synaptic computation for next neuron and so on. Grouping threads at warp-level during synaptic computation reduces the load balancing problem, increases the overall memory bandwidth due to coalesced accesses, and reduces the warp divergence problem (more details in Section 5.2). An example to illustrate the NS-parallel approach is shown in Fig. 5(c). Here we group threads to share the synaptic computation (PE1–PE3 is one group, and PE4–PE6 is the other group). In this example, the NS-parallel approach completes the computation much faster than other approaches.

5.2. Minimizing the impact of warp divergence

Warp divergence can happen in the SNN simulation if different threads within the same warp take different paths after a branch condition. If the code executed after a diverging condition is simple, then the impact due to warp divergence is minimal. On the other hand if the diverging condition takes a large number of cycles, then other threads in the warp go into null computation mode. An example is shown in Fig. 6(a), in which the GPU code (with large diverging loop) calls the function `do_firing()` whenever a neuron exceeds its threshold potential. When one or more neurons call `do_firing()`, which takes 100–200 cycles, all threads (within the warp) which did not have a firing event wait for the fired threads to finish the `do_firing()` code. We reduce the impact of warp divergence by buffering the information for diverging loop execution, and delaying the execution until sufficient data is available for all the threads to execute. An example is shown in Fig. 6(b). In this buffered scheme, each thread stores the fired neuron id in a local buffer (Line 3). After evaluating all the neurons for firing condition, each thread concurrently executes the `do_firing()` function (Line 7) using different ids (Line 5 and 8). This optimization leads to much better performance when evaluating fired neurons, and has been incorporated in our simulator. One main requirement for this optimization is availability of shared memory within SMs with atomic operations for synchronization.

(a) Large computation within diverging loop

```
1. i = threadIdx.x + blockIdx.x*blockDim.x
2. if ( membraneV[i] >= 30.0 ) {
3.    do_firing (i)          // 100-200 cycles
4. }
5. // repeat for all neurons
....
```

(b) Small computation within diverging loop

```
1. i = threadIdx.x + blockIdx.x*blockDim.x
2. if ( membraneV[i] >= 30.0 ) {
3. p=atomicAdd(&k,1);buffer[p]=i //5-10 cycles
4. } // repeat for all neurons
.... __syncthreads();
5. offset = threadIdx.x;
6. while (offset < k)
7.      do_firing (buffer[offset])
8.      offset=offset+blockDim.x
```

Fig. 6. Pseudo code showing the technique to minimize the impact of warp divergence by local buffering; (a) Normal code with large diverging warps (b) Buffered scheme with small diverging warps.

5.3. Sparse representation of network parameters

Simulating large-scale SNNs requires a large amount of memory to represent the network, and store its parameters. Without sparse-representation techniques the amount of required memory can be $O(N\,D\,M)$, where N = number of neurons, M = number of synaptic connection, and D = maximum axonal delay. By means of sparse-matrix representation the memory requirement can be brought down to $O(N.(M+D))$. A schematic of the data structures that are used for this representation is shown in Fig. 7. The representation is similar to an adjacency lists for directed graphs. Each neuron has a unique neuron id, the number of post-synaptic connections, and a list of post-synaptic connections. Each synaptic connection is identified by the (neuron id, synapses id) pair. The synapses id represents the position of the synapses in the post-synaptic neuron. For example whenever neuron 8 fires it has to send the spike to three post-synaptic neurons (length = 3, neuron id 1 at synaptic connection 1, neuron id 4 at synaptic connection 3, and neuron id 6 at synaptic connection 2). The synaptic connection for each neuron is sorted based on the delay to the destination neuron. Each spike is not transferred instantaneously, but reaches the destination neurons after a certain axonal delay corresponding to the axonal length of the connection. The delay information is stored in the parameters `delay count`, and `delay start`. The `delay count` at index k indicates the number of neurons with

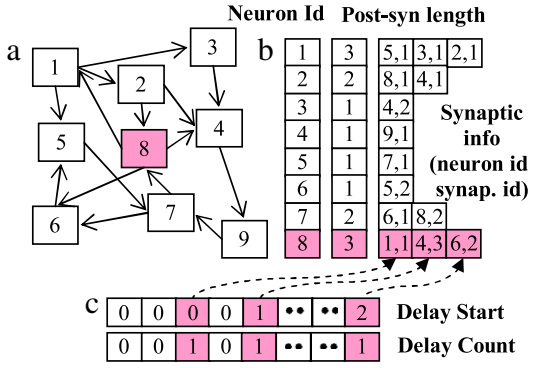


Fig. 7. Sparse representation synaptic connections and axonal delays: (a) Connectivity graph for a simple network (b) Neuron ids and the corresponding post synaptic connection in sparse representation. (c) Delay information for each neuron. The representation is similar to adjacency lists for directed graphs.

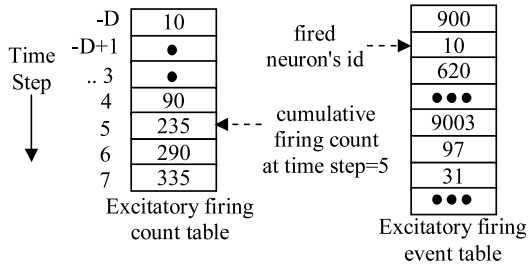


Fig. 8. Reduced AER Format for representing firing information. In the left table we indicate how many neurons have fired. In the right table, we indicate which neurons have fired.

a delay of k ms. The kth element of `delay start` identifies the first synaptic connection with a delay of k millisecond in the list of post-synaptic connections. As an example (see Fig. 7), we can identify all synaptic connections that have a delay of 20 ms for neuron id 8 using the parameters `delay count(8, 20) = 1`, and `delay start(8, 20) = 2` that correspond to neuron id 6 at connection number 2. By using the parameters `delay count` and `delay start` we can efficiently represent any arbitrary SNN model, and also retrieve the connection information quickly.

5.4. Improved event queue representation

A circular event queue mechanism is most commonly used in large-scale spiking network simulations (Ananthanarayanan & Modha, 2007; Plesser et al., 2007) to store the firing information. In this mechanism, whenever a neuron fires, the next set of synaptic events is added into the event queue. For a standalone application this approach is effective and has been incorporated in the simulator. For detailed fidelity analysis of the firing pattern, we store the firing information by means of the AER format (Merolla et al., 2007).

In the AER format, each spike is represented by an address-time pair. For example if three neurons (with ids 9003, 97, and 31) fired at time $t = 1$, then according to AER format it is stored as [1:9003], [1:97], [1:31]. But this approach leads to high memory overhead due to duplicate storing of time for each address if many neurons have the same time step value. In our current implementation we use a reduced AER format for representing the firing information (see Fig. 8) by removing the duplicate time entry for each address. Instead of storing the time along with each address we just store the cumulative count of fired neurons during each time step. This approach consists of two tables: firing count table, and firing address table. The firing count table consists of the number of excitatory neurons that have fired up until the current time step. The firing address table consists of a list of ids for the recently fired excitatory neurons. This approach requires about half of the memory compared to a conventional AER scheme. For finding out the neurons that fired at $t = 4$, we read the firing count table entry to know the starting count as 90 and the end count as 235. Hence a total of 145 neurons got fired at $t = 4$ and the neuron ids can be read from the firing address table starting from position 90. We adjust both the tables after every second of simulation by discarding older values and storing only the recent firing information.

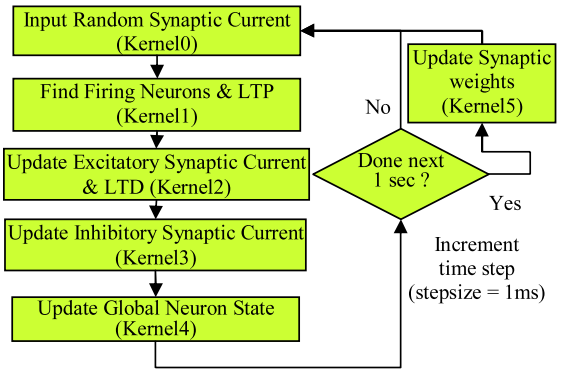


Fig. 9. Flowchart of the simulation using GPU showing various kernels. LTP (Long-Term Potentiation) represents the potentiation part of STDP. LTD (Long-Term Depression) represents the depression part of STDP. Every loop of the flowchart corresponds to one time step of the simulation. Kernel 5 is called after every second to update the synaptic weights. This simulates a slower rate of change of the synaptic weights.

5.5. Supporting large fan-in connection

In our SNN model, the number of input connections per neuron can range from 100 to 1000. In some neurons (e.g., Purkinje fibers) the number of input connections per neuron can be as large as 10,000. Such a large fan-in of synaptic inputs needs to be calculated for each neuron concurrently. A simple approach would have each fired neuron update the synaptic current of the post-synaptic neuron atomically. Atomic operation is essential because two neurons can simultaneously update the synaptic current to the same post-synaptic neuron. This approach is infeasible in current GPUs due to the lack of atomic floating point operations. Instead, we use a bit-vector based approach to realize the post-synaptic current calculation and the algorithm is outlined in the supplementary materials.

5.6. GPU simulation flow

The overall flow of the GPU version of the SNN simulator is shown in Fig. 9. The C function that is executed on the GPU is termed as 'kernel'. Each kernel is executed in parallel by all the threads in the GPU. The host CPU interfaces with the GPU to control the creation, execution, and termination of the kernel. In this version of the simulator, the GPU and CPU work in blocking mode (also called synchronous mode). In blocking mode, the CPU launches one kernel, and waits till the completion of the kernel call. In the future we plan to incorporate an asynchronous (or non-blocking) mode that allows the CPU to do more tasks concurrently along with the GPU. For each kernel, we experimented with various block sizes in the range of 30 to 120, with 128 threads in each block. The change in performance was very small for block sizes greater than 60. The simulation flow corresponds to the minimal spiking neural network model (Izhikevich, 2003). The network consist of N randomly connected excitatory (80%) and inhibitory (20%) neurons. For our experiments N ranges from 50K to 225K neurons.

Each neuron has M post-synaptic connections, where M ranges from 100 to 1000. *Kernel0* corresponds to initialization of thalamic input current for selected group of neurons. *Kernel1* implements threshold computation to create a list of fired neurons (explained in Fig. 6(b)), and LTP computation for the fired neurons. *Kernel2* and *Kernel3* implements the spike propagation part of the simulation. *Kernel4* calculates the total input current for each neuron and updates the global state of the network. A full description of different kernels is given in the supplementary materials. *Kernel5* is called every second to update all the synaptic weights using the accumulated synaptic weight changes due to LTP and LTD. This approach changes the synaptic weight at a slower-rate than the neurons (Izhikevich, 2006).

6. Programming interface and visualization

In order to enable rapid modeling of different topologies and configurations using GPU-SNN, we implemented a simple application programming interface in C++ that is similar to the PyNN programming interface (Davison et al., 2009). The PyNN programming interface is an effort by the computational neuroscience community and various simulator developers to establish a standard specification that allows a single configuration file written in Python to run on many different kinds of simulators (such as NEURON, NEST, PCSIM). For quickly constructing a large network of spiking neurons, the GPU-SNN APIs allow creation of a population of neurons and configurable connections between these population of neurons by means of projected connections. This is similar to the high-level object oriented interface that has been defined in the PyNN specification. A description of a simple program with detailed comments using the GPU-SNN API is shown in Listing 1. The listing is similar to the reference implementation given in Izhikevich (2006). The network contains 800 excitatory and 200 inhibitory neurons, with 100 synaptic connections from each neuron. The GPU-SNN API allows monitoring of the generated spikes in different groups. It is also possible to generate different kinds of interesting plots that describe the state of the network. The GPU-SNN API provides visualization tools to monitor the activity of a SNN (more details of visualization is given in the supplementary materials). As future work we plan to have full support for PyNN scripts on GPU-SNN, and also to incorporate interactive visualization features.

7. Results

7.1. Memory analysis

For a SNN model with N neurons, M number of synaptic connections per neuron, and maximum axonal delay of D, the memory required for representing different elements of the model is shown in Table 1. We used a 32-bit representation for floating point numbers, and large integers. A further saving in memory can be achieved by using half-floats (16-bit floating point numbers), or fixed-point representations (Jin, Furber & Woods, 2008), which will be explored in our future work. Based on the expressions in Table 1, the total memory required for various configurations has been plotted in Fig. 10. Each configuration is represented by the value of N (number of neurons), and M (number of post-synaptic connection per neuron). The value of N ranges from 50K to 225K (steps of 25K) for each value of M shown in the x-axis. The 1 GB limit line indicates the available memory in GTX-280 CUDA GPU. Using a GTX 280 CUDA GPU card with 1 GB of graphics memory, we were able to simulate a network with 225K neurons and 45 million synapses.

Table 1
Memory requirement (N = number of neurons, D = axonal delay, M = number of synapses per neuron).

SNN components	Memory required in words	Description
Neuron information	$(5N + \frac{MN}{32})$	u, v, a, d, firingtime, current input
Synaptic weights information	$(3 * N * M)$	s, sd, synapses firing time
Network representation, delays	$(N*M+N*D+3*N)$	Post-synaptic ids, delays and counts
Firing info (reduced AER)	$(50 * N)$	Maximum firing rate = 50 Hz
Firing info (circular buffer)	$(5 * N)$	Additional space factor = 5

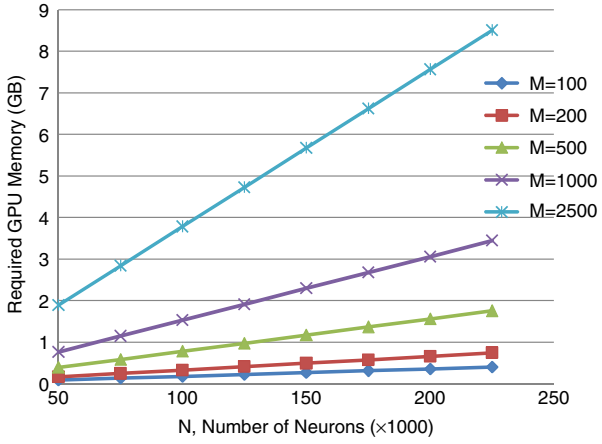


Fig. 10. Memory requirement for different configurations of SNNs on GPU. Each point represents a specific configuration M (number of post-synaptic connection per neuron) and N (number of neurons). The 4 GB limit corresponds to the maximum memory supported by a single high-performance TESLA GPU. A 4 GB T GPU card allows a SNN simulation of up to 225K neurons with 225M synapses.

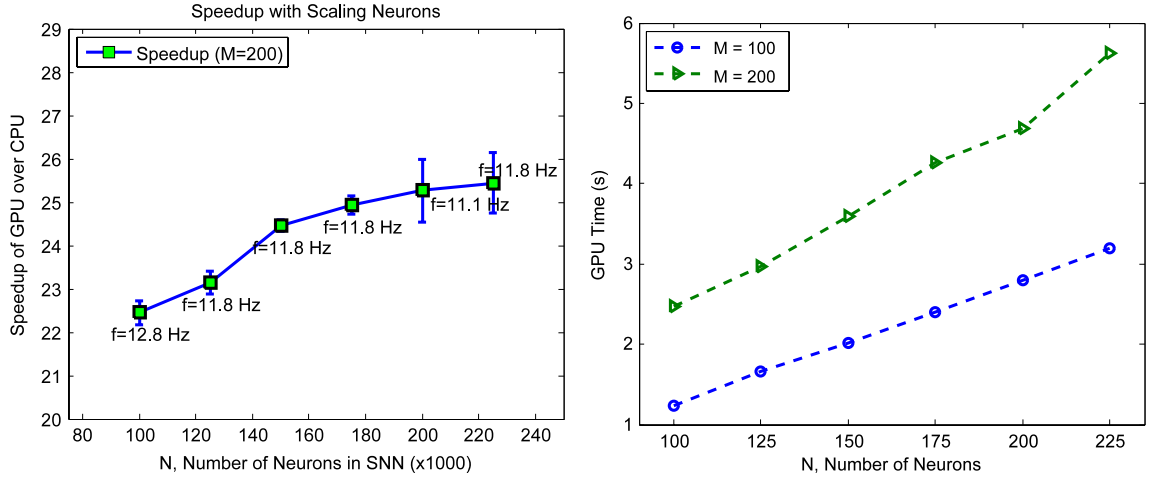
7.2. Scalability analysis

The performance of the simulator was evaluated by scaling the number of neurons, and by scaling the number of synaptic connections. The CPU version of the simulator (written in standard 'C' language) was based on the previous work reported in Izhikevich (2006). We investigated all the optimizations proposed in Section 5, comparing their potential for the single CPU machine.We included those that can help the CPU version as well, namely: sparse representation of network parameters discussed in Section 5.3, and improved event-queue representation discussed in Section 5.4. The CPU version was run on a standard workstation with Intel Core2 CPU 6400, operating at 2.13 GHz and having 4 GB of memory. An NVIDIA GTX 280 graphics card was used for running the GPU code. The system boots on a 64-bit version of Windows XP professional with CUDA 1.3 drivers. Both the GPU and CPU versions were compiled using Microsoft Visual Studio 2005 with the compiler options "/arch:SSE2 /Ox /Ob2 /Oi /Ot /Oy /fp:fast". For obtaining the speedup curves the simulator was configured to run in the GPU mode until the synaptic weight distribution becomes bimodal (Song et al., 2000). A snapshot of the simulator state was taken and then run separately in CPU only mode, and in GPU mode. The speedup curves were obtained by dividing the time taken by the CPU only mode and GPU mode for simulating 10 s of model time (10,000 times steps with 1 ms resolution) from the steady condition.

In Fig. 11(a) we show the speedup for different values of N and $M = 200$. For $M > 300$, the scale of the simulated network is limited to 100,000 neurons due to 1 GB memory limit, and hence is not included in measuring the network scalability. For

Listing 1. Listing of a sample programming using the GPU-SNN APIs similar to PyNN specification.

```
// create a network with N=1000 neurons, maximum number of synapses is 100,
// maximum delay is 20ms, name is global
CpuSNN s(1000, 100, 20, "global");

// create a population or group of neurons with the label "excit",
int g1=s.createInitGroup("excit", 800, 0.02, 0.2, -65.0, 8.0);
int g2=s.createInitGroup("inhib", 200, 0.1,  0.2, -65.0, 2.0);

// connect from group g2 to group g1, with an initial weight of -6.0,
// connection probability of 1.0 and delay min=1ms,max=1ms,
s.connect(g2,g1,"random", -6.0, 1.0, 1, 1);
s.connect(g1,g2,"random", +5.0, 0.20,  1, 20);
s.connect(g1,g1,"random", +6.0, 0.80,  1, 20);

// define the thalamic current into the network. select 0.1% of neurons
// at random and add a current of 20.0 to them
s.randomThalamicCurrent(0.1, 20.0);

// create a spike monitor. This enables storing of the generated
// spikes for specific population of neurons in a group.
s.spikeMonitor(g1);
s.spikeMonitor(g2);

for(int i=0; i < 10; i++) {
  // run the given network for 1(sec)+0(millisecond), in GPU_MODE
  s.runNetwork(1, 0, GPU_MODE);

  // Dump the synaptic weight histogram in the given file
  s.printHistogram("Histogram/synhist", true);

  // Dump the firing information in the given files
  s.rasterPlot(g1, "Raster/fireInfo1.m", true);
  s.rasterPlot(g2, "Raster/fireInfo2.m", true);
}
```



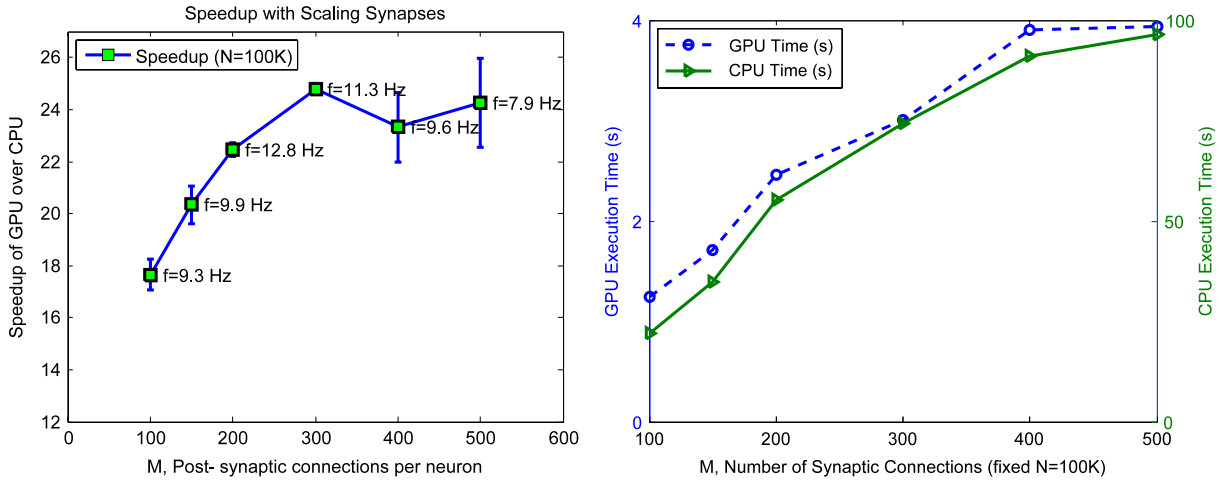
(a) Speedup with scaling neurons.

(b) GPU execution time with scaling neurons.

Fig. 11. (a) Speedup of GPU with respect to a single CPU for different network size (N) and synaptic connections per neuron, $M = 200$. (b) Actual execution time of GPU code for simulating 1 s of model time on networks with different number of neurons. Simulations were run 5 times with different random number seed. The notation $f = n$ Hz denotes the mean firing rate of 5 simulations. The error bar denotes the standard deviation and each point denotes the averaged results.

each network configuration, the average firing rate obtained is also represented. We can observe that the overall speedup does not vary significantly for various values of N ($N > 10^5$) and given value of M. The variation in the speedup curve is mainly due to the variation in the firing rate. An increase in the firing rate causes a slight improvement in the speedup. The speedup remains steady because the performance is mainly determined by the memory bandwidth and it saturates for large value of N ($N > 10^5$). A plot of the actual execution time from which the speedup curves are obtained is shown in Fig. 11(b).

The performance of the SNN simulator for scaling synaptic connections per neuron is shown in Fig. 12(a). A plot of the actual



(a) Speedup with scaling synapses.

(b) GPU execution time for scaling synapses.

Fig. 12. (a) Speedup of GPU over CPU for a network with 100K neurons and varying number of synaptic connection. (b) Actual execution time of GPU and CPU code for simulating 1 s of model time on network with varying number of synaptic connections. Simulations were run 5 times with different random number seed. The notation $f = n$ Hz denotes the mean firing rate of 5 simulations. The error bar denotes the standard deviation and each point denotes the averaged results.

execution time from which the speedup curves are obtained is shown in Fig. 12(b). Simulating 1 s of a network with $M = 100$ and $N = 10^5$ takes around 1.3 s on GPU and 23 s on CPU giving a speedup of 18. For larger values of M the speedup increases from 18 to around 25 due to improved synaptic parallelism. For $N = 100$ K and for large values of M ($M > 300$) it can be observed that the speedup curve flattens due to saturation in the mapped synaptic parallelism and memory bandwidth.

7.3. Fidelity analysis

The GPU model we implemented differs from the reference model (Izhikevich, 2006) in the following ways: implementation of STDP calculations (we use exponential functions supported by the GPU hardware), network representation, firing information representation, etc. (as detailed in Section 5). Also the GPUs only support single-precision floating point arithmetic, and not all operations meet the IEEE 754 standard. Thus a direct comparison of the SNN state (e.g., membrane potentials and synaptic weights) between the reference implementation (Izhikevich, 2006) and the GPU simulation is difficult because the SNN state can change significantly even if one spike is altered (Izhikevich & Edelman, 2008).

To ensure the accuracy and fidelity of our GPU implementation, we compared various neuronal metrics with the original MATLAB implementation. The metrics are: difference in average firing rate, difference in the synaptic weights of excitatory connections, and difference in the inter-spike intervals (ISI) for excitatory neurons and for inhibitory neurons. Since the reference model itself (written in Matlab) is significantly slow, we evaluated the fidelity metrics only for a small set of configurations ($N = 1000, 3000$ with $M = 100$). The metrics were collected for 20 s of model time after allowing the simulations to run for 900 s (15 min of model time). All metrics have been consolidated after running each configuration 5 times. We observed that the firing rates were similar in both cases. We also tested the fidelity using the inter-spike interval (ISI) and synaptic weight distributions as metrics and verified that the two implementations were not significantly different (further details are presented in the supplementary materials).

8. Conclusion

In this paper we presented strategies for efficient mapping of realistic, large-scale spiking neural network simulation models on GPUs. The GPU-SNN simulator provides high flexibility, high-performance, and low-cost for large-scale simulation of up to 250,000 neurons. The simulator was only 1.5 times slower than real-time for a network of 10^5 neurons having 10^7 synaptic connections with an average ring rate of 9 Hz. The GPU implementation (on one NVIDIA GTX-280 with 1 GB of memory) was up to 24 times faster than a CPU version for simulation of 100K neurons with 50 million synaptic connections, ring at an average rate of 7 Hz. The performance is limited by the memory bandwidth supportedby the GPU hardware rather than the number of scalar processors. For simulation of larger networks (around 1–10 Million neurons) a cluster of GPUs can be employed, building upon the strategies outlined in this paper.

Efficient mapping of realistic SNNs to the GPU platform opens the door for ubiquitous use of the GPU platform for this class of simulation. The performance of the simulator was analyzed for different configurations of the network. The fidelity of the GPU SNN implementation was analyzed using different metrics, such as firing rate and synaptic weight distribution. In order to enable easy mapping of generic topology and configuration of SNNs on GPU we implemented a high-level C++ based API. This API is similar to the PyNN based programming interface specification.

Our GPU-SNN approach and the high-level API should make large-scale SNN simulations available to a wider audience of modelers. Our simulator is publicly available and can be downloaded from www.ics.uci.edu/~jmoorkan/project. Our future works include enabling full support for PyNN based specification, extending the model to work on cluster of GPUs, increasing the number of synaptic connection per neuron, and finally supporting other interesting synaptic dynamics (Morrison, Diesmann & Gerstner, 2008) such as short-term plasticity.

Acknowledgements

This work was supported in part by DARPA subcontract 801888-BS. We would also like to thank NVIDIA Professor Partnership Program for donating the 9800GX2 and GTX-280 boards. Many thanks to Micah Richert and Michael Wei for their support during this project.

Appendix. Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.neunet.2009.06.028.

References

Abeles, M. (1991). *Corticonics: Neural circuits of the cerebral cortex*. Cambridge University Press.

Ananthanarayanan, R., & Modha, D. S. (2007). Anatomy of a cortical simulator. In *Proceedings of the 2007 ACM/IEEE conference on supercomputing*.

Bernhard, F., & Keriven, R. (2006). Spiking neurons on GPUs. In *International conference on computational science ICCS 2006* (pp. 236–243).

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J., et al. (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, *23*(3), 349–398.

Davison, A. P., Brderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: A common interface for neuronal network simulators. *Frontiers in Neuroinformatics*.

Fatahalian, K., & Houston, M. (2008). A closer look at GPUs. *Communications of ACM*, *51*(10), 50–57.

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, *14*(6), 1569–1572.

Izhikevich, E. M. (2006). Polychronization: Computation with spikes. *Neural Computation*, *18*(2), 245–282.

Izhikevich, E. M., & Edelman, G. M. (2008). Large-scale model of mammalian thalamocortical systems. *Proceedings of the National Academy of Sciences*, *105*(9), 3593–3598.

Jahnke, A., Schnauer, T., Roth, U., Mohraz, K., & Klar, H. (1997). Simulation of spiking neural networks on different hardware platforms. In *Intern. conference on artificial neural networks* (pp. 1187—1192) *Vol. 1327*.

Jin, X., Furber, S., & Woods, J. (2008). Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In *IEEE international joint conference on neural networks,* (pp. 2812–2819).

Kepecs, A., van Rossum, M. C. W., Song, S., & Tegner, J. (2002). Spike-timing-dependent plasticity: Common themes and divergent vistas. *Biological Cybernetics*, *87*(5–6), 446–458.

Khan, M., Lester, D., Plana, L., Rast, A., Jin, X., & Painkras, E. et al. (2008). SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *IEEE international joint conference on neural networks* (pp. 2849–2856).

Maass, W., & Bishop, C. M. (2001). *Pulsed neural networks*. MIT Press.

Merolla, P. A., Arthur, J. V., Shi, B. E., & Boahen, K. A. (2007). Expandable networks for neuromorphic chips. *IEEE Transactions on Circuits and Systems I*, *54*(2), 301–311.

Morrison, A., Diesmann, M., & Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics*, *98*, 459–478.

Morrison, A., Mehring, C., Geisel, T., Aertsen, T. G. A., & Diesmann, M. A. (2005). Advancing the boundaries of High-Connectivity network simulation with distributed computing. *Neural Computation*, *17*(8), 1776–1801.

Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., & Veidenbaum, A. (2009). Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. In *International conference on neural networks*.

Nageswaran, J. M., Dutt, N., Wang, Y., & Delbruck, T. (2009). Computing spike-based convolution on GPUs In *IEEE international symposium on circuits and systems*.

Niebur, E., & Brettle, D. (1993). Efficient simulation of biological neural networks on massively parallel supercomputers with hypercube architecture. In *NIPS* (pp. 904–910).

NVIDIA (2008). CUDA programming manual version 2.0. See appendix a for technical specifications.

Plesser, H., Eppler, J., Morrison, A., Diesmann, M., & Gewaltig, M. (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In *Euro-Par 2007 parallel processing* (pp. 672–681).

Song, S., & Abbott, L. F. (2001). Cortical development and remapping through spike timing-dependent plasticity. *Neuron*, *32*(2), 339–350.

Song, S., Miller, K. D., & Abbott, L. F. (2000). Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, *3*(9), 919–926.

Thorpe, S., Delorme, A., & Rullen, R. V. (2001). Spike-based strategies for rapid processing. *Neural Networks*, *14*(6–7), 715–725.

Vogelstein, R. J., Mallik, U., Culurciello, E., Cauwenberghs, G., & Etienne-Cummings, R. (2007). A multichip neuromorphic system for spike-based visual information processing. *Neural Computation*, *19*(9), 2281–2300.