Preliminary Technical Report
Efficient Simulation of Large-Scale Spiking Neural Networks Using Single CUDA Graphics Processors

Jayram Moorkanikara Nageswaran Micah Richert, Michael Wei, Nikil Dutt, Jeffrey Krichmar*

April 11, 2009

Abstract

Neural network simulators that take into account the spiking behavior of neurons are useful for studying brain mechanisms and for engineering applications. Spiking Neural Network (SNN) simulators have been traditionally simulated on large-scale clusters, super-computers, or on dedicated hardware architectures. Alternatively, 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, Izhikevich neuron based large-scale SNN simulator that runs on a single GPU. The GPU implementation (on NVIDIA GTX-280 with 1GB 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 7Hz. 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. We intend to make our simulator available to the modeling community so that researchers will have easy access to large-scale SNN simulations.

1 Introduction

Spiking neural network (SNN) models are emerging as a plausible paradigm for characterizing neural dynamics in the cerebral cortex [1, 2]. 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 syn-fire chains [3]. SNN models augmented with biologically accurate learning mechanisms such as competitive Hebbian learning [4] and axonal transmission delay have shown impressive learning, memory and adaptation capacities [5, 6]. SNNs perform an event driven data processing to spike based events leading to faster system response [7]. The SNN models have high biological fidelity, and can model many characteristics of brain architecture [8].

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 models that operates 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 require careful tuning of the applications to achieve good performance. 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 [9]. The Compute Unified Device Architecture (CUDA) from NVIDIA allows programmers to more easily harness the parallel processing capability of GPUs with

*Email: jmoorkan at uci.edu, micah at salk.edu, mwei at uci.edu, jkrichma at uci.edu dutt at uci.edu

standard C code. Some characteristics of the CUDA GPU family that makes it 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 light weight 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 SNNs models incorporating Izhikevich neuron models [10] 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 connections to neurons, and (iii) efficient usage of limited GPU memory for simulating large networks (more than 105 neurons and 107 synaptic connections) using sparse representations. The main objective of this paper is to show the implementation of biological 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 ('rat-scale' cortical model).

In the rest of this paper we outline the Izhikevich simulation model (Section 2), followed by study of related work (Section 3), and the rest of the paper describes GPU architecture, and strategies developed for efficient mapping of SNNs to GPUs.

2 Background

As shown in Figure 1, the main components for 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 neurons [10] as it can generate 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 Hodgkin-Huxley model. Izhikevich neurons are represented by the following expressions:

```
v' = 0.04v2 + 5v + 140 - u + I                    (1)
u' = a(bv - u)                                    (2)
if ( v = +30 mV)  then { v = c and u = u + d }    (3)
```

The variable v denotes the membrane potential of the neuron, and u denotes the recovery variable. The variables a,b,c,d are dimensionless constants taking different values according to the type of neuron being simulated. Our SNN networks consists 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 Figure 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 Figure 1. Whenever neuron A fires, neuron D would receive the spike after a 2ms delay, and neuron E after 8 ms. The axonal conduction delays facilitate the generation of stable, and time locked spatial-temporal neural firing patterns [5]. 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) [11]. According to the STDP rule (also termed Competitive Hebbian 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. STDP mechanism forces the synaptic connections to compete with each

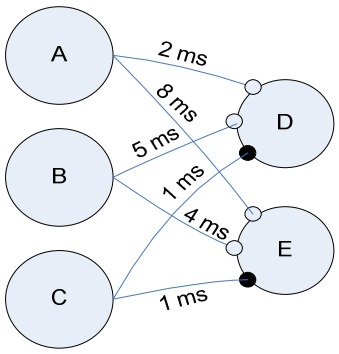

Figure 1: A simplifed illustration of the cortical network. The neurons are indicated by the labels A,B,C,D and E. The axonal delays (in ms) are annotated on the axonal connection between neurons. All axons of inhibitory neurons (e.g.,C) have a fixed delay of 1 ms. Synaptic connections of a neuron are represented as small circles

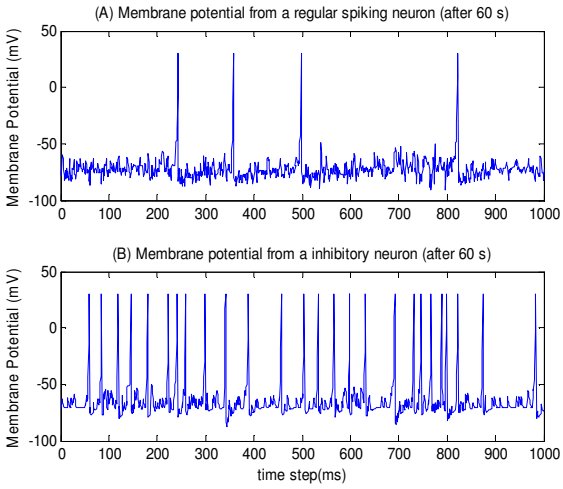

Figure 2: Membrane potential produced by Izhikevich neurons: a. Regular Spiking Neuron b. Inhibitory Neuron

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 [4].

3 Related Work

Most previous work on accelerating SNN simulations mapped large-scale SNNs on distributed computers, or on dedicated hardware architectures [12, 13]. Some of the earliest work used hyper-cubic parallel computers for modeling SNNs based on I&F neurons [12, 14]. Existing SNN simulators such as NEST, PCSIM (see [2] for more details on spiking neuron simulations) have demonstrated a parallel version that runs on simple clusters [15, 16]. IBM C2 simulator demonstrated a rat-scale cortical simulation (55 Million neurons with 442 Billion synapses) using Blue-Gene supercomputer having more than 32K processors [8]. 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 [17] approach simulates one million neurons using a multi-chip array, with each chip simulating 65K neurons. Vogelstein et. al [18] 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 [23] [13] 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. Although prior work exists in applying older generation GPUs for simulating spiking neural networks [19, 20], 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 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 aspects on SNN on GPUs.

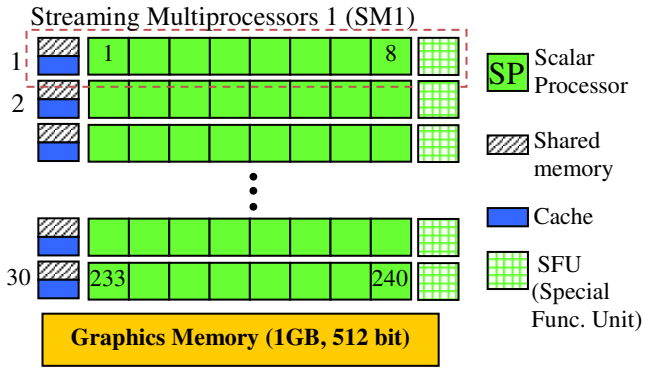

Figure 3: A simplifed illustration of the cortical network. The neurons are indicated by the labels A,B,C,D and E. The axonal delays (in ms) are annotated on the axonal connection between neurons. All axons of inhibitory neurons (e.g.,C) have a fixed delay of 1 ms. Synaptic connections of a neuron are represented as small circles

4 GPU Architecture

Figure 3 shows a simplified view of the CUDA GPU architecture from NVIDIA [21]. It contains an array of Streaming Multiprocessors (SMs). Each SM consists of eight floating-point Scalar Processors (SPs), a Special Function Unit (SFU), a multi-threaded instruction unit, a 16KB user-managed shared memory, and 16KB of cache memory (8KB constant cache and 8 KB texture cache).

In our experiments we use a single NVIDIA GTX280 GPU card that consists of 240 scalar processors grouped into 30 SMs (each operating at 1.2 GHz). The peak theoretical performance of the GTX280 GPU card works out to approximately 900 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 group issues a costly external memory operation, then the thread scheduler automatically switches to a new thread group. At any instant of time, the hardware allows a very high number of threads (768 threads per SM in GTX280) to be active simultaneously. 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 143GB/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. 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 have uniform memory access (e.g., thread0 accesses address0, thread1 accesses addr0+4, thread2 accesses addr0+8 etc.). If memory accesses are uniform, it is possible to group many memory accesses into a single large memory access (termed coalescing operation) achieving high memory bandwidth .

- Memory usage: The memory used by various data structures in the simulator strongly influences the memory bandwidth and scale of SNN simulations. In our paper, we employ techniques that

minimize memory usage by incorporating sparse connectivity and by using reduced Address-Event-Representation (AER) format for storing firing information (see Section 5.3 and 5.4). Other compression techniques for eliminating redundancy can be applied to further reduce the memory usage [15].

- Minimize thread divergence: By design, the current CUDA GPUs selects a warp of 32 threads, and executes 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 for effective execution on GPUs. For example a technique that improves the memory usage may reduce the overall parallelism resulting in lower effective performance.

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 [12]

1. Neuronal parallelism (N-parallel) [8, 16, 15]: Each neuron is mapped on a processing element and computed in parallel. The synaptic computation for each neuron is carried out sequentially on its processing element. This mapping leads to warp divergence and is ineffective for GPUs. 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.

2. Synaptic Parallelism (S-parallel) [12, 14]: For a given neuron each synaptic connection is updated in parallel by different processing element. Thus 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 connection that need to be updated in a given time step.

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 where the synapses need to be updated. S-parallel mapping can be easily applied within SMs due to the availability of shared memory and fast synchronization. When performing S-parallel computation, a group of 16 or 32 threads coordinate to simulate all synaptic operations for one neuron, and the next group implements the synaptic computation for next neuron and so on. This leads to coalesced memory access of synaptic information that improves the overall memory bandwidth performance.

5.2 Minimizing 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 a busy waiting mode.

(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
```

Figure 4: 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

An example is shown in Figure 4(a), in which the GPU code (with large diverging loop) calls the function do_firing() whenever a neuron exceeds its threshold potential. Since do_firing() takes 100-200 cycles, all threads (within the warp) which did not have a firing event waits 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 Figure 4(b). In this buffered scheme, each thread stores 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.

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 ?(NMD), 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 ?(N(M+D)). A schematic of the data structures that are used for this representation is shown in Figure 5.

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 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 Figure 5), we can identify all synaptic connections that has a delay of 20 ms for neuron id 8 using the parameters delay_start(8,20) = 2, and delay_count(8,20) = 1. By using both delay count and delay start we can efficiently represent any arbitrary SNN model, and also retrieve the connection information quickly.

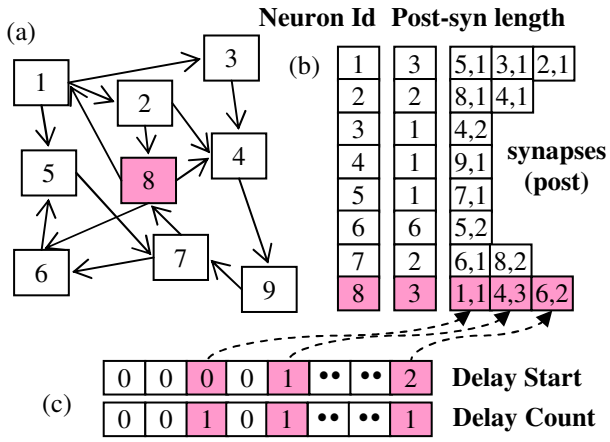

Figure 5: Sparse representation synaptic connections and axonal delays: (a) Connectivity graph for a simple network (b) Neuron ids and the corresponding post synaptic connection in sparse representation. (c) Delay information for each neuron. The representation is similar to adjacency list for directed graphs

5.4 Improved Event queue representation

A circular event queue mechanism is most commonly used in large-scale spiking network simulations [8, 16] 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 [17].

In the AER format, each spike is represented by an address-time pair. But this approach leads to high memory overhead as we need to store the time of firing along with each address. In our current implementation we use a reduced AER format for representing the firing information (See Figure 6). This approach consists of two tables: firing count table, and firing event table. The firing count table consists of the number of excitatory neurons that have fired up until the current time step. The firing event table consists of a list of ids for the recently fired excitatory neurons. This approach requires about half of the memory compared to the traditional AER scheme. We adjust both the tables after every second of simulation by discarding older values and storing only the recent firing information.

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 outlined in Algorithm 1 to realize the post-synaptic current calculation. We term the bit vector as I_fire vector, which represents the input firing status of each neuron. Each bit of the I_fire represents the presence (I_fire[x]=1) or absence (I_fire[x]=0) of firing at the input synaptic connection. The most common scenario is that either 0, 1 or 2 bits of I_fire vector is set in every time step, indicating a sparse input firing. Algorithm 1 first scans the I_fire at the word level, then at the byte level and finally at bit level. If none of the inputs are firing this approach incurs a small overhead of about 8 instruction cycles (for an input synaptic count of 128 and 32-bit loading and comparison operations). This approach is memory efficient and has low computation overhead.

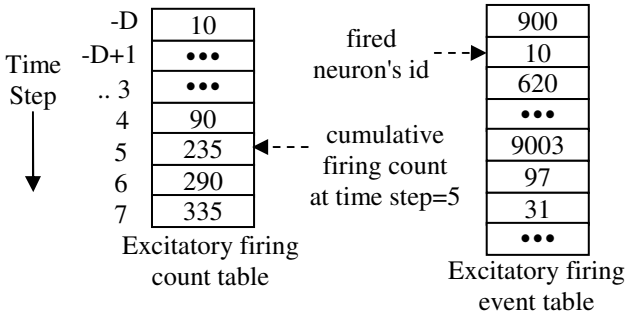

Figure 6: Reduced AER Format for representing firing information. In the left table we indicate how many neurons have fired. In the right table, we indicate which neurons have fired

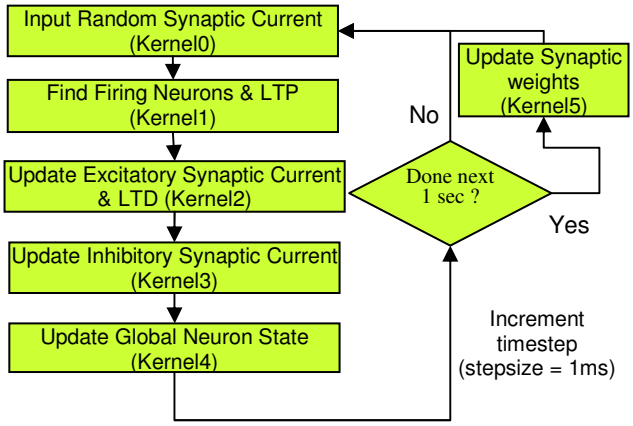

Figure 7: Pseudo code showing the technique to minimize the impact of warp divergence by local buffering, (a) Normal code with large diverging warps (b) Buffered scheme with small diverging warps

Algorithm 1: Synaptic Current Calculation in GPU

Inputs: nid => Neuron Id,
I_fire => Input Fired Vector
s[i][j] => weight of i^{th} neuron, j^{th} synapses
len[i] => number of synaptic connections
Output: I_sum => Total synaptic current
Require: find_one[x] => pre-computed 256 entry table that returns the position of the first set bit in a given byte (e.g. find_one[0x10]=4, find_one[0x77]=0

```
0.  I_sum=0, y_end = ceil(len[i]/32)
1.  for y=0:(y_end-1)
2.      part_I = read32(I_fire, y)     // Read y^th 32 bit
                                       // from I_fire vector
3.      x = 0
4.      while part_I ≠ 0
5.          byte_I = byte(part_I, x)   // Read x^th byte
6.          while byte_I ≠ 0
7.              idx = find_one[byte_I]
8.              set byte_I(idx) ←'0'
9.              I_sum = I_sum + s[nid][y*32+x*8+idx]
10.         part_I(x) ← 0; x = x+1;
11.  return I_sum
```

Figure 8: Synaptic current update algorithm

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

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

5.6 GPU Simulation Flow

The overall flow of the GPU version of the SNN simulator is shown in Figure 8. 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 future we plan to incorporate asynchronous mode (or non-blocking) 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 [10]. 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. The amount of change in the synaptic weight (using STDP rule) is accumulated during each time step; and the weight is updated once a second by Kernel5 (Figure 7) such that synaptic weight changes at a slower-rate than the neurons [5].

6 Results

6.1 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. Further saving in memory can be achieved by using half-floats (16-bit floating point numbers), or fixed-point representations [22], 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 Figure 9. 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 250K (steps of 25K) for each value of M shown in the x-axis. The 1GB limit line indicates the available memory in GTX-280 CUDA GPU. Using a GTX 280 CUDA GPU card with 1GB of graphics memory, we were able to simulate a network with 225K neurons and 45 million synapses.

6.2 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

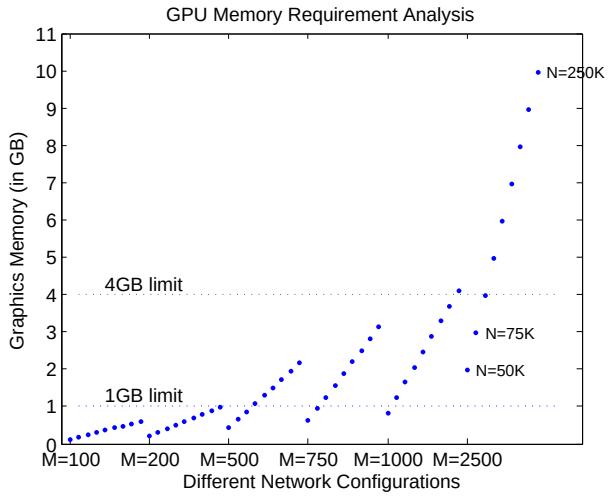

Figure 9: Memory requirement for different configurations of SNNs on GPU. Each point represents a specific configuration M (number of post-synaptic connection per neuron) and N (number of neurons). The 4GB limit corresponds to the maximum memory supported by a single high-performance TESLA GPU. A 4GB Tesla GPU card allows a SNN simulation of up to 225K neurons with 225M synapses

the previous work reported in [5]. 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 DELL 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 [4]. A snapshot of the simulator state is 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 seconds of model time (10,000 times steps with 1ms resolution) from the steady condition. In Figure 10 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, and hence is not included in measuring the network scalability. For 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 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$>10^5$).

The performance of the SNN simulator for scaling synaptic connections per neuron is shown in Figure 11. For M=100 and N=105, the speedup is limited to 18. The GPU takes 15 seconds to simulate 10 seconds of model time. For larger values of M the speedup jumps from 18 to around 25 due to increases in the available synaptic parallelism. For N=100K 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.

6.3 Fidelity Analysis

The GPU model we implemented differs from the reference model [5] 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

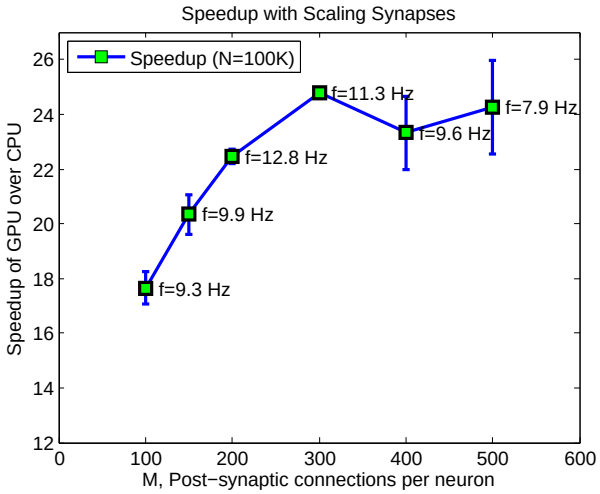

Figure 10: Speedup of GPU with respect to a single CPU for different network size (N) and different synaptic connections per neuron (M). Simulations were run 5 times with different random number seed. The notation f = n Hz denotes the mean firing rate of 5 simulations. Error bar denotes the standard deviation of the speedup

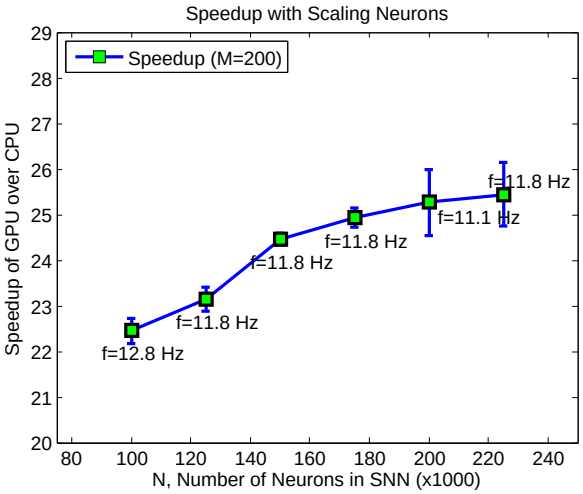

Figure 11: Speedup of GPU over CPU for a network with 100K neurons and varying number of synaptic connection. Simulations were run 5 times with different random number seed. The average firing rate is reported as f = n Hz. Error bar denotes the standard deviation of the speedup.

Firing Rate Metrics	N=1000,M=100	N=1000,M=100	N=3000,M=100	N=3000,M=100
	Matlab	GPU	Matlab	GPU
Excitatory Neurons (mean and std)	3.1423 (0.4934)	3.1693 (0.7034)	3.8242 (0.1413)	3.3855 (0.0843)
Inhibitory Neurons (mean and std)	24.95 (3.3683)	22.0345 (4.5293)	31.5863 (1.0140)	24.9593 (0.5713)

Table 2: Comparison of Firing Rate between Matlab and GPU simulation (mean and standard deviation collected from 5 runs of simulation

floating point arithmetic, and not all operations meet the IEEE 754 standard. Thus direct comparison of the SNN state (e.g., membrane potentials and synaptic weights) between the reference implementation [5] and the GPU simulation is difficult because the SNN state can change significantly even if one spike is altered [23].

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 seconds of model time after allowing the simulations to run for 900 seconds (15 minutes 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 (see Table 2). We also tested the fidelity using inter-spike interval (ISI) and synaptic weight distributions as metrics and verified that the two implementations were not significantly different (see Table 3).

Metrics	N=1000, M=100	N=3000, M=100
Synaptic Weights	0.992	0.099
ISI (Excitatory)	0.799	0.144
ISI (Inhibitory)	0.677	0.261

Table 3: Comparison of distribution of synaptic weights and inter-spike-interval (ISI) between Matlab and GPU (p-value using Kolmogorov-Smirnov test. Data consolidated from 5 runs of simulation)

7 Conclusion

In this paper we presented strategies for efficient mapping of realistic, large-scale spiking neural network simulation models on GPUs. We believe this is the first piece of work that demonstrates efficient mapping of realistic SNNs to the GPU platform, and opens the door for ubiquitous use of the GPU platform for this class of simulations. The performance of the simulator was analyzed for different configurations of the network. Also, the fidelity of the GPU SNN implementation was analyzed using different metrics (like firing rate and synaptic weight distribution) ensuring the accuracy of the simulations. The GPU based spiking network simulator provides high flexibility, performance and low-cost for large-scale simulation (of up to 250,000 neurons). The simulator was only 1.5 times slower than a real-time for a network of 105 neurons having 107 synaptic connections with an average firing rate of 9Hz. The GPU implementation (on one NVIDIA GTX-280 with 1GB of memory) was up to 24 times faster than a CPU version for simulation of 100K neurons with 50 million synaptic connections, firing at an average rate of 7Hz. The performance is limited by the memory bandwidth supported by 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. In the near future we plan to release APIs that allow GPU SNN models to be used in diverse simulation environment (MATLAB, C/C++). Our approach and the subsequent release of an API should make large-scale SNN simulations available to a wider audience of modelers.

References

[1] W. Maass and C. M. Bishop, *Pulsed Neural Networks*, 2001.

[2] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. Bower, M. Diesmann, A. Morrison, P. Goodman, F. Harris, M. Zirpe, T. Natschlger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. Davison, S. E. Boustani, and A. Destexhe, "Simulation of networks of spiking neurons: A review of tools and strategies," *Journal of Computational Neuroscience*, vol. 23, no. 3, pp. 349–398, Dec. 2007.

[3] M. Abeles, *Corticonics: Neural Circuits of the Cerebral Cortex*. Cambridge University Press, Feb. 1991.

[4] S. Song, K. D. Miller, and L. F. Abbott, "Competitive hebbian learning through spike-timing-dependent synaptic plasticity," *Nat Neurosci*, vol. 3, no. 9, pp. 919–926, 2000.

[5] E. M. Izhikevich, "Polychronization: Computation with spikes," *Neural Comput.*, vol. 18, no. 2, pp. 245–282, 2006.

[6] S. Song and L. F. Abbott, "Cortical development and remapping through spike timing-dependent plasticity," *Neuron*, vol. 32, no. 2, pp. 339–50, Oct. 2001, PMID: 11684002.

[7] S. Thorpe, A. Delorme, and R. V. Rullen, "Spike-based strategies for rapid processing," *Neural Networks: The Official Journal of the International Neural Network Society*, vol. 14, no. 6-7, pp. 715–25, 2001, PMID: 11665765.

[8] R. Ananthanarayanan and D. S. Modha, "Anatomy of a cortical simulator," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. Reno, Nevada: ACM, 2007, pp. 1–12.

[9] K. Fatahalian and M. Houston, "A closer look at GPUs," *Commun. ACM*, vol. 51, no. 10, pp. 50–57, 2008.

[10] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–72, 2003, PMID: 18244602.

[11] A. Kepecs, M. C. W. van Rossum, S. Song, and J. Tegner, "Spike-timing-dependent plasticity: common themes and divergent vistas," *Biological Cybernetics*, vol. 87, no. 5-6, pp. 446–58, Dec. 2002, PMID: 12461634.

[12] A. Jahnke, T. Schnauer, U. Roth, K. Mohraz, and H. Klar, "Simulation of spiking neural networks on different hardware platforms," *In ICANN 1997, Int. Conf. on Artificial Neural Networks*, vol. 1327, pp. 1187—1192, 1997.

[13] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras, and S. Furber, "SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor," in *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, 2008, pp. 2849–2856.

[14] E. Niebur and D. Brettle, "Efficient simulation of biological neural networks on massively parallel supercomputers with hypercube architecture," in *NIPS*, 1993, pp. 904–910.

[15] A. Morrison, C. Mehring, T. Geisel, T. G. A. Aertsen, and M. A. Diesmann, "Advancing the boundaries of High-Connectivity network simulation with distributed computing," *Neural Comput.*, vol. 17, no. 8, pp. 1776–1801, 2005.

[16] H. Plesser, J. Eppler, A. Morrison, M. Diesmann, and M. Gewaltig, *Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers*, 2007, pp. 672–681.

[17] P. A. Merolla, J. V. Arthur, B. E. Shi, and K. A. Boahen, "Expandable networks for neuromorphic chips," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 54, no. 2, pp. 301–311, 2007.

[18] R. J. Vogelstein, U. Mallik, E. Culurciello, G. Cauwenberghs, and R. Etienne-Cummings, "A multichip neuromorphic system for Spike-Based visual information processing," *Neural Comput.*, vol. 19, no. 9, pp. 2281–2300, 2007.

[19] J. M. Nageswaran, Y. Wang, and T. Delbruck, "Computing spike-based convolution on GPUs," in *Accepted for ISCAS*, 2009.

[20] F. Bernhard and R. Keriven, *Spiking Neurons on GPUs*, 2006, pp. 236–243.

[21] "NVIDIA programming manual version 2.0. see appendix a for technical specifications," Tech. Rep., 2008.

[22] X. Jin, S. Furber, and J. Woods, "Efficient modelling of spiking neural networks on a scalable chip multiprocessor," in *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, 2008, pp. 2812–2819.

[23] E. M. Izhikevich and G. M. Edelman, "Large-scale model of mammalian thalamocortical systems," *Proceedings of the National Academy of Sciences*, vol. 105, no. 9, pp. 3593–3598, Mar. 2008.

A Appendix: Details of different algorithm in simple C or pseudocode

In this section we will describe the details of the kernel and the corresponding algorithms used for implementing each of the kernels.

Algorithm 1: CPU version: Find Firing Neurons

```
for(int i=0; i < N; i++) {
  if (new_v[i] >= 30.0) {
   new_v[i] = -65.0;
   u[i] += d[i];
   firingTable[2*(sec_fire_cnt)]   = t;
   firingTable[2*(sec_fire_cnt)+1] = i;
   sec_fire_cnt++;
   if(i >= Ne)
      sec_inh_cnt++;

   for(int j=0; j < Npre_exc[i]; j++) {
      int pos_ij = i*PRE_FACTOR*M+j;
      if(!(timeStep-t_pre_firing_time[pos_ij]>=0)) {
         {cout << "Two many spikes at t=" << t << " (ignoring all)";N_firings=1;}
      }
      if((timeStep-t_pre_firing_time[pos_ij])/TAU<25)
         t_sd[pos_ij] += LTP(timeStep-t_pre_firing_time[pos_ij]);
    }
}
```

Algorithm 2: GPU Kernel1: Find Firing Neurons

```
    __shared__ volatile int fireCnt;          // counts fired neuron
    __shared__ volatile int IfireCnt;         // counts inhibitory fired neurons
    __shared__ int fireTable[FIRE_CHUNK_CNT]; // local table to store fired neurons
    // ASSUMPTION: This code assumes local table can store all firing within a BLOCK

int gid=blockIdx.x*blockDim.x+threadIdx.x;
int gnthreads=blockDim.x*gridDim.x;

for (int i=blockDim.x*blockIdx.x; i < N; i += gnthreads) {
    int  myI = i + threadIdx.x;
    bool needToWrite = false;

    if (myI < N)
       needToWrite = (gpu_v[myI] >= 30.0);
    if(needToWrite)
      fireId = atomicAdd((int*)&fireCnt, 1);

    if (needToWrite && (fireId<(FIRE_CHUNK_CNT))) {
       fireTable[fireId] = myI;           // store ID of the fired neuron
       if((myI >= Ne)) {
          atomicAdd((int*)&IfireCnt, 1);  // fired neuron is inhibitory
       }
       needToWrite = 0;
     }
     __syncthreads();
  }

  // we copy from local table to global table
  if(fireCnt) {
     //write from local table to global table
       gpu_newFireUpdate(fireTable, (fireCnt-IfireCnt), IfireCnt, timeStep);
       if(gpu_with_stdp)
          gpu_updateLTP(fireTable, (fireCnt), timeStep);
    }
  }
```

Algorithm 3: Kernel2: Update Neuron Synaptic Current

Input:
shTimingTable ↔ Store recent firing count table in shared memory
D ↔ Maximum Delay of Axon
delay ↔ Store the network delay parameters
gpuPostSyn ↔ Stores the post-synaptic connection information
globalFiringTable ↔ Array having the entire firing info

Data:
Store Id of currently processing neuron ↔ shFiredIdTable
Store delay length in shared memory ↔ shDelayLength
Store delay index Start in shared memory ↔ shDelayIndexStart

```
1  for kPos ← shTimingTable[0] to shTimingTable[D] do
2      if threadIdx.x < NUM THREADS/WARP SIZE then
           // Get time, delay parameters for nid
3          nid(warpId) ← put neuron Id from globalFiringTable
4          tD(warpId) ← getTime(nid, shTimingTable)
5          delayLength(warpId) ← getDelayLength(nid, tD)
6          delayIndexStart(warpId) ← getDelayIndexStart(nid, tD)
7      syncthreads();
       // all threads within WARP would update synatic current from
       // same presynaptic neuron to different post synaptic neuron
8      if threadIdWithinWarp ≥ delayIndexStart(warpId) then
9          continue;
10     preNid = nid(warpId)
11     postNid = gpuPostSyn(preNid, offset, "neuron")
12     postSid = gpuPostSyn(preNid, offset, "synapse")
13     setSynapticCurrent(postNid, postSid)
       // jump to the next fired neuron
14     kPos=kPos+(NUM THREADS*NUM BLOCKS/WARP SIZE)
```