GPU-Based Simulation of Spiking Neural Networks with Real-Time Performance & High Accuracy

Dmitri Yudanov, *Member, IEEE*, Muhammad Shaaban, *Member, IEEE*, Roy Melton, *Member, IEEE*, and Leon Reznik, *Member, IEEE*

***Abstract* — A novel GPU-based simulation of spiking neural networks is implemented as a hybrid system using Parker-Sochacki numerical integration method with adaptive order. Full single-precision floating-point accuracy for all model variables is achieved. The implementation is validated with exact matching of all neuron potential traces from GPU-based simulation versus those of a reference CPU-based simulation. A network of 4096 Izhikevich neurons simulated on an NVIDIA GTX260 device achieves real-time performance with a speedup of 9 compared to simulation executed on Opteron 285, 2.6-GHz device.**

***Index Terms*—GPU, CUDA, STDP, spiking neural network, high accuracy, parallel computing, shared memory.**

I. Introduction

A spiking neural network (SNN) is a model of a biological neural network with a simplified process of synaptic transmission. Neurons communicate with each other by spikes, modeled as time-stamped potential pulses. The accuracy of a spike time depends on the choice of a numerical integration system classified by Brette *et al.* [1] into the following categories. Clock-driven (synchronous) systems evaluate model variables only at fixed points in time. The resolution of the time grid, defined by the magnitude of a time step, determines the simulation accuracy and affects the execution time. More complex event-driven (asynchronous) systems update variables at the exact time of a spike event. The accuracy of the event time in these systems is not tied to a precision of any time grid, but depends on floating-point format chosen (double or single precision). Hybrid systems combine advantages of event-driven and clock-driven systems. They refresh the model variables at fixed points in time, but yet they process events at the exact time. Two identical SNNs excited with identical stimuli, but implemented as a clock- and event-driven systems do not produce the same spiking pattern unless a time step in the clock-driven implementation is small enough to achieve the desired accuracy [1], [2].

Some biological mechanisms, for example spike time dependent plasticity (STDP), require accurate timing. STDP simulation in a clock-driven system is contaminated with quantization error and thus may result in incorrect evolution of the network topology due to inability of the system to distinguish between long-term potentiation and long-term depression [1].

The Parker-Sochacki [3] (PS) numerical integration method, recently applied by Stewart and Bair [2] to the biologically plausible phenomenological neuron model developed by Izhikevich [4] (IZ), provides an accuracy appropriate for simulation of SNNs with biological mechanisms requiring exact event timing. In fact, such simulations achieve full double-precision integration accuracy.

This research applies graphical processing units (GPUs) and Compute Unified Device Architecture (CUDA), which offers for GPU-based computing a powerful development framework integrated with the C language. GPUs are cost effective commodity devices designed to exploit parallel shared memory-based floating-point computation. They provide memory access speeds superior to those of commodity CPU-based systems. High-speed memory access is essential in the simulations of neural networks because all model variables have to be updated every iteration. These features make GPUs attractive for SNN simulations compared to other solutions based on programmable logic, integrated circuits, custom shared memory solutions, and cluster message passing computing systems.

Successful implementations of CUDA GPU-based SNN simulations have already been demonstrated in a number of publications. Nageswaran *et al.* reported a real-time simulation of leaky integrate-and-fire (LIF) SNN [5] and near real-time simulation of IZ-based SNN with STDP model [6]. In the latter work the simulation time, the network size, and the connectivity density are emphasized, but not the accuracy. Moreover, verification of the GPU-based implementation with the reference one is performed indirectly. The difference between GPU-based and the reference simulation increases with the network size. Various factors are listed as causes of this error, but the exact origin is left unclear.

Fidjeland *et al.* [7] presented a real-time implementation of IZ-based SNN for the cluster-oriented network

Manuscript received February 7, 2010.

D. Yudanov is with Rochester Institute of Technology, Department of Computer Engineering, Rochester, NY 14623 USA (e-mail: dxy7370@gmail.com).

M. Shaaban is with Rochester Institute of Technology, Department of Computer Engineering, Rochester, NY 14623 USA (e-mail: meseec@rit.edu).

R. Melton is with Rochester Institute of Technology, Department of Computer Engineering, Rochester, NY 14623 USA (e-mail: Roy.Melton@mail.rit.edu).

L. Reznik is with Rochester Institute of Technology, Department of Computer Science, Rochester, NY 14623 USA (e-mail: lr@cs.rit.edu).

978-1-4244-8126-2/10/$26.00 ©2010 IEEE

topologies. No verification with reference implementation is performed.

Both above works use the original implementation by Izhikevich [4] based on an Euler-type numerical integration method, a synchronous system, and a one-ms time step. As a result, quantization error is introduced.

Tiesel *et al.* [8] reported an OpenGL implementation of a planar integrate-and-fire SNN with nearest neighbor connectivity. The forth order Runge–Kutta (RK4) integration method in a synchronous system is used. The verification of results from NVIDIA and ATI GPUs with results of an exact closed-form solution and Matlab RK4 solution is performed. However, the error introduced by the binary floating point standard itself is not differentiated. Consequently, the origin of the error is left unclear.

This novel implementation addresses both issues: simulation accuracy and verification error. It provides a real-time parallel simulation of IZ-based SNNs with the PS method, hybrid system, and full single-precision integration accuracy on CUDA-enabled GPUs. The simulation results are verified directly. They exactly match that of the reference implementation within the domain of IEEE-754 standard. Examples of potential applications of this work are high-accurate real-time neural systems, high-precision robotics, and scientific simulations.

The second section introduces the reader to the main concepts of the system: PS method, IZ neuron model, and CUDA. The third section provides insight into design and implementation of this project. Section four concentrates on the results and analysis of the system. Finally, the last section draws overall conclusions and proposes future work.

II. Essential Concepts

A. Parker-Sochacki Numerical Integration Method

The Parker-Sochacki (PS) [3] numerical integration technique is based on application of the Maclaurin series to a solution of differential equations with an initial value problem (IVP),

$$y'(t) = \frac{dy}{dt} = f\big(t, y(t)\big), \qquad y(t_0) = y_0, \quad t \in [t_0 - \alpha, t_0 + \alpha] \tag{1}$$

The method was developed based on the Picard iteration [9] under the assumption that the solution function is locally Lipschitz continuous in y and continuous in t (Picard–Lindelöf theorem, [10]), and therefore can be described with power series. Consequently, based on the fact that next coefficient in the series can be represented with the derivative of previous coefficient,

$$\sum_{p=0}^{\infty}(p+1)y_{p+1}t^p = \sum_{p=0}^{\infty} y_p' t^p\,, \qquad y_p = \frac{y^{(p)}(0)}{p!}, \tag{2}$$

and after substituting Eq. (2) in Eq. (1), the IVP Eq. (1) can be described in terms of power series:

$$\sum_{p=0}^{\infty}(p+1)y_{p+1}t^p = f\left(t, \sum_{p=0}^{\infty} y_p t^p\right). \tag{3}$$

Provided that f is a linear function, $f\big(t, y(t)\big) = ky(t) + b$, Eq. (3) becomes (constant term is temporary dropped):

$$\sum_{p=0}^{\infty}(p+1)y_{p+1}t^p = k\left(\sum_{p=0}^{\infty} y_p t^p\right), \tag{4}$$

which can be represented with pseudo code.

```
y = y0;  t = 1;
FOR:  p = 0,      p < N,      p++:
    yp = k*yp/(p + 1);
    y = y + yp*t;                                   (Listing 1)
    t = t × Δt;
END
y = y + b;
```

Code in Listing 1 exhibits loop level parallelism (LLP) and parallel reduction, which can be exploited if all coefficients are pre-calculated.

However, provided that f is a quadratic function, $f\big(t, y(t)\big) = ay^2(t) + by(t) + c$, after series multiplication, Eq. (3) becomes:

$$\sum_{p=0}^{\infty}(p+1)y_{p+1}t^p = a\sum_{p=0}^{\infty}\left(\sum_{i=0}^{p} y_i y_{p-i}\right)t^p + b\left(\sum_{p=0}^{\infty} y_p t^p\right), \tag{5}$$

which can be represented with pseudo code.

```
y = y0;  t = 1;
FOR:  p = 0,      p < N,      p++:
    d = 0;
    FOR:  i = 0,      i ≤ p,      i++:
        d = d + yi*yp−i;
    END                                             (Listing 2)
    yp = (ad + byp)/(p + 1);
    y = y + yp*t;
    t = t × Δt
END
y = y + c;
```

Exploiting parallel computation is problematic in this case because of linearly scaled convolution, which introduces loop-carried circular dependence. Partial parallelism still can be exploited in the convolution itself and in term $by_p/(p+1)$.

PS method allows a representation of an IVP with partial sums, where number of summands defines solution accuracy. PS method can be used for systems of simultaneous equations. Although parallel techniques can be applied with PS method, a discrete convolution (Cauchy product) appearing as a result of power operations reduces LLP. At the same time, in contrast to the finite order methods, PS method provides an adaptive order for a given error tolerance. Error tolerance is defined as a largest acceptable difference between values of an integration variable from two consecutive PS iterations. Consequently, execution time of computation naturally depends on the Lipschitz constant of the local solution function.

B. Izhikevich neuron membrane model

Targeting large-scale cortical SNN simulations, the phenomenological IZ model [4] was designed with the goal of providing simultaneous biological plausibility and computational simplicity at the expense of parametric space transformation and consequent change in definitions of parameters compared to those of classical Hodgkin-Huxley model [11]. The model has been gaining popularity especially for SNNs originally targeting leaky integrate-and-fire (IF) models, Lapicque [12]. The model equations are as follows, Izhikevich *et al.* [13]:

$$C\frac{dv}{dt} = k(v - v_{rest})(v - v_{thresh}) - u + I$$
$$\frac{du}{dt} = a(b(v - v_{rest}) - u) \quad (6)$$
$$if\ v \geq v_{peak}: \begin{cases} v = c \\ u = u + d \end{cases}$$

In these equations C represents the membrane capacitance; v, membrane potential; v_{rest}, resting membrane potential; v_{thresh}, threshold potential; v_{peak}, action potential escape limiting value; u, recovery variable that models Na^+ and K^+ currents; I, injected current; a, a parameter that describes time scale of recovery variable u; b, a parameter that describes sensitivity of u to subthreshold fluctuations of membrane potential; c, a parameter that describes after-spike reset value due to hyperpolarizing outward K^+ current (typically set to a value less than v_{rest}); d, a parameter that describes after-spike reset of u; and k, describes sensitivity of v to the fluctuations of itself.

The model functions in the following way: injected current I is integrated as a charge on the membrane capacitance C. This process results in membrane potential fluctuations v. If v crosses v_{thresh}, the quadratic part of the equation, $k(v - v_{rest})(v - v_{thresh})$, accelerates membrane potential dynamics, which results in a spike. At the same time, this high spiking value of v accelerates u, which provides negative feedback to v and to itself playing a role of K^+ ionic current at that time. In contrast to v, which gives a quadratic dependence to its acceleration, the equation that governs u has only first power dependence on v. Thus, u shaped by a and b follows v. As a consequence, v has to be artificially reset in order to keep it in the plausible range. Hence, once v reaches v_{peak}, it is reset to value c. At the same time, u is incremented rather than reset, and therefore it "memorizes" previous spike dynamics and affects the refractory period. Variable u without term $b(v - v_{rest})$ would play a role similar to the one K^+ conductance plays in IF model with spike frequency adaptation. However, this term gives it the functionality of controllable response to membrane potential dynamics, and therefore enriches the model with various types of dynamics [4].

C. Compute Unified Device Architecture

CUDA supports heterogeneous computing with asynchronous concurrent execution of CPU (host) and GPU (device) threads. GPU code is executed as a kernel. A kernel divides computation into parallel operations and separates GPU code from the CPU code. The division into parallel tasks is based on notions of a thread, block, and grid. A thread is a copy of an executed kernel at runtime with its own state and register set. A block is a collection of threads. A grid is a collection of blocks (Fig. 1).

Threads can be packed into blocks using 1D, 2D, or 3D indexing, and blocks can be packed into a grid using 1D or 2D indexing. The resulting indices, thread ID, and block ID respectively, are convenient for task-parallel and data-parallel computations. Each data piece can be allocated to threads/blocks based on these IDs. Thus, various levels of parallelism with various granularities can be exploited.

Each GPU is a collection of streaming multiprocessors (SM), which execute blocks of threads in arbitrary order. Hence, SNN size and the density of synaptic connections can be easily scaled with the number of blocks.

Threads are executed on SM cores in arbitrary order in small batches called warps; each consists of 32 threads. The

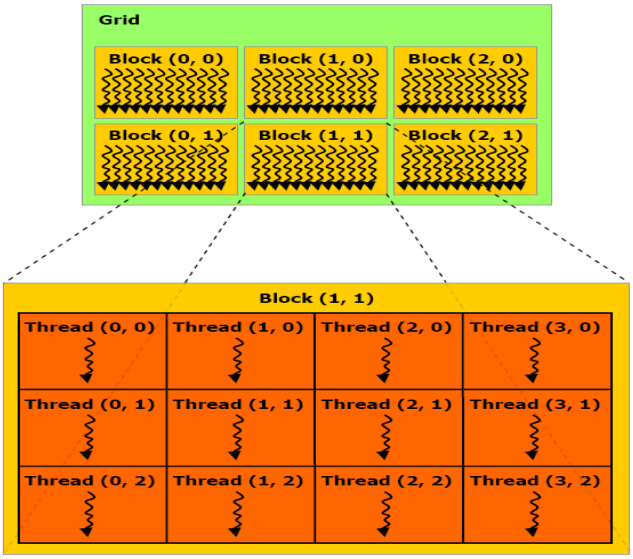

Fig. 1. CUDA thread hierarchy [17]. Threads are organized in blocks, and blocks are organized in a grid with a desired dimensionality. The kernel code is executed on the grid.

SM scheduler issues a single instruction to the entire warp. Threads in a warp are free to take different execution paths (branch divergence). However, these execution paths are processed in sequence until they reach a common execution path and continue in parallel from that point. Potential sources of branch divergence in PS SNN implementation are handling of spiked neurons, variability of synaptic events per neuron, and variability of PS steps per neuron.

A CUDA device has several memory types, which have their own distinct properties: not-cached large-capacity DRAM-type high-latency global memory, accessed by any thread; fast on-chip small-capacity shared memory, accessed by any thread from a block currently executing on an SM; small constant memory cached and optimized for broadcast, accessed by any thread; and texture memory, cached and optimized for 2D locality, accessed by any thread.

A typical flow of program execution with CUDA is the following: 1) host code allocates and initializes global memory on the device and calls kernel; 2) kernel loads data from the global memory to the shared memory, synchronizes, processes data through the algorithm, and writes results back to the global memory; and 3) host transfers results back to the host memory.

The global memory allows exchanging spikes between all neurons in all blocks. However, kernel re-launch is required for global synchronization. The shared memory provides storage for model variables and data structures during execution. Texture memory can be used for storing synaptic data structure.

Among the major challenges of CUDA programming are these issues: balancing limited resources (registers, shared memory) for maximum parallelism; aligning and coalescing global memory access operations for maximum throughput; avoiding access conflicts in shared memory operations; reducing and hiding global memory access and synchronization latency; and obtaining an execution configuration (collection of kernels with various grid and block sizes, allocation of computation) with the best performance.

III. Design and Implementation

Stewart and Bair [2] applied PS method to IZ-based hybrid system of SNN and demonstrated full double-precision accuracy. Hybrid system in general can be represented as a block diagram (Fig. 2). Specifically, the update iteration in Fig. 2 consists of several steps: First, running a PS integration step an arbitrary number of times until the error tolerances for all model variables reach their required limits. The number of runs defines the integration order. Second, testing for a spike. If a spike is detected, the computation proceeds with the Newton-Raphson (NR) root-search method, which is used to obtain the exact spike time. The NR computation is followed by a PS step, which is required for the model variables to update at the corrected spike time. The update iteration runs sequentially for all synaptic events arrived within the current integration step.

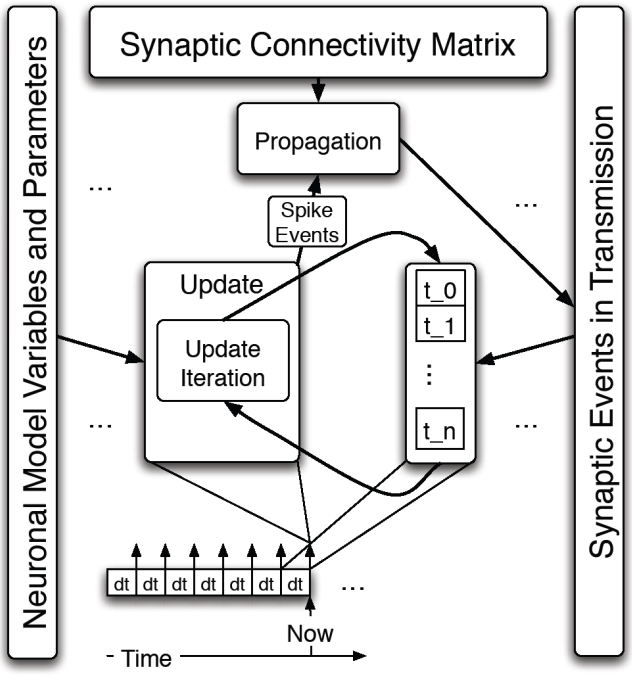

Fig. 2. Hybrid SNN system: simulation execution flow. An integration step *dt* consists of iterations through time-sorted synaptic events *t_0...t_n* for every neuron in the network. With every iteration the update of model variables takes place relative to the time of a processed synaptic event. The value of *dt* is small enough (0.25 ms) so that a neuron can produce at most one spike event during this period. In the propagation phase all spike events are distributed to *Synaptic Events in Transmission* data structure for future execution with appropriate weights and delays. The distribution is processed according to the matrix of synaptic connectivity.

Concluding iteration updates model variables at the end of a simulation step.

The arbitrary order and full event time scale introduce difficulties for GPU-based implementation, because of the variable computation size per neuron.

A. Software architecture

The implementation is based on three kernels: update, propagation, and sorting (Fig. 3). Computation is done in

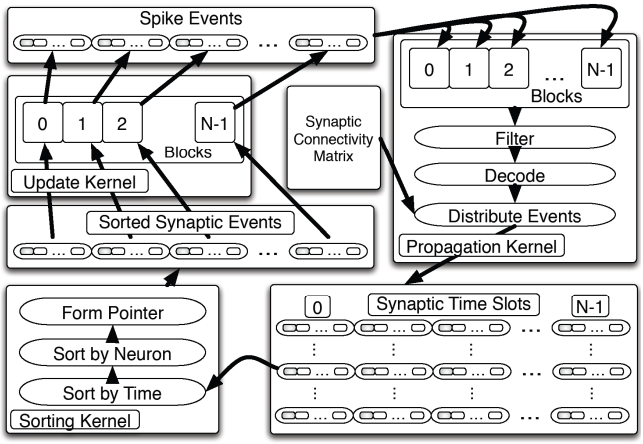

Fig. 3. Data Flow in GPU-based implementation of PS-based SNN with IZ neurons.

single-precision floating-point format.

The update kernel loads the sorted array of synaptic events from the global memory, performs numerical integration, and stores spike events to the global memory.

The propagation kernel loads the spike events produced by the update kernel. For each spike event it queries the synaptic connectivity matrix for the destination neurons, weights (GABAergic or AMPAergic synaptic conductance deltas) and delays. From these data it forms the synaptic events and stores them into the global memory based on their arriving time slot. The time slot for an event is determined by its delay in terms of *dt*, (i.e. in which *dt* relative to the current time this event is scheduled to arrive). Since delays are bounded by maximum and minimum values, the time slot data structure can be implemented as a circular buffer.

The sorting kernel loads all synaptic events from the current time slot, sorts them by the target neuron index and by the event time, forms a pointer-based compact array, and stores it to the global memory.

The execution configuration of each kernel is tuned based on the respective computation algorithm for the best performance. Communication abstraction between kernels allows flexible partitioning of work between grids with blocks of various sizes. Coalesced memory access is achieved.

B. Update phase

The update phase performs PS integration at *dt* time increments for each neuron in the network. Within *dt* time period, the update is performed at the time of synaptic events if there are any scheduled (Fig. 2). The computation in the update iteration is based on PS equations describing IZ-neuron with conductance-based synapses [2].

$$y(t+\Delta t) = y(t) + \sum_{p=0}^{n} y_p \times (\Delta t)^p, \qquad y = \{v, u, \eta, \gamma\}$$

$$v_1 = \left((\chi v)_0 + E_\eta \eta_0 + E_\gamma \gamma_0 - u_0 + I\right)\frac{1}{C}$$

$$(\chi v)_p = \sum_{j=0}^{p} \chi_j v_{p-j}$$

$$v_{p+1} = \left((\chi v)_p + E_\eta \eta_p + E_\gamma \gamma_p - u_p\right)\frac{1}{C(p+1)} \qquad (7)$$

$$u_{p+1} = a\left(b v_p - u_p\right)\frac{1}{(p+1)}$$

$$\eta_{p+1} = -\lambda_\eta \eta_p \frac{1}{(p+1)}$$

$$\gamma_{p+1} = -\lambda_\gamma \gamma_p \frac{1}{(p+1)}$$

$$\chi_{p+1} = k v_p - \eta_p - \gamma_p, \qquad \chi_0 = k v_0 - \eta_0 - \gamma_0 - k v_{thresh}$$

$$if\ v \geq v_{peak}: \begin{cases} v = c \\ u = u + d \end{cases}$$

Parameters in Eq. (7) are the same as in Eq. (6) with addition of these terms: η, excitatory conductance; γ, inhibitory conductance; Δt, time step; n, maximum PS order; p, current PS order; E_η, excitatory synaptic reversal potential; E_γ, inhibitory synaptic reversal potential; $(\chi v)_p$, second order term that requires a convolution; and λ_η and λ_γ, excitatory and inhibitory synaptic conductance decay rate constants respectively.

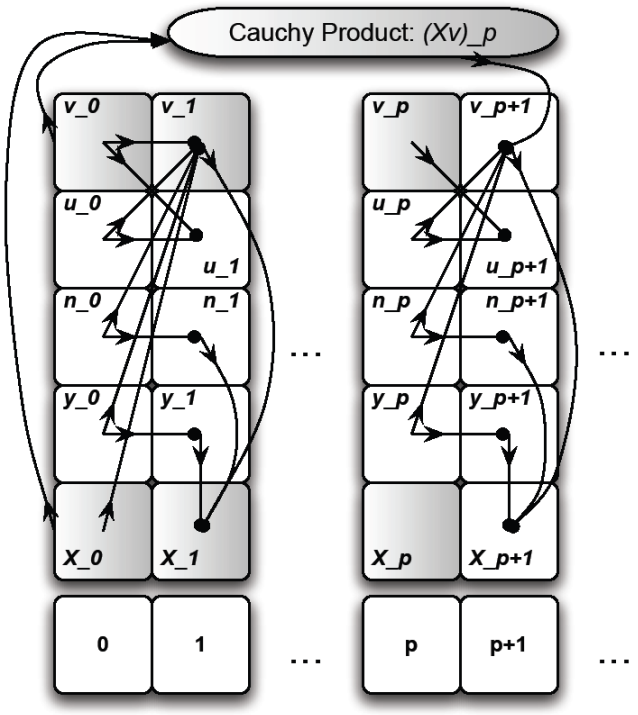

Fig. 4. Dependency graph of PS iterations in IZ-based SNN model. The update of model variables at any point in time is an iterative process. The iterations are suspended if the error tolerance limits on all variables are reached. This graph reflects the data dependencies of Eq. 7. Variables participating in Cauchy product computation are highlighted in grey.

Eq. (7) can be described as a dependency graph (Fig. 4). In Fig. 4 and Eq. (7), the PS step has both data and task parallel computation. A few attempts to exploit parallelism within the PS step on GPU resulted in more overhead than the benefit. As a result, in the most recent implementation the PS step is considered as a basic block of computation.

A neuron is mapped to a single thread during the integration step. The maximum number of neurons that an SM can handle is 64. The major limitation is the shared memory space, which is used for the PS step data structures, for storage of model variables and parameter, and for spike and the synaptic events. Another limitation is the register space, which limits the number of active threads to 256 per SM. As a result block partition range is 1-4 active blocks.

There are three major sources of branch divergence. First, the number of PS steps per PS update (Fig. 4) is not the same for all neurons. Second, the number of PS updates varies among neurons and depends on the number of synaptic events scheduled for each neuron (Fig. 2). Third, Newton-Raphson (NR) algorithm increases computation size for spiked neurons. In order to reduce branch divergence the

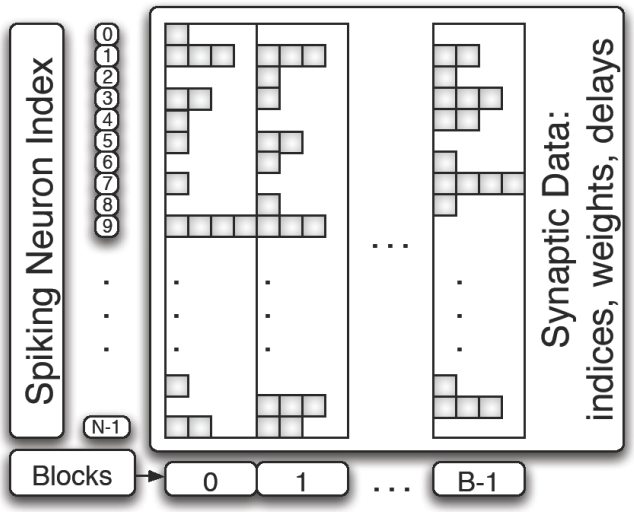

Fig. 5. Synaptic connectivity matrix. A vertical coordinate is a spiking neuron index. A horizontal coordinate represents all target synapses for the spiking neuron for a block of the update phase. If a spike event is produced, the entire target array is loaded by a warp via texture cache into the shared memory for distribution to the time slot buffer.

allocation of neurons to threads is done in sparse manner, (*i.e.* every 2^{nd} – 4^{th} thread performs the update computation).

C. Propagation phase

The propagation kernel distributes spike events to the time slots organized in the circular buffer array. Several stages are performed: filtering, synapse fetching, and time slot binning.

Filtering allows only spikes from the neurons with connections in the block of interest to participate in the further computation. It is done as a bitwise "AND" of incoming spikes with a block-specific filtering mask loaded from the texture memory.

A fetch of the synaptic data is done from the block-partitioned synaptic matrix stored as a 2D texture (Fig. 5).

The spiking behavior of neurons is suitable for both temporal and spatial cache locality. Indeed, spikes usually arrive as spike trains (temporal locality). Besides, there are usually several post-synaptic target neurons for a pre-synaptic spiking neuron (spatial locality). CUDA provides a cache with spatial 2D locality. The data access is the most efficient if the threads in a warp are accessing a coalesced segment in the texture memory. The synaptic data structure is based on ELLPACK format, Bell and Garland [14], and is optimized for uniform connectivity and warp-oriented access (Fig. 5). Although this approach may seem to increase the number of data loads during the synapse look-up, because the synaptic connectivity data of a source neuron is present in multiple blocks of the connectivity matrix, the data loads with temporal or special locality are generally well suited for caches, which are expected to grow in size in the next GPU generations.

The spike binning stage distributes synaptic events into bins or time slots according to their arrival times approximated to the nearest *dt*. Each event is a triplet: neuron number, time, and weight. Within a bin the exact time is preserved. The bins are emptied by the sorting kernel and reused again by the propagation phase in the circular manner. The number of bins is determined by the maximum and minimum synaptic delay in the network. Each block has its own set of bins. In the future implementation optimized for large event throughput, each neuron may have a set of bins, which would reduce sorting time.

D. Sorting phase

The sorting phase is required because the propagation kernel distributes synaptic events by appending them to the time slot arrays. As a result events are unsorted. The sorting phase can be a part of the update kernel. In this case the number of data accesses to the global memory is reduced. However, a separate kernel for sorting phase provides opportunity for an optimal execution configuration and computation decomposition, which results in better performance.

The task of the sorting kernel is to sort the synaptic events by time and by the neuron index, and to reduce the resulting sorted data structure to a pointer array. In this array, the pointer part provides a reference for each target neuron to a segment with sorted time-weight pairs. Thus, the update phase can sequentially iterate through these pairs for each neuron and perform the PS updates. Sorting by neuron index can be avoided in the future implementation if each neuron has its own set of time slots. Such implementation can be justified if the number of synaptic events per neuron per integration step is large enough to use available global memory bandwidth fully.

The sorting algorithm is based on the radix sort designed by Satish *et al*. [15], which is considered as the fastest GPU sort at the time of publication. The code for this algorithm is available as a part of CUDPP (CUDA Data Parallel Primitives) Library on the terms of BSD license [16].

The original radix sort implementation was modified in order to accommodate the specifics of this application. The kernel loads blocks of data into shared memory. Every block is sorted by an optimal algorithm based on the data size in the block (e.g., warp size, 128, 256 events). This implementation helps to reduce synchronization overhead. The formation of pointer array is done by the warp-scan algorithm, which is a part of CUDPP.

IV. Results and Analysis

A. Verification

Functionality of the implementation was verified with that of provided reference CPU code [2] for networks with all sizes and connectivity densities presented in this paper. The reference implementation was modified to accommodate variable delay time. All neuron parameters, weights, and delays are randomly initialized with values about original values. Connectivity is randomized, but kept at a specific percentage. Zero PS error tolerance was applied. Voltage traces for all neurons for the ten seconds of simulation have been tested for equality between the CPU and GPU versions.

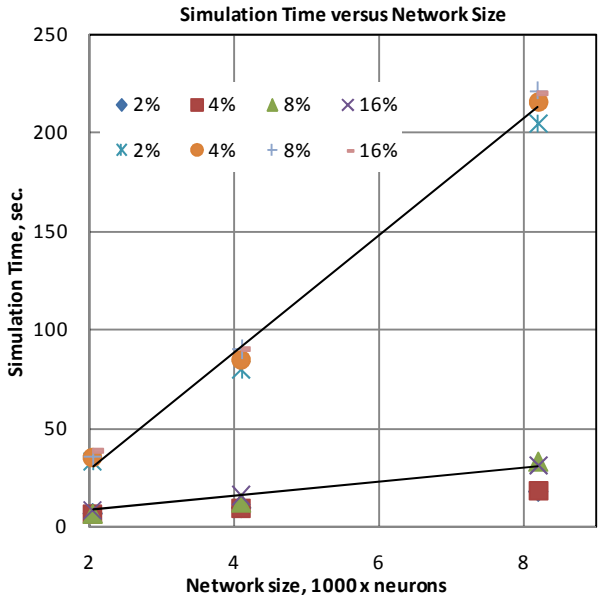

Fig. 6. Simulation time versus network size for a range of network densities (number of synapses per neuron represented as a percent of the total network size). Upper plot characterizes PC simulation and lower plot characterizes GPU simulations. Each data point on the plot is an average result of three 10-second simulations of network activity.

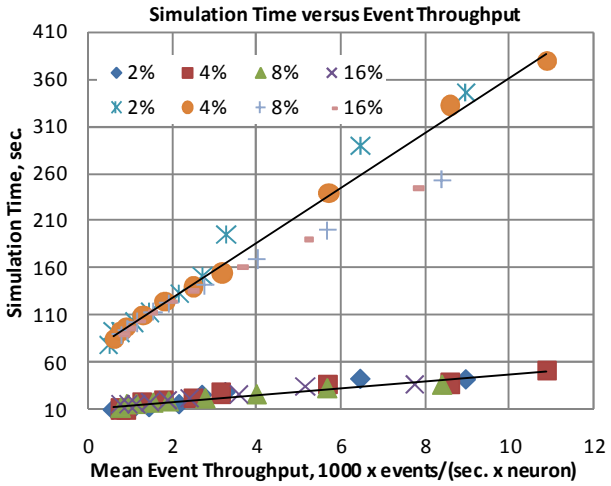

Fig. 7. Simulation time versus throughput per neuron for a range of network densities (number of synapses per neuron represented as a percent of the total network size). Upper plot characterizes PC simulation and lower plot characterizes GPU simulations. Each data point on the plot is an average result of three 10-second simulations of network activity.

Exact match was achieved in all cases.

CUDA supports IEEE-compliant basic mathematical floating-point operations in special instructions with explicit rounding modes. These instructions incur a higher computational time penalty and prevent the compiler from optimizing separate multiply and add into the multiply-add operation, which is faster but less accurate [17]. This implementation provides both options to the user: IEEE compliant version, (which uses instructions with round-to-nearest-even mode), and compiler-optimized version, (which provides 5–8 % reduction in execution time at the expense of accuracy).

B. Results

For profiling purposes the networks of various sizes and connectivity densities with randomized values of model parameter for all neurons have been simulated on a GTX260 device. This device has 24 SMs, has 16 KB shared memory per SM, has 938 MB global memory, and operates at a clock rate of 1.3 GHz. Two characterization tests have been performed. PS update step never diverged in both tests.

The first test characterizes how simulation time scales with the network size and the connectivity density. All tested networks are randomly connected with a constant ratio of 80% excitatory and 20% inhibitory synapses. All parameters are loaded into the device memory before starting simulation. The excitation is done by injecting current into each cell with a random magnitude in the 0–200 pA range during first 50 ms of simulation time. The time is measured using CUDA events [18] starting from the first kernel launch and ending with the last kernel execution. No data are transferred between the host and the device during the simulation. Zero error tolerance on model variables is applied in the PS integration step. The reference simulation is performed on a PC with Opteron 285, 2.6-GHz processor. The characterization plot of simulation time is depicted in Fig. 6.

As seen in the Fig. 6, the execution time has a near linear scaling factor for both devices. This linearity is expected, since hardware resources are fully used at a single block, and increasing the network size results in more blocks executed sequentially. However, due to the shared memory limitations, the networks with high connectivity density and larger network sizes cannot be simulated in the current implementation, which requires all the synaptic events to be loaded into shared memory before proceeding with the PS update. This limitation can be surmounted if synaptic events are fetched in batches.

GPU simulation is 8–9 times faster than PC simulation. Real time or faster performance is achieved for all networks with size of 2048 neurons and for 2%-connected (81 synapses per neuron) and 4%-connected (162 synapses per neuron) networks with 4096 neurons.

Previous works mentioned earlier achieve faster performance, larger network sizes, and larger connectivity density. This is expected because of several factors. First of all they use faster devices. Secondly, the complexity of computation is different since they use Euler-type method with Eq. (6) compared to PS method used in this work. Since they use synchronous simulation, the time of events is averaged to the nearest millisecond. Consequently, the change in synaptic conductance is done once per millisecond for all events. In this work, besides synchronous updates every 0.250 ms, the handling of an event is done exactly at the time when it arrives to the synaptic cleft. Thus, all events are processed sequentially per neuron, but every event is unique in time within the boundaries of single precision

floating-point accuracy. The previous works do not provide comparable accuracy.

The second test characterizes how the simulation time varies with the event throughput per neuron for the networks with various connectivity densities (Fig. 7). The mean throughput is generated by increasing the excitatory/inhibitory synaptic ratio from 0.8/0.2 to 0.98/0.02 for a network of 4096 neurons. As a result, more events are generated per second per neuron.

As seen in the Fig. 7, the execution time has near linear scaling factor for both devices. In this case, GPU simulation is 6–9 times faster than PC simulation. The system is capable of handling the throughput of unique events on the order of 10,000 events per second per neuron. The maximum detected PS integration order is 23. It is directly proportional to the event throughput.

V. Conclusion

Real-time performance for SNN with 4% connected 4096 IZ neurons was achieved in this implementation on the GTX260 GPU. The GPU simulation is 9 times faster than PC simulation based on Opteron 285, 2.6-GHz. The major part of the computation (about 80 %) is done by the update kernel.

Implemented as a hybrid type, the system has a potential for applications requiring high computational accuracy. This potential becomes prominent if the system is extended with an STDP algorithm. Very low quantization error in hybrid or event-driven systems makes these systems preferable if high accuracy is required.

Functionality of the system was verified with a reference implementation. Transient potential waveforms for the entire network were compared to those of reference program. An exact match between CPU and GPU results has been achieved.

There are many more potential improvements, which could be investigated: 1) parallel implementation of Newton-Raphson method or replacing it with another more parallel-friendly root-search algorithm; 2) optimization of synaptic connectivity matrix and its memory access; 3) using page-locked mapped memory and/or streams for interface and overlapping communication between a host and a device with computation on the device; 4) optimization of the block-level network allocation, and reduction of inter-block connections based on provided topology and heuristics; 5) extending to a double-precision floating-point format; 6) verification of system results with devices other than GTX260, enabling multi-GPU functionality; 7) extending the range of biological features (e.g., synaptic plasticity, long-term potentiation and depression, and STDP); 8) applying the system in robotics; 9) reducing effects of shared memory limitations and branch divergence; and 10) synchronizing every simulation step with the device clock, and implementing a real-time system.

VI. References

[1] R. Brette, et al., "Simulation of networks of spiking neurons: A review of tools and strategies," *Journal of Computational Neurscience*, vol. 23, no. 3, pp. 349-398, 2007.

[2] R. Stewart and W. Bair, "Spiking neural network simulation: numerical integration with the Parker-Sochacki method," *Journal of Computational Neuroscience*, vol. 27, no. 1, pp. 115-33, Aug. 2009.

[3] G. E. Parker and J. S. Sochacki, "Implementing the Picard iteration," *Neural, Parallel Sci. Comput.*, vol. 4, pp. 97--112, 1996.

[4] E. M. Izhikevich, "Simple model of spiking neurons," *Neural Networks, IEEE Transactions on*, vol. 14, pp. 1569--1572, 2003.

[5] J. M. Nageswaran, N. Dutt, Y. Wang, and T. Delbrueck, "Computing spike-based convolutions on GPUs," in *Intl. Symposium on Circuits And Systems*, 2009, pp. 1917-1920.

[6] J. Nageswaran, N. Dutt, J. Krichmar, A. Nicolau, and A. Veidenbaum, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," *Neural Networks*, Jul. 2009.

[7] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk, "NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs," *Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, vol. 0, pp. 137-144, 2009.

[8] J.-P. Tiesel and A. S. Maida, "Using parallel GPU architecture for simulation of planar I/F networks," in , 2009, pp. 754--759.

[9] E. Picard, *Traite D'Analyse*. Gauthier-Villars, 1922-1928, vol. 3.

[10] E. A. Coddington and M. Levinson, *Theory of Ordinary Differential Equations*. New York: McGraw-Hill, 1955.

[11] A. Hodgkin and A. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve.," *The Journal of physiology*, vol. 117, pp. 500--544, Aug. 1952.

[12] L. Lapicque, "Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarisation," *J Physiol Pathol Gen*, no. 9, pp. 620-635, 1907.

[13] E. M. Izhikevich and G. M. Edelman, "Large-scale model of mammalian thalamocortical systems," *Proceedings of the National Academy of Sciences*, vol. 105, pp. 3593-3598, 2008.

[14] N. Bell and M. Garland. (2008) Efficient Sparse Matrix-Vector Multiplication on CUDA. [Accessed online 04/30/2010]. http://mgarland.org

[15] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in , 2009, pp. 1--10.

[16] (2010, Apr.) CUDA Data Parallel Primitives Library. [Accessed online 04/30/2010]. http://code.google.com/p/cudpp/

[17] (2008) NVIDIA CUDA Programming Guide 2.3. [Accessed online 04/30/2010]. http://developer.nvidia.com

[18] (2009, Jul.) NVIDIA CUDA C Programming Best Practices Guide. [Accessed online 04/30/2010]. http://developer.nvidia.com