Scalable Multi-Precision Simulation of Spiking Neural Networks on GPU with OpenCL

Dmitri Yudanov, *Member, IEEE*, Leon Reznik, *Senior Member, IEEE*

***Abstract* — Biologically-realistic multi-precision spiking neural network (SNN) simulation is designed and implemented on a new GPU device *Radeon™ HD 7970* using OpenCL framework. The implementation aims to investigate the role of time precision in simulated SNNs. Simulation methods and GPU platforms are reviewed. Simulation model and process are presented and analyzed. The GPU model is capable of simulating a SNN with up to two million neurons. GPU and CPU results are directly verified and found to match exactly.**

***Index Terms*—spiking neural network simulation, high precision, GPU implementation, OpenCL.**

I. Introduction

Simulations of spiking neural networks on graphical processing units (GPUs) have been rapidly gaining popularity. SNNs provide a wide gamut of features for implementation of artificial intelligence, control systems, pattern/voice/video recognition techniques, data analysis and other applications. These features come from biological plausibility of SNNs. SNN simulation algorithms naturally suite for execution on GPUs due to their flexibility for parallel implementations. Besides, GPU vendors have recognized new market opportunities in the field of general compute in addition to traditional graphics domain. Consequently, GPU designs acquire features that provide efficient execution of general compute code.

Current implementations touch various aspects of SNN simulation: architectural and programming aspects, such as cache performance, branch divergence, instruction throughput, device occupancy [1], GPU acceleration [2], [3]; high-level API for developing cortical models [4]; parallelization of existing neural simulators [5]; biologically-realistic large-scale simulator for cortical process modeling with STDP and PyNN-like interface [6]; flexible neuron network simulation framework with semi-automated code generation [7]; real-time spiking neural network simulator NeMo for robotics, control, and gaming [8].

The selection of SNN implementation or algorithm and suitable GPU device is a complex problem due to a wide variety of choices. As a result, a tradeoff between various simulation aspects is expected. Time precision is one such aspect. As a rule, SNN algorithms and implementation types are selected for the GPU implementation to provide a larger network size, bigger connection count, faster algorithm execution time with simulation precision of the algorithm being left as a less significant aspect (see section III). However, time precision can be viewed as another dimension for information processing, which is naturally observed in biological neural networks and researched in neuroscience (see section II). What is the purpose of having a SNN with a million imprecise neurons if the target algorithm can be simulated more efficiently with a thousand highly precise neurons? What if a SNN requires high temporal precision only for some parts, and in general it could be efficiently simulated with lower precision? In which applications the high temporal precision is desired? This work takes a step towards answering these questions. It introduces a multi-precision SNN simulation for evaluation of precision-per-performance aspects of SNN simulation.

In our previous work we introduced a GPU implementation of SNN with real-time performance and high accuracy [9]. This work extends the previous implementation in several ways. First, we moved the implementation from CUDA to OpenCL, because it provides a greater potential for future development. Secondly, we are targeting a new GPU device *Radeon™ HD 7970* with the architecture code named *Tahiti* [10]. Radeon™ HD 7970 extends GPU features and benchmarks beyond those known before April 2012. The third aspect of the work is scalable software architecture of the simulation.

II. Time Precision in Biological Neural Networks

High temporal precision is present in biological neural networks. Auditory systems from peripheral and subcortical [11] to cortical levels use high temporal precision for information processing. Auditory cortex neurons encode complex sounds by using finely timed modulations of the firing rate that occur on the scale of a few milliseconds [12]. For complex, rapidly varying, random, and natural sounds the precision lower than a few milliseconds significantly reduces the encoded information [13]. High degree of tolerance and selectivity of neural responses in the auditory cortex (A1) alone with multidimensionality of auditory processing provide grounds for invariant representation of auditory objects and simultaneous sensitivity of neural responses to a large number of stimuli [14]. A classic example of highly precise auditory system is that of the barn owl, which is able to determine prey direction in the dark by measuring inter-aural time differences (ITDs) with an accuracy of 1-2 degrees. This is achieved by a process of binaural sound localization with temporal precision of a few microseconds [15].

Manuscript received April 2, 2012.

D. Yudanov is with Advanced Micro Devices (AMD), Austin, TX 78741 USA (e-mail: dxy7370@amd.com, Dmitri.Yudanov@amd.com).

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

In the olfactory system the odor can evoke local field potential (LFP) gamma frequency oscillations (30–70 Hz) in the mitral cell layer of the mammalian olfactory bulb. Mitral cells participate in synchronization of a signal from different odorant receptors and provide a mechanism for downstream cortical neurons to decode and integrate stimuli information. The bulb can exhibit rapid gamma synchronization in mitral cells of all types on a timescale of less than 5 ms. The synchrony is mainly due to recovery from inhibitory post-synaptic (IPS) potentials. The IPS current lag between synchronized cells occurs within a narrow window of 500 us and for some cells down to 50 us [16].

Cortical information processing relays on temporal precision of spikes. Various factors intrinsic to a neural circuit seem to jeopardize the time precision: stochastic nature of synaptic transmission and ion channel gating mechanisms, dendritic filtering, and background synaptic noise. However, it has been recognized in the past that rapid fluctuations in somatic voltage contribute to precise axonal action potential initiation [17]. The inputs coincidently activated at the synapses of basal dendrites within sub-millisecond window are mediated by active dendritic mechanism. This mechanism is based on initiating local dendritic spikes that amplify and sharpen the somatic potential resulting in precisely-timed action potentials with sub-millisecond temporal jitter over a wide range of activation intensities and background synaptic noise [18].

It is well known that spike time dependent plasticity (STDP) critically relies on the precise timing of spikes, takes pre- and post-synaptic spike timing and adjusts synaptic strength accordingly. In turn, there is evidence that the spike coincidence detection is aided by group oscillation and synchronization with millisecond precision [19]. Synchronization between and within cortical areas is a self-organized ubiquitous group-communication process that results in dynamic creation and dissolution of synchronized cell assemblies as a response to stimulus and attention. The precision of synchronization can exist in the millisecond range. Timing of individual spikes relative to the phase of group oscillation encodes information and facilitates its transmission and processing.

Temporal filtering is a mechanism of detection and selection of behaviorally-relevant stimulus features from sensory temporal spike patterns. This mechanism is based on synaptic plasticity, intrinsic excitability, synaptic and somatic temporal integration and other processes. The filter accuracy, its type and dynamic range are dependent on the ability of both central and peripheral neurons to support precisely timed spikes. For example, Mormyrid electric fish communicate by means of inter-pulse intervals (IPIs) between electric organ discharges (EODs). On the receiving side electroreceptors respond to each EOD with a single precisely timed spike. The signals are further filtered and presented to central neurons as IPI sequences of neighboring fish. Central neurons possess IPI selectivity and temporal filtering features based on temporal summation of excitation and GABAergic inhibition, which establishes high-pass and low-pass tuning behavior, respectively [20].

In SNN models the relative spike timing directly affects information capacity and pattern recognition capability. For example, the model of Sterne *et al.* [21] is capable of completing and correcting multiple spiking patterns simultaneously given only partial noisy versions. The information is decoded using relative timing of spikes. More patterns can be stored and recalled if precision parameters are tightened and even driven beyond their biologically plausible values to infinite bounds.

III. Simulation methods

Spiking neural network simulations can typically be comprised of several phases [22]: 1. Solving model equations for each neuron. 2. Detecting and registering spikes. 3. Transforming spikes into synaptic events. 4. Scheduling the events. 5. Delivering the events.

The solutions for majority of spiking neuron model equations cannot be expressed analytically in a closed form. Therefore, numerical integration techniques are normally used. These techniques are based on time discretization and computation of an approximate solution for the next discrete time point given the solution at a current time instant [23]. The solution accuracy and precision capabilities of simulated neural network depend on numerical integration technique and simulation type.

Several types of simulation methodologies have been developed. They can be classified in respect to time-event synchronization. Clock-driven (time-driven, synchronous) simulations evaluate model variables only at fixed time points (steps). As a rule, simulation accuracy improves if time interval between points gets smaller. However, computation complexity increases if time interval decreases. In event-driven (asynchronous) simulation types the equation update occurs at the exact time of synaptic and spike events. Consequently, computationally intense spike prediction methods are used for calculating firing times. However, the accuracy of the event time in these systems is not linked to a time grid, but is limited by computer number standards and hardware. The accuracy of a clock-driven simulation can approach that of asynchronous type if a step size has an upper bound of reciprocal of total spikes in the network per second. In this case each spike can be uniquely resolved in time, but not necessarily it will be resolved.

Thus, it would be useful to have an efficient simulation type that exploits activity of the system states and performs computation only if such activity is detected and only on the parts of the system exhibiting activity. Systems that attempt to combine advantages of event-driven and clock-driven systems are hybrid (embedded event-driven) systems. They refresh the model variables at fixed points in time and also process (embed) uniquely timed events in between the fixed points. This allows one to maintain causality and stability of simulation independent of event occurrence. Hybrid systems calculate spike times retrospectively using

interpolation or root-search techniques after a threshold crossing by membrane potential is detected [24]. Computational complexity of hybrid simulation depends on neuron count, network-wide synaptic event rate, spike rate, and minimum event delay. Hybrid system possesses greater accuracy at smaller computational cost compared to costs of time-driven and event-driven simulations [25].

For example, quantized state system (QSS) methods [26] provide an approximate solution by quantization of system state vector based on its piecewise evolution in respect to a change in state variables by some quanta. Steps in QSS are only produced when a state differs from quantized variable by quanta. As a result, the model variable update occurs only for the "active" neurons in each step. QSS was shown to be superior to synchronous systems based on Bulirsch–Stoer and (Runge–Kutta) RK integration methods [27]. However, QSS is only a 3rd order accurate algorithm and may not be suitable for some high-precision simulations.

Voltage-stepping scheme is based on discretization of the voltage state-space and variable activity-dependent time discretization for a network of integrate-and-fire (IF) neurons [28]. Discretization of the voltage state-space is achieved by using piecewise constant, linear or quadratic interpolation of the non-linear membrane potential differential equation. This approximation allows local linearization, which is suitable for local-in-time application of event-driven scheme. The results show superior performance of this method as compared to RK2-based time-driven simulation for irregularly spiked neurons driven by balanced excitation-inhibition event trains.

A numerical integration technique is another aspect of simulation method. Euler, RK are the examples of low-precision numerical integration methods typically used in time-driven or hybrid simulation types. Bulirsch–Stoer (BS) [29] or Parker-Sochacki (PS) [30] methods can provide higher-precision and are more suitable for event-driven or hybrid simulation systems targeting high accuracy. BS method is mostly suited for smooth functions, unlike the functions from neuron models with discontinuities. It is based on rational extrapolation on some interval spanned by sequences of integration sub-steps. The PS [30] technique is based on Maclaurin series applied to an initial value problem (IVP). The method was developed based on the Picard iteration under the assumption that the solution function is locally Lipschitz continuous in y and continuous in t, and therefore can be described with power series. Power series representation allows expressing the next coefficient with the derivative of previous coefficient. Consequently, the IVP can be described derivative-free in terms of power series, where larger size of series defines the precision. This is very useful for software implementation. However, depending on neuron model complexity the application of this method may be prohibitive due to transformation of power terms into discrete convolution terms.

IV. GPU Simulation platform

GPU simulation platform is a hardware-software solution for simulation, which consists of: GPU, CPU, PC board, OS, software framework. GPU is the main computational device. CPU normally schedules work for GPU through software framework and provides sequential computation resource. Software framework is the programming and execution environment.

Most popular GPU software frameworks are OpenCL [31] and CUDA [32]. The frameworks are similar, but OpenCL has a greater potential because it is designed to support APU, CPU, GPU, FPGA and possibly other compute devices. A typical OpenCL program (kernel) is executed in parallel by work items (WI), batched in wavefronts (WF). Each WF autonomously executes vector instructions for all its WIs. WFs are organized in a grid of work groups (WGs). WIs in a WG share some memory resources. A kernel orchestrates the computation by mapping it to WGs, WFs, and individual WIs. Computation can be split into several kernels executed sequentially or concurrently, in- or out-of-order as a dependency graph. This flexibility provides a basis for mapping parallel abstraction forms into neural simulator: neurons, synapses, concurrent tasks.

Recently introduced AMD GPU architecture *Radeon™ HD 7970* [10] is an efficient occupancy-bound device for general compute. The work on *Radeon™ HD 7970* is orchestrated by the Command Processors (CP). CP fetches command queues and schedules WF allocation to compute units (CUs). The allocation is done as a concurrent multi-task priority scheduling process with maximization of WF throughput and occupancy. The task priority levels range from real-time tasks to background tasks. User-defined task graphs are managed in hardware without involvement of CPU. Task interleaving is achieved by means of independent and concurrent WG dispatching and context-switching between the task queues. Such task priority timing management is well suited for SNN simulations with various timing priorities. It allows mixed implementations including real-time critical tasks for neural control together with long-latency processes such as STDP.

Besides scheduling, each CP has access to all memory resources (GDDR, caches, shared memory and the platform memory, which is accessed through tunneling over PCIe). CP uses this access for global hardware synchronization across command queues, other CPs, and other devices on the platform. These features, for example, can be used in AI control systems with global workspace architectures, in which separate neural processes working in parallel have to be coordinated to produce a single control loop [33].

A CU (Fig 1) consists of 7 execution units: four 16-wide single-instruction-multiple-data vector units (SIMDs or VUs), 1 scalar unit (SU), 1 branch unit (BU), 1 Local Data Store (LDS). Each VU has 64 KB of port conflict-free vector registers shared across the executed WFs. Vector ALU has 32-64 bit/cycle instruction bandwidth. It supports IEEE-compliant floating point arithmetic for both double and

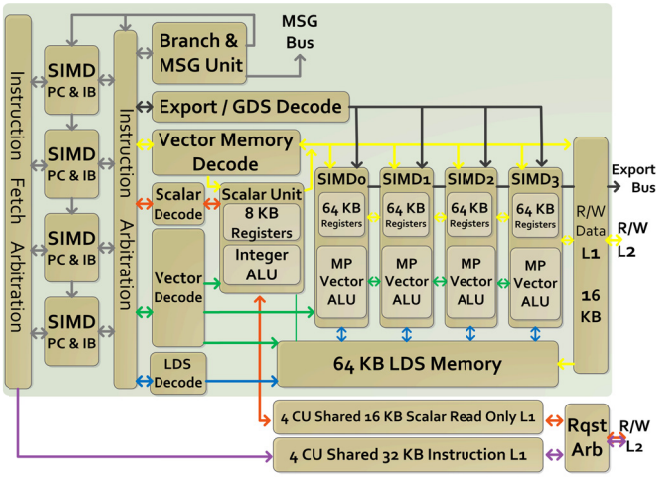

Fig. 1. Compute Unit Architecture. *Tahiti* GPU [10].

single precision data. A standalone BU pre-computes branches for SIMDs. The SU provides low latency and flexible low power flow control. It services 4 VUs executing one instruction per cycle, which is balanced with VUs executing autonomously one instruction for 4 WFs over 4 cycles.

The data is routed via cache hierarchy. Hardware-enhanced address coalescing and format conversion in L1 address sub-unit facilitate the routing. It allows high performance with flexible memory access patterns. L2 data cache is a right back R/W coherent across CUs cache. It can be efficiently used for inter-CU integer/float atomics, and data exchange via synchronization semantics. This relaxed memory model allows to control locality and coherency domains from the code: coherent code behavior across L2 and visible to all CUs, or local to CUs, and synchronized globally, or a mix of any combination of these.

LDS operates on vector and scalar registers, and its own memory space executing loads, stores, and atomics independent of SU and VUs. This allows to offload VU resources, increases performance, and reduces power. In addition, LDS provides bandwidth amplification for L1 cache and broadcast value delivery to vector units without extra latency.

WFs on CU are independent and can belong to different programs. They are executed autonomously and returned in any order. Each WI in a WF can follow a unique code path via predication. At least 10 WFs are executed in each SIMD. The actual occupancy is established by resource availability and algorithm demand.

This CU architecture has advantages over various parallel execution models. MIMD model: 4 vector units allow executing 4 unique instruction streams (WFs) in parallel. SIMD model: each vector unit executes its stream for 64 WIs. Simultaneous multithreading (SMT) model: each thread can execute unique instruction via predication. Temporal multithreading model: 40 WFs per CU are active and 4 are executed at a time.

In addition, the architecture supports virtual address space between CPU and GPU. GPU is able to access all system memory and handle page faults. It is ready to support x86 virtual memory. Exception support, function call support, and recursion provide a foundation for high level languages such as C++. Memory error detection and correction is supported throughout external GDDR and internal SRAM resources.

V. Design and Implementation

The implementation from our previous work [9] was transferred from CUDA to OpenCL with CU2CL tool [34] and re-designed to improve scalability, increase the range of synaptic connections and neurons in the network. Algorithms were orchestrated to use intrinsic WF synchronization whenever possible in order to reduce occurrence of expensive inter-WF barrier synchronization. Barrier synchronization was also minimized by replacing it with fast LDS atomics.

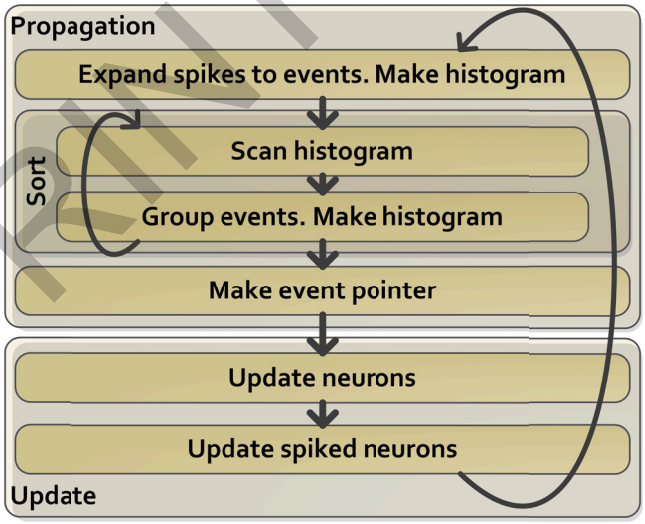

Fig. 2. Computation flow diagram.

The implementation consists of two major computation segments: update and propagation (Fig. 2). The update segment feeds synaptic events into SNN, solves neuron model equations for each neuron, and generates spike events. The propagation phase transforms produced spike events into synaptic events, sorts and delivers them to the update phase. The event exchange between kernels is done through DRAM on the GPU.

The update phase is a parallel implementation of hybrid simulation of SNN with Izhikevich (IZ) neurons [35]. The core of the solver is PS method with single-precision floating point standard. Stewart and Bair [24] demonstrated a great potential for PS method by developing a sequential implementation of the simulation with double-precision standard. The execution in the update phase proceeds as follows (Fig. 3). Every time step (0.25 ms) each of 64-wide WFs iterates a part of allocated to it SNN in the main loop. With every iteration a WI within a WF loads model parameters and variables, unique to each neuron, and a

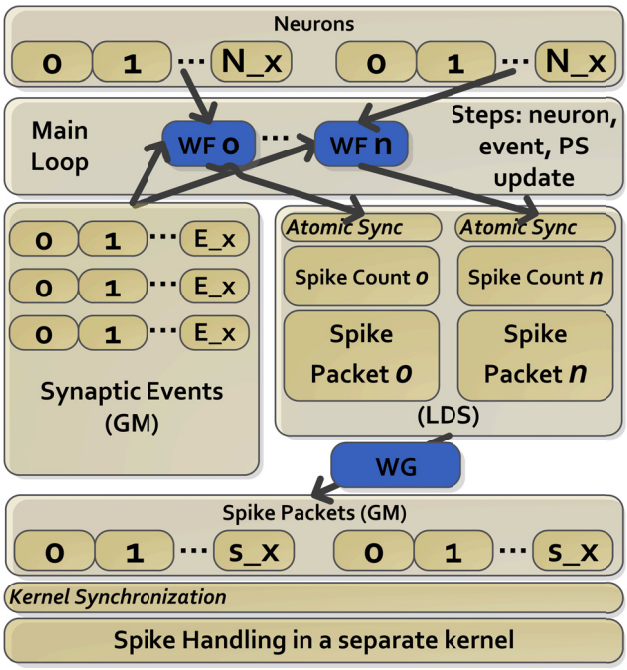

Fig. 3. Update kernel flow diagram.

pointer to its pre-sorted by time synaptic events that are due for execution in this step. After that the WI proceeds with numerical integration at every event time. It loads an event and performs an update of model variables. If no events are due for a neuron, the WI performs a single update at the end of a time step. The update consists of several PS steps. The step count is defined as PS integration order. The order depends on the tolerance for each model variable: integration proceeds until all tolerances are satisfied or the order hits the limit. The order varies depending on the value of continuity modulus (Lipschitz constant) at the local integration point: higher modulus value admits higher order. Thus, the computation increases if neuron model variables experience rapid change, for example, during a spike. This is similar to variable activity-dependent time discretization in the voltage state-space [28], where the step size decreases in proximity to the spike. However, in the former case the time is still quantized. The multi-precision capability of simulation is achieved by allocating tolerances on per-WF basis. With such scheme the branch divergence is minimized.

The order limit can be viewed as a precision control parameter. In the current implementation it is fixed for all neurons. Fixed order limit is more favorable for implementation on GPU: it helps to reduce branch divergence and provides computation in the register space, which is facilitated if the register allocation is defined at compile time. There is a rare possibility that for some neurons the limit is hit before the tolerance requirements are satisfied. The occurrence of these cases depends on the network spiking activity, and the value of the limit, which is restricted by the GPU register resources. Although the occurrence is extremely rare with the correctly tuned limit, in our future program revisions its handling will be covered, for example, by extending computation into local memory in order to meet tolerance requirements.

The penalty of adaptive order is the variability of program execution path across the WIs. Due to branch divergence WF execution path serialization is present in all current GPU architectures. Thus, unstructured, highly divergent flows have to be re-structured in software algorithmically if possible. For example, the branch divergence due to variability of events per neuron could be mitigated by pre-sorting neurons by event count. Because the neurons mapped to WIs are completely independent of each other and the next-in-queue kernels, the implementation can easily adapt the strategy of pre-sorting. However, this would incur an additional penalty. Instead of this tactic, the implementation of update kernel takes approach of a light, predication-oriented program with no barriers and fences, using only the intrinsic WF-based synchronization and fast atomic synchronization when a spike occurs. Potential improvement can be directed to the structure of the flow with the requirement of cooperative work between WIs. However, the overhead of this approach has to be carefully estimated.

Upon completion of PS step a WI tests its neuron for a spike (spike threshold crossing by membrane potential). In case of spike, the WI stores the state of spiked neurons for handling in a separate kernel and proceeds with its WF to work on the rest of neurons. Separation of spike handling is done to prevent large branch divergence. The spike occurrence is quite rare (normally 1-3% of the network size), and therefore, the kernel that handles spikes and the post-spike update incurs small execution time penalty. Also, per-WF neuron data allocations are small. The WF and WI mapping strategy to neuron data is the same as in the main update kernel with emphasis on independency and intrinsic synchronization. In the spike handling kernel a WI adjusts variables in such a way as to place the unknown spike time at zero crossing. After that it performs several NR steps to find the spike time. Although NR is very efficient in speed of function root search, its convergence is not guaranteed compared to the bisection method, for example [36]. In this implementation it was observed that NR diverges and the divergence rate (less than 1%) depends on the spiking activity. Currently the divergence is handled by defining the spike time at the half of the interval between the last two updates. The future work will address this issue and possibly explore other root-search techniques not limited to functional iterations, but possibly including matrix methods, Eigen-Solving and others [37]. Once the spike time is known the WI performs an update at the time of spike. It then inserts the spiked neuron ID and spike time in a spike packet data structure (one per WF) and continues iterating over the rest of the events. Similarly to the previous kernel, the last update is at the end of time step. This update is required for event causality. Upon completion all neuron variables are updated and spike data structure is formed.

The next computational segment is propagation phase. It consists of several parts. The first part (expansion)

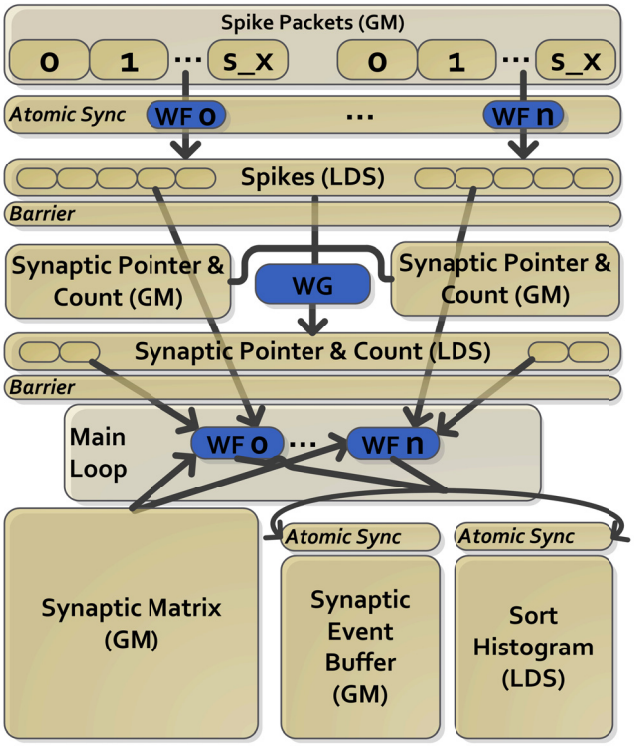

Fig. 4. Expand kernel flow diagram.

transforms spike packets into synaptic events (Fig. 4). In the expansion kernel each WF reads spike counts for a batch of allocated to it x spike packets from a standalone buffer. Since a spike packet can only be partly filled with spikes, this allows to load just the relevant part of it. A spike is a datum consisting of spiked neuron ID and spike time. Each WF brings spikes into LDS while synchronizing with other WFs through local atomics. Shared spike data across the WFs in a WG allows uniform workload distribution. Sort histogram is also loaded into LDS (not shown on the figure). The histogram keeps track of the event time distribution according to the value of its four least significant bits (0 - 16) for each time slot and each WG. This histogram is used in the sorting stage. Synaptic pointer and synapse count for each spike are also pre-loaded to reduce the load/store pressure in the main execution loop. Pre-loading scheme assumes that all loaded buffers are small. This affects scaling. In the future work the scalability of this scheme will be addressed by partial loading in the main loop with interleaving or inter-WF task parallelism.

After all required buffers are loaded the main execution loop starts: each WF picks a spike and corresponding synaptic pointer data from LDS. Then WIs of each WF retrieve the target synaptic data for spiked neuron from synaptic matrix, calculate event times and their histogram bin IDs according to the four LSBs. After that WIs insert the synaptic events into synaptic event buffer and increment histogram bin value. Direct global memory accesses to synaptic matrix and event buffer is enhanced by L1/L2 read-write capability of the device and data structure spatial locality. The synchronization between WIs is achieved by means of local atomics. The main loop is executed at the WF granularity in deterministic manner. Potentially this may affect workload balance if synaptic size variability is large. The future work will address this by implementing atomic-based workload distribution. WF granularity may also perform suboptimal at the event boundaries. For large synaptic ratios it may not be an issue, but it becomes more prominent for the small synapse counts. This can be addressed by fine-grain synaptic pointer mapping (at the expense of extra resource/computation penalty).

The synaptic event queue is implemented in a classic way [38] as a per-WG circular buffer with the size of largest synaptic delay in the network measured in time steps, S. This buffer consists of S segments, each with the size of maximum count of detected events in the simulation plus some margin. A WI simply adds delay time to the spike time, extracts the integer part of this sum, which determines the segment ID for placement of this event in the synaptic queue relative to the current queue pointer. The WI inserts the event into that segment in the queue. The event actually consists of three parts and there is a queue for each part: target neuron ID, event time, and synaptic weight. There is a separate queue for sort histogram (much smaller in size than the event queue). The queue pointer circles around the buffer directing to the segment of events, which is due for delivery in the current step. S limit guarantees that the events are taken from the last relative to the pointer segment of the queue and delivered to their targets every step before populating this segment with new events.

From the execution latency perspective the circular buffer implementation is efficient because it expands spikes and distributes events right away. However, it is not ideal if memory efficiency is a concern, especially for network size maximization. The queue is sparse and it uses maximum possible size in order to consider the worst case occupancy and avoid buffer overflow. There are sequential implementations of more memory-efficient synaptic event buffering schemes with just-in-time spike decode and event delivery [22], which may be considered for parallelization in the future work. Compression is another way to explore more efficient memory utilization. For example, the queue size can be reduced by 1/3 if pointers to synaptic weights and neuron IDs are stored instead of events. However, it is not without extra penalty.

Before delivery the events have to be sorted by event time and by neuron ID. This ensures spatial (network space) and temporal (event time) event ordering. The implementation of sort is a modified version of GPU radix sort developed by Harada and Howes [39] based on Merrill *et al.* [40]. The sort is an LSD-directed with 4-bit radix mask. The original implementation consists of 3 steps: 1) Generating count table (histogram) of bit-masked keys on per-WG basis. 2) Scanning WG count tables and generating global offset table. 3) Reordering keys in global memory according to the offset table. Various optimizations were applied in each step, which makes this implementation quite efficient: the sorting

power is more than 500 Mega Keys/s achieved on *Radeon™ HD 6970*. Modifications applied to this sort were directed by the requirements from update kernel. Values were added because it is required to sort synaptic events consisting of neuron ID, event time and weight. The values are the pointers to the relevant elements in the unsorted data structure. The key is replaced twice: with event time and with neuron ID. Step 1 and 3 were integrated into a single step to save on key load, parse and count. This became possible because in step 3 the keys are loaded for reorder, and their mapping to the WG grid in the next iteration is deterministic. The last reorder step moves unsorted events from queue segment to a new buffer according to the sorted keys. This step results in a random global memory access, which is suboptimal. However, such implementation is still more efficient as compared to the alternative, in which 2 values would have to be carried during the sort. Besides, L1/L2 cache facilitates transaction. Potentially this could be done in the update stage as well. Sorting by neuron ID can potentially be eliminated with per-neuron circular buffer at the expense of memory size increase. The last step of propagation phase is to form a pointer for each neuron to its events. This is done in a separate kernel by means of parallel prefix sum on the event counts per neuron.

VI. Implementation Analysis

A. Verification

Complete verification of all model variables and event times was performed between GPU device and CPU host simulation at every step of simulation. Various network sizes, precision parameter values, and synaptic connection sizes were used to exercise the implementation. GPU and CPU sets of neuron variables match exactly for the entire duration of simulations.

B. Results

The simulation is able to scale with SNN size according to Table 1. SNN statistics are reflected in Table 2. The largest speedup 88 is achieved with a small-size SNN and 11.5K synapses because of efficient GPU sort algorithm compared to insertion sort in the host implementation. These results are for partly optimized implementation. The optimization and further development is a work in progress.

A requirement for a comparable time-driven simulation is a unique resolution of each event in time. This can be approached if a step size has a minimum upper bound, which is no more than the reciprocal of total spikes in the network per second. On average about 200-400K events per step (0.8-1.6 billion per simulated second) were generated, which corresponds to a nanosecond time step or less. Most of recently reported time-driven simulations have millisecond-long steps. This implies 1 million times longer simulation execution time for comparable precision in a time-driven simulation.

Network Size (neurons)	Average Synapses per Neuron	GPU Time per Step, (ms)	CPU Time per Step, (ms)	Speedup Factor
2^{21}	90	13.5	659	48
2^{17}	1458	5.7	279	48
2^{14}	11677	3.2	283	88

Table 1. Simulation results: size-connection scalability in multi-precision networks with per-WF precision allocation. 1000 iterations. 250 us step.

Performance-per-precision can be estimated by scalability of execution time in respect to variable tolerances. The time / speedup gain: (5.7s – 4.4s) / (48 - 58) for highest to lowest precision for SNN case with 2^17 neurons. *Radeon™ HD 7970* handles work allocation well in this complex implementation regardless of precision requirement.

The simulation requires all memory buffers to reside on the device in order to avoid PCIe data transfer latency during the simulation. In all presented results the device global memory and maximum buffer allocation size was the bounding box and was nearly full. 6-core AMD *Phenom™ II*, 3.2 GHz CPU was used in the platform.

Network Size	Average Events per Iteration	Average Spikes per Iteration	Total synapse count
2^{21}	227,152	2,522	190,190,561
2^{17}	373,174	257	191,209,680
2^{14}	296,469	25	191,323,616

Table 2. Simulation results: SNN statistics. Only excitatory connections were used in order to generate high event traffic.

VII. Conclusion

The paper presents a multi-precision scalable software implementation for SNNs simulation with Izhikevich neurons on GPU. The implementation is designed with OpenCL language and designated for use on any OpenCL-supporting GPU. The implementation is functional and is verified with original sequential version: model variables and spike times for all neurons in a network in both CPU and GPU simulations match exactly for the entire duration. Although the algorithm is irregular and exhibits high rate of branch divergence, its GPU implementation (only partly optimized) still delivers worthy results due to GPU *Radeon™ HD 7970* architecture advantages. The implementation is scalable in three dimensions: neurons, connections, and precision.

The major scalability limiting factor is the size of global event buffer (3 GB) and PCIe bottleneck. With recent introduction of device named APU (Acceleration Processing Unit) [41] the standard GPU simulation platform may soon change. APU combines CPU and GPU on the same chip and provides efficient memory management, CPU-GPU data exchange, parallel and sequential task allocation, and access to the entire platform memory directly through DDR interface. Thus, it is possible to overcome 3 GB limit with APU device.

Besides exploiting APU for simulation, the future work will include the following: optimization, reduction of sort iterations by increasing radix width, splitting the execution graph into several paths to avoid kernel serialization, STDP features, just-in-time spike delivery, a library interface, and application examples for analysis of time precision role in

SNNs. For the most recent state of the project, contributions and the source code the reader is directed to https://code.google.com/p/neurosim/.

VIII. Acknowledgements

We thank Dr. Wu-Chun Feng for providing help with CU2CL tool [34]. We thank Lee Howes for initial assessment of performance.

IX. References

[1] V K Pallipuram, M Bhuiyan, and M C Smith, "A comparative study of GPU programming models and architectures using neural networks," *The Journal of Supercomputing*, pp. 1-46, May 2011.

[2] J Hoffmann, K El-Laithy, F Güttler, and M Bogdan, "Simulating biological-inspired spiking neural networks with OpenCL," in *Artificial Neural Networks – ICANN 2010*, Konstantinos Diamantaras, Ed.: Springer, 2010, vol. 6352/2010, pp. 184-187.

[3] C M Thibeault and R V Hoang, "A Novel Multi-GPU Neural Simulator," in *BICoB*, 2011, pp. 146-151.

[4] T Poggio, U Knoblich, and J Mutch, "CNS: a GPU-based framework for simulating cortically-organized networks," Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory Technical Report 2010.

[5] S H Lee and K Skadron, "Highly Parallel Implementation of NeuroJet using GPUs," School of Engineering and Applied Science-University of Virginia, A Technical Report in STS 4020 2010.

[6] M Richert, J M Nageswaran, N Dutt, and J L Krichmar, "An Efficient Simulation Environment for Modeling Large-Scale Cortical Processing," *Frontiers in Neuroinformatics*, vol. 5, 2011.

[7] T Nowotny, "Flexible neuronal network simulation framework using code generation for NVidia® CUDA™," *BMC Neuroscience*, vol. 12, p. P239, July 2011.

[8] Z Fountas, D Gamez, and A K Fidjeland, "A neuronal global workspace for human-like control of a computer game character," in *2011 IEEE Conference on Computational Intelligence and Games (CIG)*, Seoul, 2011, pp. 350-357.

[9] D Yudanov, M Shaaban, R Melton, and L Reznik, "GPU-based simulation of spiking neural networks with real-time performance & high accuracy," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, Barcelona, 2010, pp. 1-8.

[10] M Houston and M Mantor. (2011, June) Fusion Developer Summit: AMD Graphics Core Next. [Online]. http://developer.amd.com/afds

[11] N A Lesica and B Grothe, "Dynamic spectrotemporal feature selectivity in the auditory midbrain," *The Journal of Neuroscience*, vol. 28, p. 5412, May 2008.

[12] R Azouz and C M Gray, "Cellular mechanisms contributing to response variability of cortical neurons in vivo," *The Journal of Neuroscience*, vol. 19, pp. 2209-2223, March 1999.

[13] C Kayser, N K Logothetis, and S Panzeri, "Millisecond encoding precision of auditory cortex neurons," *Proceedings of the National Academy of Sciences*, vol. 107, pp. 16976-16981, September 2010.

[14] T O Sharpee, C A Atencio, and C E Schreiner, "Hierarchical representations in the auditory cortex," *Current Opinion in Neurobiology*, vol. 21, no. 5, pp. 761-767, October 2011.

[15] M Konishi, "Listening with two ears," *Scientific American*, vol. 268, pp. 66-73, 1993.

[16] N E Schoppa, "Synchronization of olfactory bulb mitral cells by precisely timed inhibitory inputs," *Neuron*, vol. 49, no. 2, pp. 271-283, January 2006.

[17] Z F Mainen and T J Sejnowski, "Reliability of spike timing in neocortical neurons," *Science*, vol. 268, pp. 1503-1506, June 1995.

[18] G Ariav, A Polsky, and J Schiller, "Submillisecond precision of the input-output transformation function mediated by fast sodium dendritic spikes in basal dendrites of CA1 pyramidal neurons," *The Journal of neuroscience*, vol. 23, p. 7750, August 2003.

[19] L M Chalupa, N Berardi, M Caleo, L Galli-Resta, and T Pizzorusso, *Cerebral Plasticity: New Perspectives*.: MIT Press, 2011.

[20] A A George, A M Lyons-Warren, X Ma, and B A Carlson, "A Diversity of Synaptic Filters Are Created by Temporal Summation of Excitation and Inhibition," *The Journal of Neuroscience*, vol. 31, pp. 14721-14734, October 2011.

[21] P Sterne, D MacKay, and S Wills. (2012) The Inference Group. Cavendish Laboratory, Cambridge. [Online]. www.inference.phy.cam.ac.uk/pjs67/sterne_2012_information_recall.pdf

[22] R D Stewart and K N Gurney, "Spiking neural network simulation: memory-optimal synaptic event scheduling," *Journal of Computational Neuroscience*, vol. 30, pp. 1-8, 2011.

[23] E Hairer, S P Norsett, and G Wanner, *Solving Ordinary Differential Equations I and II*.: Springer Verlag, 2010.

[24] 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. 115-133, August 2009.

[25] A Hanuschkin, S Kunkel, M Helias, A Morrison, and M Diesmann, "A general and efficient method for incorporating precise spike times in globally time-driven simulations," *Frontiers in Neuroinformatics*, vol. 4, October 2010.

[26] E Kofman and S Junco, "Quantized-state systems: a DEVS Approach for continuous system simulation," *Transactions of the Society for Computer Simulation International*, vol. 18, no. 3, pp. 123-132, September 2001.

[27] G L Grinblat, H Ahumada, and E Kofman, "Quantized state simulation of spiking neural networks," *SIMULATION*, vol. 88, no. 3, pp. 299-313, March 2011.

[28] G Zheng, A Tonnelier, and D Martinez, "Voltage-stepping schemes for the simulation of spiking neural networks," *Journal of Computational Neuroscience*, vol. 26, no. 3, pp. 409-423, 2009.

[29] R Bulirsch and J Stoer, "Numerical treatment of ordinary differential equations by extrapolation methods," *Numerische Mathematik*, vol. 8, no. 1, pp. 1-13, 1966.

[30] G E Parker and J S Sochacki, "Implementing the Picard iteration," *Neural, Parallel & Scientific Computations*, vol. 4, pp. 97-112, March 1996.

[31] B Gaster, D R Kaeli, L Howes, and P Mistry, *Heterogeneous Computing with OpenCL*.: Morgan Kaufmann Pub, 2011.

[32] J Sanders and E Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*.: Addison-Wesley Professional, 2010.

[33] R Arrabales, A Ledezma, and A Sanchis, "Towards conscious-like behavior in computer game characters," , 2009, pp. 217-224.

[34] G Martinez, M Gardner, and W Feng, "CU2CL: A CUDA-to-OpenCL Translator for Multi-and Many-Core Architectures," phd thesis 2011.

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

[36] A Jeffrey and H H Dai, *Handbook of mathematical formulas and integrals*.: Academic Press, 2008.

[37] *Algorithms and theory of computation handbook: general concepts and techniques*.: Chapman & Hall/CRC, 2010.

[38] 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.

[39] T Harada and L Howes. (2011, Dec.) Heterogeneous Compute. [Online]. http://www.heterogeneouscompute.org/wordpress/wp-content/uploads/2011/06/RadixSort.pdf

[40] D G Merrill and A S Grimshaw, "Revisiting sorting for GPGPU stream architectures," in *Proceedings of the 19th international conference on Parallel narchitectures and compilation techniques*, 2010, pp. 545-546.

[41] M Daga, A M Aji, and W Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing," in *Symposium on Application Accelerators in High-Performance Computing*, 2011, pp. 141-149.