Implementation of configurable and multipurpose spiking neural networks on GPUs

Antonio Arista-Jalife and Roberto A. Vázquez

Abstract— **In recent years, scientists and researchers have paid special attention to the implementation of Spiking Neural Networks (SNN), for approaching simulations of the human brain mechanisms, or to solve practical problems, such as epilepsy and seizure detection [1]. Nevertheless, large-scale SNN simulations are expensive from the computational point of view. These models quite often require the usage of parallel computing implementations on several devices[2], such as FPGAs [3], supercomputers [4], and recently on GPUs. Particularly on these last class of devices, a variety of techniques have been posed, like OpenGL pixel shade rendering [5], as well OpenCL and CUDA C/C++ [1], [6]. In this paper, we introduce a configurable large-scale fully parallel implementation that could well model a wide range of SNN using NVIDIA's GPU Common unified device architecture. Because of the achieved scalability, and flexibility in the definition of synaptic connections and neuron population, we show that our implementation shall be easily adapted as a back-end system for a wide variety of purposes. To examine some advantages of this implementation, we pose a comparative between Izhikevich's serialized program and our implementation using a large amount of neurons and synaptic connections. Also, as an additional experimental data, we report execution times, generated spikes and firing rates in a huge amount of neurons and a comparative between different neuron-synapse relations. We also achieved the simulation of 3.8×10^6 synaptic connections in real time (which is 1,000 time steps, one millisecond each, executed in one second). Finally, we demonstrated the increased performance on different hardware environments.**

I. Introduction

THE third generation of artificial neural networks (ANN) is the so-called Spiking Neural Networks (SNN) which uses spike trains as the mechanism to process and propagate information among other neurons. These spike trains allow researchers and scientists in neuroscience to model biologically realistic processes of the human brain [7], [8], [9]. Also this characteristic provides practical solutions and applications such as pattern recognitions using a single spiking neuron [10], [11], [12], audiovisual information processing [13], [14], olfactory information processing [15], string patterns recognition [16], and associative memory [17].

Some of the most popular spiking neural models used are the Hodgkin-Huxley, integrate-and-fire and the Izhikevich model, being the latter one the most computationally plausible at the moment because of the rich firing patterns that produces and its low cost in terms of execution time.

Antonio Arista-Jalife and Roberto A. Vázquez are with the Intelligent Systems Group, Universidad La Salle, Mexico City, Mexico (email: arista.antonio@gmail.com, ravem@lasallistas.org.mx).

The authors thank to Universidad La Salle for the economical support under grant number ULSA I-054/11 and to Marco P. Ramírez Tachiquin for letting us use the GPU hardware acquired with the CONACyT project 106722. Antonio Arista-Jalife thanks Roberto A. Vázquez and Marco P. Ramírez Tachiquin for their support and guidance.

The nature of the mentioned SNN models is completely parallel, which means that a serialized procedure to model neurons in large-scale often requires a huge amount of computational cost. This constraint has forced scientists and researchers to look for other paradigms in the implementation of a SNN. In [3] the author defines the usage of FPGAs in order to construct a hardware-based integrate-and-fire model. In [5] is also expressed the usage of OpenGL pixel shaders as data output to manipulate graphics processors units (GPUs). Also, in [4] is described the usage of supercomputer platforms to model large-scale SNNs.

Recently, the usage of multi-core GPU technologies such as OpenCL and NVIDIA Common Unified Device Architecture (CUDA), has increased its popularity, which makes a large-scale implementation plausible, without losing the desired parallelism and reducing drastically the computational and economical cost of a large-scale SNN model.

The implementation of SNN using NVIDIA's CUDA architecture provides a wide range of advantages such as the reduced computing time by using the arithmetic logic units of a GPU to parallelize and accelerate a SNN implementation, the inherent scalability of created programs, which can be executed in different GPUs, and the usage of the standard C programming language to facilitate general purpose computing. These combined advantages lead to an independent, fully parallel, high-speed SNN implementation, which can be combined with other processes such as front-end systems and/or database management systems.

Several implementations of SNNs on GPUs have been proposed; nevertheless, some of these implementations are constrained to the employment of pixel shaders techniques such as [18] and even pixel shaders interpretation is required [19]. Others are bounded to a single SNN model ([20] and [21] uses only the Izhikevich's model with STDP delay, [19] uses only the integrate-and-fire model, etc.). Beside, in almost every proposed implementation little or no configuration capacity in terms of neuron population, neuron-synapse relation, or neural model has been proposed.

Finally, it is important to remark that in some of these implementations (such as [21] and [20]) a specific GPU and hardware is required to realize the experiments. These limitations obstruct the fulfilment of some necessities: A SNN implementation that can adapt to several environments is required to freely utilize any GPU or hardware scenario. Also, the capacity to change the employed neural model, neuron-synapse relation, employed time steps and neuron population is required to generate different simulations because it has been demonstrated that some neural models are able to solve specific problems more efficiently than others [10], [11], [12]. And finally, the generated results must not require a pixel shading manipulation, including if desired an interpretation, to facilitate the merging of different systems and technologies with the SNN implementation.

In this paper, we present a multipurpose configurable large-scale SNN implementation written in CUDA C language, which can be easily modified in terms of synaptic connections, neuron population, neural model and number of execution time steps. Also, due to its nature, our implementation could be attached with several other programs, processes or even different hardware environments with small or none source code modification. This can lead to be used as a back-end system for multiple experiments.

To test the performance of our proposal several experiments were employed. We exert three evaluation metrics: the utilized execution time on the simulation, the firing rates and the total amount of neural connections per time step, which is defined as the multiplication of neurons and synapses per neuron.

The first experiment consists into a comparative among an adaptation of the Izhikevich's model in C language, the Izhikevich's original implementation and our proposed implementation in terms of execution times.

As a second experiment, we realized performance tests in huge simulations using more than 200,000 neurons and 1,000 synapses each to verify if the proposed SNN implementation can be simulated in a low execution time, additionally we include the number of spikes generated in every experiment and their respective firing rate.

In a third experiment, we posed a series of comparatives between pairs of neural networks with different neuron-synapse relations to analyse which neural network can be simulated faster.

As a fourth experiment, we tested several topologies that can be modelled in real-time. We remark that a real-time model consists in 1000 time steps (a millisecond each) simulated in a second of execution time.

And finally, we tested the versatility of our implementation on different hardware scenarios. In these experiments, we present a comparative among the original Izhikevich's implementation, the serialized C implementation of the Izhikevich model and our proposed implementation using different hardware platforms.

In section II, we describe the basic concepts needed to fully understand our implementation, whereas in section III we introduce our multipurpose configurable large-scale SNN implementation, a description of its functionality and the adopted parallel methods that increases the execution speed. Section IV presents the experimental results for all the proposed comparatives and experiments. Section V presents some conclusions and future work and in section VI some considerations in the implementation of our proposal are described.

II. Background

A. Spiking neural networks

A neural network is a massively parallel, distributed processor made of simple units called neurons, which have the natural capacity of storing their experimental knowledge and make it usable. This peculiar characteristic gives a neural network not only the ability to learn, but also generalize: a neural network can deliver accurate results without being trained specifically to receive a determinate input [22].

The third generation of the artificial neural networks (ANN), is called spiking neural networks (SNN), which introduced a new concept called "action potential" or "spike": an abrupt and temporary change of the membrane's action potential that propagates to other neurons and can occur at regular or irregular intervals [8], [9]. These spikes are generated when a determinate threshold is reached by the membrane potential, and after that, the membrane potential recovers a known value.

Spike trains can be considered as the elementary units of information interchange among the neurons. The number of spikes (their firing rate), and also their timings, are the information carriers, because there is no difference between spikes. In other words, the action potential by itself does not carry any information at all [9]. Particularly, one of the most popular SNN models is the Izhikevich's model (see [7]) which attains the biological plausibility of the Hodgkin-Huxley model and the computational efficiency of the integrate-and-fire model. The two-dimensional system of ordinary differential equations used, is defined as:

$$\begin{array}{ll} v' = 0.04v^2 + 5v + 140 - u + I & \text{if} v \geq v_{peak} \text{then} \\ u' = a\left\{b\left(v\right) - u\right\} & v \leftarrow c, u \leftarrow u + d \end{array} \tag{1}$$

where I is the neuron's input current (directly injected into it), $'$ is the d/dt, v is a dimensionless variable that represents the membrane potential of the neuron, u represents a membrane recovery variable, a represents the time scale of the recovery variable u, b represents the sensitivity of the recovery variable u, c represents the after-spike reset value of v, d represents the after-spike reset of the recovery variable u and v_{peak} is the threshold for the neuron's spike (firing).

B. NVIDIA's Common Unified Device Architecture.

Due to the increased market demand on high-definition graphics, graphics processor units (GPUs) have become into multi-core, highly parallel and multi-threaded processors [23]. Although every arithmetic logic unit can be treated as a separate neuron, some years from now the only method to create neural network models on GPUs was the employment of programmable arithmetic units called pixel shaders to combine inputs and generate an output color [24], [18], [19].

To facilitate the programming and pixel shaders' interpretation tasks, NVIDIA posed the CUDA architecture along with some modifications to the standard C language, and a compiler for the newly created language called CUDA C. These architecture and features were developed to provide

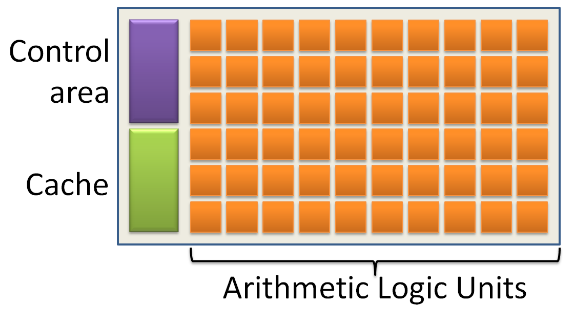

Fig. 1. Simplified representation of a NVIDIA CUDA GPU

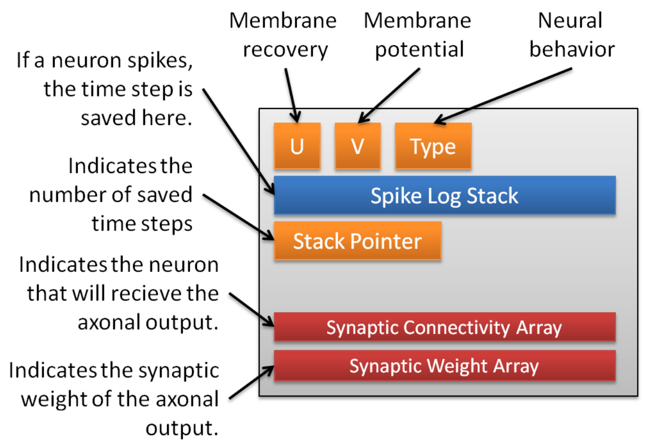
Membrane recovery
Membrane potential
Neural behavior
If a neuron spikes, the time step is saved here.
U
V
Type
Spike Log Stack
Indicates the number of saved time steps
Stack Pointer
Indicates the neuron that will recieve the axonal output.
Synaptic Connectivity Array
Synaptic Weight Array
Indicates the synaptic weight of the axonal output.

Fig. 2. Representation of a neuron in a C structure. The synaptic array is compounded of a determinate amount of these structures

an easy and flexible way to exploit the inherent parallelism on NVIDIA's GPUs. In addition, NVIDIA deployed a specialized hardware driver to exploit CUDA's computational capabilities. CUDA architecture is based on the GPU dedication to data processing, which means that NVIDIA's GPUs contain a limited cache and control area, but an increased number of arithmetic logic units (ALUs) (refers to figure 1).

This allows the partitioning of an algorithm in multiple processing blocks that can be treated as separate entities on the GPU's ALUs. Also, these processing blocks can be divided into finer pieces called execution threads to solve the algorithm cooperatively in parallel. It is important to remark that every thread shares the same execution code. Therefore, a neural network implementation can be divided into multiple processing blocks and threads to achieve a parallel separated computation with the same code in every thread.

There is a wide range of memory scopes implemented in CUDA: A global memory that is persistent across kernel launches in the same application, a two read-only global memory spaces (constant and texture memory), a private memory space in every CUDA thread and a shared memory that can be accessed by every thread within a block. It is important to remark that the shared memory has the same lifetime that the block that contains it.

III. Our SNN Implementation.

Our implementation consists mainly of two subsystems: A synaptic connections' editor and a SNN modeler. The synaptic connections' editor generates a neuronal topology and all the synaptic weights into a binary file. The connectivity and synaptic weights can be randomly generated or manually edited to create any required topology. Also, the synaptic connections' editor can define the number of synapses, neurons' population size, and the different types of behaviour in the neural model. The SNN modeler receives a series of parameters:

1) The maximum amount of neurons per execution.
2) Particularly on the Izhikevich's model the portion of excitatory and inhibitory neurons.
3) The amount of time steps that will be employed.
4) The number of CUDA's execution blocks and threads per block.
5) The synaptic connections input file's paths.
6) The spikes output file's path.
7) The execution time output file's path.

All parameters are contained in a single configuration file; because of this, several configuration files can be created to model different simulations and scenarios.

Using these parameters, the SNN modeller allocates the necessary memory in RAM to generate a dynamic array of neural structures (structs). This neural structure contains a membrane potential variable, membrane recovery variable, a fixed-length synapse array, a fixed-length spike log stack and an auxiliary stack pointer (see figure 2). Optionally, the model's parameters can be stored inside the structure instead of being stored in a static read-only applied-for-all neurons' value. These values can be modified to adapt a particular neural behaviour type.

The next step consists on creating a copy of the newly-generated dynamic array into the GPU's global memory. In the static read-only global memory the model's parameters are also copied if needed in order to reduce the used GPU global memory, which is a scarce computational resource.

The model's parameters can be stored statically in the read-only constant memory, or it can be stored inside the neural structure as extra variables. There are a couple of advantages and disadvantages of both implementations:

1) In the case of the constant read-only implementation the required memory reduces from $(4f(N_n))$ bytes to $(4f)$ bytes (where f is the size of "float" data type, which in a CPU and GPU architecture is typically 4 bytes, and N_n is the number of neurons employed). Also, constant read-only memory has been proved to be accessed faster than global memory [24], nevertheless all the neurons receive the same parameters; therefore, all neurons are bounded to be of the same type [7].
2) On the other hand, if the model's parameters are stored independently inside of the proposed structure as extra variables, different types of neurons can be simulated in the same model [7]. However, the disadvantages

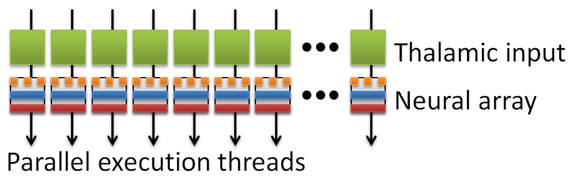

Fig. 3. Every thread is devoted to the calculation of the neurons' equation in the array, using the thalamic input vector.

of this approach are a slower access to the model's parameters and an increased amount of required GPU memory.

As a third step, every neuron in the dynamic array of structures is assigned to an execution thread and block in order to commence a parallel three-step simulation procedure:

1) Neural model: Every thread approaches the solution for its respective neuron's ordinary differential equation in a single time step, using a thalamic input vector, their respective membrane potential value and the model's parameters. The employed algorithm is forward Euler. (see figure 3).
2) Neuron evaluation: Every thread and block evaluates the membrane potential value and determines if a neuron has entered a spike status, if a spike was generated, the neuron sets a flag in an array of spiked neurons, and the time step is saved in the neuron's spike log stack. If the stack of any neuron is full, all the spike log stacks are copied to RAM and then deleted from the GPU (see algorithm 1).
3) Next time step's input calculation: If a neuron is on a spike status, every thread and block realizes a parallel vector sum between the next step inputs and the synaptic weight array (see figure 4).

Algorithm 1 GPU evaluation

```
1: for Every neuron do
2:   if The neuron is on a spike status then
3:     Set the spike flag array at the neuron's ID position.
4:     Save in the neuron's spike log stack the current time
       step.
5:     Increment the neuron's stack pointer.
6:     if The stack is full then
7:       Empty stack on CPU RAM
8:       Reset the neuron's stack pointer.
9:     end if
10:  end if
11: end for
```

At the end of the simulation all the gathered data (execution time and accumulated spike logs) are transferred from GPU's RAM to normal RAM and finally, written down in the corresponding output file.

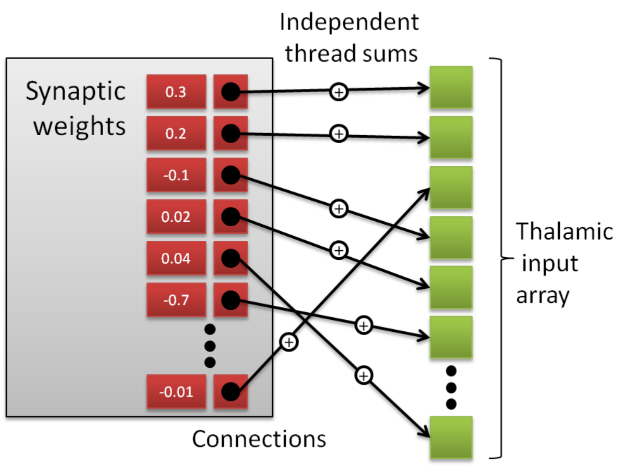

Fig. 4. For every neuron in a spike status, a parallel sum is calculated using the synaptic array and the thalamic input.

IV. Experimental Results

We realized a couple of experiments applying the Izhikevich's SNN model in our implementation to realize comparatives employing the same hardware environment: an eight-cored Intel Xeon CPU @2.6 GHz with 24 GB on RAM and a NVIDIA Tesla c2050 GPU with an internal GPU RAM of 2687 Mbytes and 448 CUDA cores @1.15 GHz each. In these experiments the model's parameters are stored inside the neuronal structure. Because the used data types have the same size in bytes in CPU and GPU architectures, there is no difference between the experiments in terms of numerical precision (In other words, there are no jitters on the spiking behaviour and timings).

A. Comparative between the original Izhikevich's implementation, a simple C implementation and our proposed implementation.

We developed an implementation of the Izhikevich's SNN model using standard C language without CUDA architecture to achieve the same spiking results as [7]. After that we posed a comparative between Izhikevich's original implementation, the Izhikevich's algorithm C language implementation and our proposed CUDA implementation in terms of execution time. The proposed experiments employ the same amount of neurons and synapses per neuron to generate the same spiking behavior as the Izhikevich's original implementation. To test different scenarios we also increased the amount of neurons and synapses gradually from 1,000 neurons and synapses to 7,000 neurons and synapses, and measured the execution time in every increment.

As shown on table I and figure 5, on the first experiments the difference between any implementation in terms of execution speed is almost the same, but once an increment in neuron or synapses per neuron is required our implementation excels in performance because it needs less execution time to obtain the same results. This increment in performance allows us to test huge neuronal connections

TABLE I
EXECUTION TIME OF SNN IMPLEMENTATIONS, MEASURED IN SECONDS (LESS IS BETTER).

# of Neurons	# of Synapses	Izhikevich's impl.	C impl.	Proposed impl.
1K	1K	0.129	0.18	0.698
2K	2K	1.626	1.124	1.998
3K	3K	7.6902	14.588	5.584
4K	4K	54.845	74.946	15.752
5K	5K	129.4038	148.328	18.126
6K	6K	201.204	286.048	28.078
7K	7K	268.4484	388.524	32.76

Every implementation is using one thousand ms time steps, a normally distributed random thalamic input with mean 0 and variance 1 (generated in GPU using CUDA random generator libraries), and random neuronal parameters, which are similar to the ones used in [7]. The results presented here are five execution's average time.

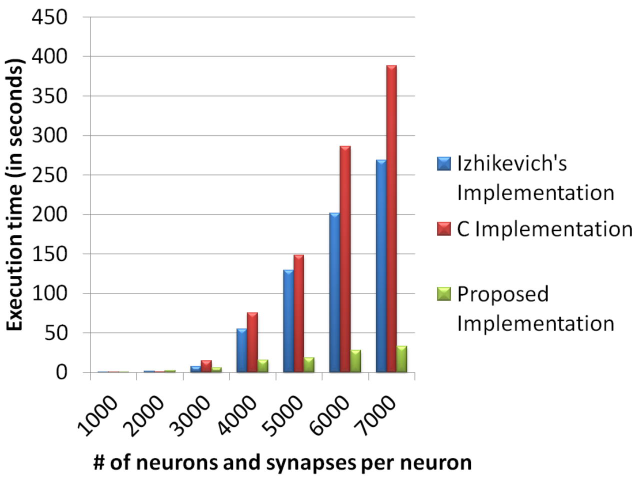

Fig. 5. Average execution time versus number of neurons and synapses per neuron in the SNN implementations. (Less is better)

using an extended quantity of neurons, as exhibited in the next experiment.

B. Performance tests: Huge neural connections

We realized performance tests employing a huge number of neurons, but with a constant amount of synaptic connections and time steps. Our goal in this experiment was to achieve the highest possible amount of synaptic connections with a thousand time steps employing as low execution time as possible. Again, our implementation is using the Izhikevich's SNN model adapted in CUDA C, with a standard (mean 0, variance 1) normally distributed random thalamic input generated in GPU with CUDA random generation libraries.

The results in table II clearly demonstrate that even a huge amount of connections can be simulated on a low time, leading to any possible application of a SNN that requires a fast response and a huge amount of neurons simultaneously.

TABLE II
PERFORMANCE TEST RESULTS USING VARIABLE NUMBER OF NEURONS.

# of Neurons	# of connections	Execution time	Firing Rate	Spikes per second
1K	1×10^6	0.7	6.389	9,125.71
2K	2×10^6	1.094	12.5166	11,490.859
5K	5×10^6	1.26	31.217	24,846.03
10K	1×10^7	2.47	63.1872	25,597.165
15K	1.5×10^7	3.644	95.132	26,111.96
20K	2×10^7	4.832	126.743	26,188.53
50K	5×10^7	12.27	319.5132	26,048.65
75K	7.5×10^7	19.212	478.4436	24,892.723
100K	1×10^8	25.712	638.0754	24,815.68
150K	1.5×10^8	39.734	956.4144	24,084.86
200K	2×10^8	54.802	1277.0678	23,294.16
225K	2.25×10^8	64.286	1435.3274	22,329.71

The number of synapses in this experiment is constant (1k per neuron). The total amount of connections is the multiplication of number of neurons and number of synapses per neuron. The execution time and spiking rate are calculated as the average value of five experiments (Execution time is measured in seconds and firing rate = generated spikes / time steps).

C. Comparative between different neuron-synapse relations

As a third experiment, we proposed a comparative between several pairs of connections; in every pair both neural networks have the same amount of connections but different amount of neurons and synapses per neuron: one of them employs a *1:1* neuron-synapse relation and the other utilizes a *100:1* neuron-synapse relation. We incremented the range of connections from 1 million to 100 million connections and measured execution times in every pair.

TABLE III
COMPARATIVE OF EXECUTION TIMES BETWEEN DIFFERENT NEURONAL - SYNAPSES RELATIONS.

# of Neurons	# of Synapses	# of connections	Execution time	Spikes per second
10K	100	1×10^6	2.3	23,387.39
1K	1K	1×10^6	0.696	9,137.93
20K	200	4×10^6	4.626	23,576.30
2K	2K	4×10^6	1.806	28,148.39
30K	300	9×10^6	7.334	22,622.71
3K	3K	9×10^6	3.9	109,649.23
40K	400	1.6×10^7	9.808	22,900.79
4K	4K	1.6×10^7	15.278	163,198.84
50K	500	2.5×10^7	11.938	23,948.90
5K	5K	2.5×10^7	18.38	224,720.56
60K	600	3.6×10^7	14.452	24,250.27
6K	6K	3.6×10^7	28.14	210,708.56
70K	700	4.9×10^7	16.97	24,600.88
7K	7K	4.9×10^7	32.778	211,137.03
80K	800	6.4×10^7	19.514	24,910.83
8K	8K	6.4×10^7	39.898	179,704.04
90K	900	8.1×10^7	21.964	25,478.60
9K	9K	8.1×10^7	45.678	194,918.516
100K	1K	1×10^8	24.516	26,024.31
10K	10K	1×10^8	51.474	192,283.424

The amount of time steps used is constant (1000 time steps), and execution time is measured in seconds (less is better). Notice that the amount of neuronal connections is calculated by the multiplication of the amount of neurons and synapses per neuron.

Our results in figure 6 and table III indicates that a *1:1*

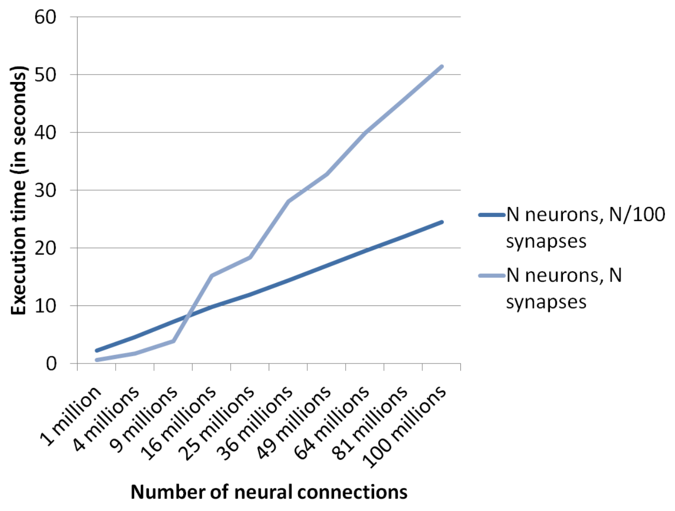

Fig. 6. Comparative between different neuronal-synapses relation execution times in seconds (less is better). In the X axis the amount of neuronal connections is shown. Notice that the amount of neuronal connections is calculated by the multiplication of the amount of neurons, synapses per neuron and time steps.

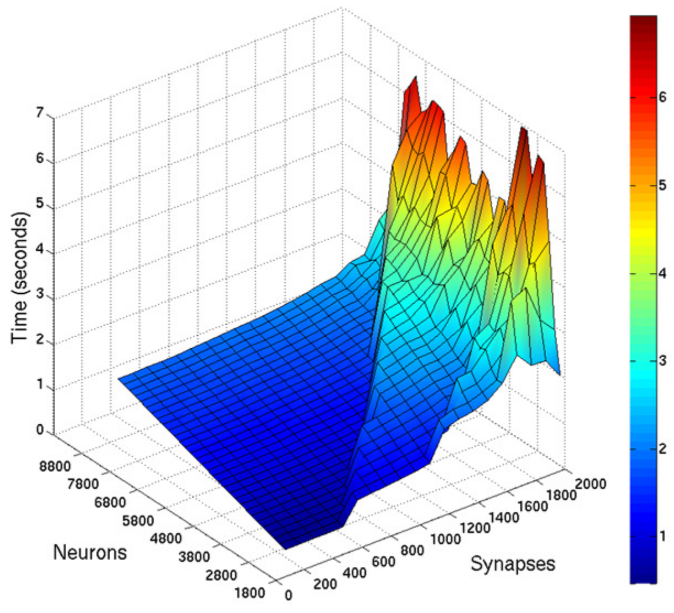

Fig. 7. Real time simulations' experiments: Employed time in seconds for every combination of amount of neurons and synapses per neuron. The results presented in this graphic are the average of 3 executions.

neuron-synapse relation requires more execution time than a *100:1* relation when the number of neurons and synapses per neuron increases. Due to the increment of generated spikes per second of execution time, a greater demand of computing resources are required in the simulation.

D. Proposed topologies with real-time simulations.

Another proposed experiment was the simulation of real-time SNN topologies: the maximum amount of neurons and synapses per neuron with the constraint of a very low execution time. Using the same 1,000 time steps we have proposed 620 different configurations: from 200,000 connections to 16,000,000 connections every time step. Our goal is to determine the biggest connection within an execution time of 2 seconds and 1 second for real time simulations.

With an execution time constraint of less than 2 seconds, we managed to compute 11,560,000 connections per time step in 1.98 seconds (6800 neurons, 1700 synapses, 1000 time steps). Also, with an execution time constraint of a second or less, our best result was 3,800,000 simulated connections per time step (3800 neurons, 1000 synapses, 1000 time steps in exactly 1 second) as shown in figure 7.

E. Proposed implementation under different conditions and operative environments.

To prove the versatility of our implementation, we applied our proposal in several hardware environments. Our objective on such experiments was to prove that even on clearly inferior hardware our implementation can surpass the execution speed of Izhikevich's original and serialized C implementation even under the best conditions. The employed environments are:

1) Environment 1 (Original computer, the experiments' best conditions):
   a) Intel Xeon CPU (8 cores) @2.6 GHz (64 bits).
   b) 24 GB RAM.
   c) NVIDIA Tesla c2050 GPU with an internal GPU RAM of 2687 Mbytes and 448 CUDA cores @1.15 GHz each.
2) Environment 2 (Original computer with a different GPU):
   a) Intel Xeon CPU (8 cores) @2.6 GHz (64 bits).
   b) 24 GB RAM.
   c) NVIDIA Quadro 4000 GPU with an internal GPU RAM of 2048 Mbytes and 256 CUDA cores @950 MHz each.
3) Environment 3 (a totally different environment):
   a) Dell XPS 710 with Intel Core 2 Duo (2 cores) @2.40Ghz (32 bits).
   b) 4 GB RAM.
   c) NVIDIA GeForce GTX 460 SE GPU with an internal GPU RAM of 1024 Mbytes and 288 CUDA cores @1320 MHz each.
4) Environment 4 (another different environment):
   a) Laptop Dell Alienware M11x R2 with Intel Core i7 (4 cores) @1.20Ghz (64 bits).
   b) 4 GB RAM.
   c) NVIDIA GT 335M GPU with an internal GPU RAM of 1024 Mbytes and 72 CUDA cores @450 MHz each.

Table IV, V and figure 8 demonstrates that even on inferior environments our implementation excels the Izhikevich's original implementation and the simple C implementation. This peculiar characteristic allows us to generate SNN implementations and applications without the requirement of

TABLE IV

Comparative among the Izhikevich's implementation, the serialized C implementation and our proposal in environment 1 and 2.

# of connections	C Impl	Izhikevich's Impl	Prop. Impl. Env. 1	Prop. Impl. Env. 2
1×10^6	0.18	0.129	0.698	1.684
4×10^6	1.124	1.626	1.998	4.344
9×10^6	14.588	7.69	5.584	14.296
1.6×10^7	79.946	54.845	15.752	23.536
2.5×10^7	148.328	129.4038	18.126	32.838
3.6×10^7	286.048	201.204	28.078	50.276
4.9×10^7	388.524	268.4484	32.76	64.878

Notice that the C implementation and Izhikevich's implementation uses the hardware environment 1 (the best available)

TABLE V

Comparative among the Izhikevich's implementation, the serialized C implementation and our proposal in environment 3 and 4.

# of connections	C Impl	Izhikevich's Impl	Prop. Impl. Env. 3	Prop. Impl. Env. 4
1×10^6	0.18	0.129	0.994	0.856
4×10^6	1.124	1.626	2.222	2.122
9×10^6	14.588	7.69	8.954	7.496
1.6×10^7	79.946	54.845	13.814	18.374
2.5×10^7	148.328	129.4038	21.804	33.984
3.6×10^7	286.048	201.204	34.668	58.504
4.9×10^7	388.524	268.4484	40.512	81.678

Notice that the C implementation and Izhikevich's implementation uses the hardware environment 1 (the best available)

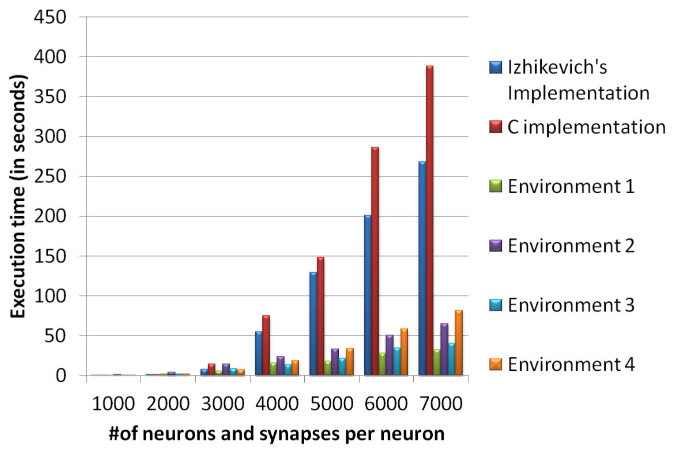

Fig. 8. Comparative among the Izhikevich's implementation, the serialized C implementation and all the posed environments.

powerful hardware in less time than the application of a serialized C or MATLAB implementation.

V. Conclusions

In this paper, we explained broadly our implementation of a SNN modeler and its composition (see algorithm 2). In our experimental results we have confirmed that our proposed SNN implementation excels the Izhikevich's original implementation and even a serialized Izhikevich's model written on C language in terms of execution speed and performance. In other experimental results posted on this paper, we tested the computability of huge neural connections in a considerably low time (200,000,000 neural connections every time step in less than a minute), the time differences between two different neuron-synapse relations with their respective increments in neuron population and synapse per neuron, and connections with execution time constraints of 2 seconds (11,560,000 connections computed every time step) and a one second, real-time simulation (3,800,000 connections computed every time step). Finally, we tested our implementation in different hardware environments and obtained a dramatically boosted performance. Such performance excels the Izhikevich's original and serialized C implementation even on the disadvantages of a clearly inferior hardware. This proposal allows us to explore future research and development such as the optimization of several training algorithms(STDP, genetic algorithms, etc.) an intuitive synaptic connections' editor graphic user interface, improvements on the proposed model in terms of the employment of simultaneous GPUs cooperatively, and a set of applications focused on specific areas such as computer vision, pattern recognition and biologically realistic cortical circuits' simulations.

VI. Considerations.

The size of the neural array in bytes is calculated using the equation:

$$S = N_n(2f + u(N_l + 1) + N_s(f + u) + c) \quad (2)$$

Where S is the total amount of bytes used, N_n is the number of neurons used in the execution, f is the size of the "float" data type (typically 4 bytes), u is the size of "unsigned int" data type (typically 4 bytes), N_l is the size of the log stack, N_s is the size of the synaptic array and c is the size of "char" data type (typically 1 byte). If Izhikevich's model's variables A, B, C & D are used as a non-static randomly similar to [7], floating point value, the equation is modified to be:

$$S = N_n(6f + u(N_l + 1) + N_s(f + u) + c) \quad (3)$$

The size of the neural array S cannot exceed the maximum amount of bytes available in the GPU's RAM. It may be a common pitfall the swapping between host RAM and device RAM, but this approach is computationally costly in terms of execution time because the CUDA memory copy function is relatively slow [24]. This can lead to poor performance in a neural simulation because in this approach several copy operations are needed for a single time step. Due to lack of space in this paper we only post comparatives using the Izhikevich's model. Nevertheless, any other model such as Morris-Lecar, integrate-and-fire or Hodgkin-Huxley can be employed.

Algorithm 2 Proposed SNN implementation's algorithm in pseudocode

```
1: Allocate the neural array.
2: Allocate the thalamic input array.
3: Load the synapses from file.
4: Initialize the neural variables.
5: Copy the neural array to GPU RAM.
6: if Neurons' parameters are static then
7:     Set Neural static values on read-only memory space.
8: end if
9: if The thalamic input is generated from a file then
10:    Read the thalamic input from file.
11:    Copy the thalamic input to GPU RAM.
12: else
13:    Generate a noisy input in GPU.
14: end if
15: Start the timer.
16: for Each time step do
17:    Realize the parallel neural equation.
18:    if The thalamic input is generated from a file then
19:        Read the thalamic input from file.
20:        Copy the thalamic input to GPU RAM.
21:    else
22:        Generate a noisy input in GPU.
23:    end if
24:    Realize the parallel neural evaluation.
25:    if The time step log stack overflows then
26:        Save the stacks on CPU RAM.
27:        Reset the stacks to zero.
28:    end if
29:    Do a parallel vector sum between synaptic weights
       and inputs.
30:    Increment the time step counter.
31: end for
32: End the timer.
33: Save Stacks from GPU RAM to CPU RAM.
34: Save Stacks from CPU RAM to HDD.
35: Save the execution time.
36: Free the neural array in CPU & GPU RAM.
37: Free the thalamic input in CPU & GPU RAM.
```

REFERENCES

[1] S. Ghosh-Dastidar and H. Adeli, "A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection," *Neural Networks*, vol. 22, pp. 1419–1431, 2009.

[2] D. P. M. T. G. Brendan. and W. Qing, "Performance comparison of a biologically inspired edge detection algorithm on cpu, gpu and fpga." *International Conference on Fuzzy Computation and 2nd Int Conference on Neural Computation.*, 2010.

[3] A. R. Ormondi and J. C. Rajapakse, *FPGA implementations of neural networks*. Netherlands: Springer, Dordrecht, 2006.

[4] E. Izhikevich and G. Edelman, "Large-scale model of mammalian thalamocortical systems." *Neural Networks*, vol. 23, pp. 16–19, 2009.

[5] J.-P. Tiesel and A. S. Maida, "Using parallel gpu architecture for simulation of planar i/f networks." *Proceedings of international joint conference on neural networks*, pp. 3118–3123, 2009.

[6] J. H. K. E.-L. F. Güttler. and M. Bodgan, "Simulating biological-inspired spiking neural networks with opencl," *Springer*, vol. 6352, pp. 184–187, 2010.

[7] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, pp. 1569–1572, 2003.

[8] ——, *Dynamical systems in neuroscience*, ser. Neural Networks, E. M. Izhikevich, Ed. Massachusetts, United states of america: Massachusetts Institute of Technology, 2007.

[9] W. Gerstner and W. Kistler, *Spiking neuron models*. New York, United states of america: Cambridge university press, 2009.

[10] R. A. Vazquez, "Izhikevich neuron model and its application in pattern recognition," *Australian Journal of Intelligent Information Processing Systems*, vol. 11, no. 1, pp. 53–60, 2010.

[11] ——, "Training spiking neural models using cuckoo search algorithm," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*, june 2011, pp. 679 –686.

[12] R. A. Vazquez and A. Cachon, "Integrate and fire neurons and their application in pattern recognition," in *Electrical Engineering Computing Science and Automatic Control (CCE), 2010 7th International Conference on*, sept. 2010, pp. 424 –428.

[13] L. Gomes Wysoski, Simei. Benuskova and N. Kasabov, "Evolving spiking neural networks for audiovisual information processing," *Elsevier neural networks*, vol. 23, pp. 819–835, 2010.

[14] R. A. Vazquez and J.-C. Q. Bernard Girau, "Visual attention using spiking neural maps," in *International Joint Conference on Neural Networks. IJCNN 2011*, june 2011.

[15] R. A. Vazquez, "A computational approach for modeling the biological olfactory system during an odor discrimination task using spiking neuron," *BMC Neuroscience*, vol. 12, no. supp1, p. P360, 2011.

[16] Z. M. Haza Nuzly Abdull Hamed, Nikola Kasabov and S. M. Shamsuddin, "String pattern recognition using evolving spiking neural networks and quantum inspired swarm organization." *Lecture Notes in Computer Science*, vol. 5864, pp. 611–619, 2009.

[17] W. Gerstner and J. L. van Hemmen, "Associative memory in a network of spiking neurons," *Network: Computation in Neural Systems*, vol. 3, no. 2, pp. 139–164, 1992.

[18] F. Bernhard and R. Keriven, "Spiking neurons on gpus," *Research report 05-15*, 2005.

[19] J.-P. Tiesel and A. S. Maida, "Using parallel gpu architecture for simulation of planar i/f networks," *Proceedings of international joint conference on neural networks*, pp. 3118–3123, 2009.

[20] J. M. N. N. D. J. L. K. Nicolau and A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using cuda graphics processors," *International Joint Conference on Neural Networks*, pp. 2145–2152, 2009.

[21] F. A. K. R. E. B. S. M. P. and L. Wayne, "Nemo: A platform for neural modelling of spiking neurons using gpus." *Application-specific Systems, Architectures and Processors.*, pp. 137–144, 2009.

[22] S. Haykin, *Neural networks and learning machines*, P. Hall, Ed. Pearson education, 2009.

[23] N. Corporation, *NVIDIA CUDA C Programming guide*, N. corporation, Ed., 2010.

[24] J. Sanders and E. Kandrot, *CUDA by example*, N. corporation, Ed. Massachussets, United States of America: Addison wesley, 2009.