An Improved GPU Simulator For Spiking Neural P Systems

Francis George C. Cabarle
Algorithms and Complexity lab Dept. of Computer Science University of the Philippines Diliman
fccabarle@up.edu.ph

Henry Adorna
Algorithms and Complexity lab Dept. of Computer Science University of the Philippines Diliman
ha@dcs.upd.edu.ph

Miguel A. Martínez–del–Amor
Research Group on Natural Computing Dept. of Computer Science and AI University of Seville, Spain
mdelamor@us.es

***Abstract*—Spiking Neural P (SNP) systems, variants of P systems (under Membrane and Natural computing), are computing models that acquire abstraction and inspiration from the way neurons 'compute' or process information. Similar to other P system variants, SNP systems are Turing complete models that by nature compute non-deterministically and in a maximally parallel manner. P systems usually trade (often exponential) space for (polynomial to constant) time. Due to this nature, P system variants are currently limited to parallel simulations, and several variants have already been simulated in parallel devices. In this paper we present an improved SNP system simulator based on graphics processing units (GPUs). Among other reasons, current GPUs are architectured for massively parallel computations, thus making GPUs very suitable for SNP system simulation. The computing model, hardware/software considerations, and simulation algorithm are presented, as well as the comparisons of the CPU only and CPU-GPU based simulators.**

***Keywords*-Computational modeling; Parallel processing; Multicore processing**

I. Introduction

Membrane computing uses *P systems* (named after their inventor, Gheorghe Păun) as computing models and was introduced in 1998 [6]. The objective, as with other disciplines of *natural computing* (e.g. DNA/molecular computing, quantum computing, etc.) is to obtain inspiration and abstraction from the way nature (in this case $cells$) computes. By 'compute' we mean to say the system (whether formal/mathematical in the case of SNP systems, or biological as in real living cells) processes information: data is read from memory, gets processed and is acted on accordingly due to some rules and environmental stimuli, and is written back to memory for use in future processes [2]. Another objective, among others, is to be able to solve current and perhaps newer hard problems (e.g. NP-complete) and go beyond the classical model of computation, the Turing machine. We obtain ideas from the way nature computes, since nature has been efficiently doing so for billions of years (as current researches point out nature itself can solve lots of our hard problems), and thus we introduce unconventional models of computation from the area of natural computing. Membrane computing can be thought of as an extension of DNA or molecular computing, zooming out from the individual molecules of the DNA and including other parts and sections of the cell in the computation, introducing the concept of distributed computing [6].

P systems (most variants at least) compute in a non-deterministic and maximally parallel manner, oftentimes requiring exponential space as trade off to solve hard problems in polynomial to even constant time. However, due to this nature and trade off, P systems are yet to be fully implemented *in vivo*, *in vitro*, or even *in silico*. We thus refer to their simulations using parallel devices such as *GPUs* to further study them.

Since P systems were introduced, many simulators using different parallel devices have been produced [11], including computer clusters [12], reconfigurable hardware as in FPGAs [13], as well as GPUs [9], [8]. These efforts show that parallel devices are very suitable in simulating P systems, at least for the first few P system variants to have been introduced. Efficiently simulating SNP systems would thus require new attempts in parallel devices.

GPUs on the other hand are currently one of the foremost candidates for simulating P systems due to several significant reasons. One is that because of GPGPU computing (general purpose GPU computing), their architecture which is specifically designed for massively parallel computations, are laid bare to programmers [1]. Programmers aren't limited to graphics processing alone, as was done in the early days of GPUs. Instead, general purpose computations such as trigonometric and linear algebra operations can now be performed on GPUs. Another reason is that GPUs offer very large speedups versus CPU only implementations, including clustered CPUs, by consuming less energy at the fraction of the cost of setting up and maintaining CPU clusters [14], [15]. Parallel computing concepts such as hardware abstraction, scaling, and so on are also handled efficiently by current GPUs.

Given that SNP systems have already been represented as matrices due to their graph-like properties [5], simulating them in parallel devices such as GPUs is the next natural step. Matrix algorithms are well known in parallel computing literature, including GPUs [16], [17], due to the highly parallelizable nature of linear algebra computations mapping directly to the data-parallel architecture of GPUs.

Previously, SNP systems have been faithfully implemented in GPUs using their matrix representation [10]. We thus extend this previous work to improve the performance of the simulator, as well as include speedups comparing the CPU only simulator versus the CPU-GPU simulator.

This paper is organized as follows: Section II introduces

SNP systems formally, as well as their matrix representation. Section III provides background for GPGPU computing with CUDA. The design of the simulator and simulation results are given in Section IV and Section V, respectively. Finally, conclusions, future work, acknowledgements, and references end this paper.

II. Spiking Neural P systems

A. Computing with SN P systems

Within SNP systems there are further variations, such as those without delays, those with extended rules, deterministic systems, and so on. Many of these SNP system $variants$ have been shown to be *Turing complete* [3], [4]. The type of SNP systems focused on by this paper (scope) are those without delays, and they are of the form:

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out),$$

where:

1. $O = \{a\}$ is the alphabet made up of only one object a, called spike.
2. $\sigma_1, \ldots, \sigma_m$ are m number of neurons of the form
$$\sigma_i = (n_i, R_i), 1 \leq i \leq m,$$
where:
   (a) $n_i \geq 0$ gives the initial number of as i.e. spikes contained in neuron σ_i
   (b) R_i is a finite set of rules of the following forms:
      (b-1) $E/a^c \rightarrow a^p$, are known as *Spiking rules*, where E is a regular expression over a, and $c \geq 1$, such that $p \geq 1$ number of spikes are produced, one for each adjacent neuron with σ_i as the originating neuron and $a^c \in L(E)$.
      (b-2) $a^s \rightarrow \lambda$, are known as *Forgetting rules*, for $s \geq 1$, such that for each rule $E/a^c \rightarrow a$ of type (b-1) from R_i, $a^s \notin L(E)$.
      (b-3) $a^c \rightarrow a$, a special case of (b-1) where $L(E)$ = $\{a^c\}$, $k = c, p = 1$.
3. $syn = \{(i,j) \,|\, 1 \leq i,j \leq m, i \neq j\}$ are the synapses i.e. connection between neurons.
4. $in, out \in \{1, 2, \ldots, m\}$ are the input and output neurons, respectively.

Rules of type (b-1) are applied if σ_i contains k spikes, $a^k \in L(E)$ and $k \geq c$. Using this type of rule consumes k spikes from the neuron, producing a spike to each of the neuron/s connected to it via the syn graph. In this manner, for rules of type (b-2) if σ_i contains s spikes, then s spikes are 'forgotten' or removed from the neuron once the rule is applied.

The neurons in an SNP system operate in parallel and in unison, under a *global clock* [3]. However, only one rule can be applied at a given time in each neuron [3], [5]. The *non-determinism* of SNP systems come with this fact: if more than one rule is applicable at a given time, given enough spikes in a neuron, then the rule to be applied is chosen non-deterministically.

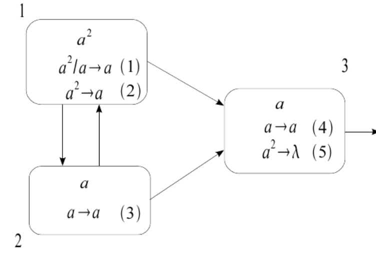

Figure 1. An SNP system Π, generating all numbers in the set $\mathbb{N}$ - $\{1\}$, from [5].

The SNP system Π shown in Figure 1 generates all numbers in the set $\mathbb{N}$ - $\{1\}$ once all rules are applied non-deterministically, given initial conditions. The computation halts on each element of the set Π generates. The *outputs* of the computation of Π are derived from the time difference between the first spike of the output neuron (to the environment) and its succeeding spikes. It can be seen that a total system ordering is given to neurons (from (1) to (3)) and rules (from (1) to (5)) of the system. This SNP system is of the form $\Pi = (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, out)$ where $\sigma_1 = (2, \{R_1, R_2\})$, $n_1 = 2$, $R_1 = \{a^2/a \rightarrow a\}$, $R_2 = \{a^2 \rightarrow a\}$, (neurons 2 to 3 and their n_is and R_is can be similarly shown in the same manner), $syn = \{(1,2), (1,3), (2,1), (2,3)\}$ are the synapses for Π, and the output neuron $out = \sigma_3$, as can be seen by the arrow not pointing to any neuron. Π has no input neuron.

B. Matrix representation of SNP systems

In [5], a matrix representation of SNP systems without delays was introduced. This representation makes use of the following vectors and matrix definitions:

Configuration vector C_k is the vector containing all spikes in every neuron on the kth computation step/time, where C_0 is the initial vector containing all spikes in the system at the beginning of the computation. For Π (the example in Figure 1) $C_0 =< 2, 1, 1 >$.

Spiking vector S_k shows, at a given configuration C_k, if a rule is applicable (having value *1*) or not (having value *0* instead). For Π we have a spiking vector $S_k =< 1, 0, 1, 1, 0 >$ given C_0. Note that a second spiking vector, $S_k =< 0, 1, 1, 1, 0 >$, is possible if we use rule (2) over rule (1) instead (but not both at the same time). Given C_0 of Π we thus cannot have an $S_k =< 1, 1, 1, 1, 0 >$ so this S_k is invalid. $Validity$ in this case means that only one among several applicable rules (chosen non-deterministically) is used and thus represented in a spiking vector for a given σ. We recall the applicability of rules from the definitions of (b-1) to (b-3).

Spiking transition matrix M_{SNP} is a matrix comprised of a_{ij} elements where a_{ij} is given as

$$a_{ij} = \begin{cases} -c, & \text{rule } r_i \text{ is in } \sigma_j \text{ and is applied consuming } c \text{ spikes;} \\ p, & \text{rule } r_i \text{ is in } \sigma_s \ (s \neq j \text{ and } (s,j) \in syn) \text{ and is applied producing } p \text{ spikes in total;} \\ 0, & \text{rule } r_i \text{ is in } \sigma_s \ (s \neq j \text{ and } (s,j) \notin syn). \end{cases}$$

In such a scheme, rows represent rules and columns represent neurons. For Π, the M_{SNP} is as follows:

$$M_{\Pi} = \begin{pmatrix} -1 & 1 & 1 \\ -2 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & -2 \end{pmatrix} \quad (1)$$

Finally, the following equation provides the configuration vector at the $(k+1)th$ step, given the configuration vector (C_k) and spiking vector (S_k) at the kth step, and M_{Π}:

$$C_{k+1} = C_k \ + \ S_k \cdot M_{\Pi} \quad (2)$$

III. Computing with GPUs

A. NVIDIA CUDA

NVIDIA, a well known manufacturer of graphics processors, introduced the Compute Unified Device Architecture (CUDA) in 2007 [14]. CUDA is a programming model and hardware architecture for general purpose computations in NVIDIA's GPUs (G80 and newer family of GPUs)[14]. CUDA, by extending popular languages such as C, allows programmers to easily create software that will be executed in parallel, avoiding low-level graphics and hardware primitives[15]. Among the other benefits of CUDA include abstracted and automated scaling: more cores will make the parallelized code run faster than GPUs with fewer cores [15]. GPUs introduce very large speedups over CPU only implementations in linear algebra computations (among other types of computations) because of the GPU architecture. The common CPU architectures are composed of transistors which are divided into different blocks to perform the basic tasks of CPUs (general computation): control, caching, DRAM, and ALU (arithmetic and logic). In contrast, only a fraction of the CPU's transistors allocated for control and caching are used by GPUs, since far more transistors are used for ALU [14]. This architectural difference is a very distinct and significant reason why GPUs offer larger performance increase over CPU only implementation of parallel code working on large amounts of input data.

A CUDA program is often divided into two parts: the *host* (CPU side) and the *device* (GPU side). The host/CPU part of the code is generally responsible for controlling the program execution flow, allocating memory in the host or device/GPU, and obtaining the results from the device. The device (or devices if there are several GPUs in the setup) act as *co-processors* to the host. The host outsources the parallel part of the program as well as the data to the device since it is more suited to parallel computations than the host. Code written for CUDA can be split up into multiple threads within multiple thread blocks, each contained within a grid of (thread) blocks. These grids belong to a single device/single GPU. Each device has multiple cores, each capable of running its own *block of threads* [14], [15]. A function known as a *kernel function* is one that is called from the host but executed in the device. Using kernel functions, the programmer can specify the GPU resources: the layout of the threads (from one to three dimensions) and the blocks (from one to two dimensions).

B. SNP system GPU simulation

A first version of an SNP system GPU simulator is described in [10] (we designate this as *snpgpu-sim1*). The focus in [10] was primarily on the simulation algorithm as well as the implementation of the computations (equation (2)) in parallel using the GPU. The comparison of the CPU only and CPU-GPU running times was not analysed in [10]. The *snpgpu-sim1* was also written in Python and C.

The reason for using Python was to be able to use an object oriented programming language (OOPL) since OOPLs are very suited in manipulating strings. Checking whether a rule is applicable given the regular expression E in a neuron requires manipulation of the matrix/vector elements (i.e. elements of *M*, S_k, C_k) as strings. The C language was then used to work on the elements when they are treated as integers, since C is very suited for computations involving integers and floating numbers. OOPLs provide the necessary expressivity for strings while the structured languages (e.g. C) are for integral computations. Also, the current dichotomy of the CUDA programming is to have the host/CPU work on the input/output parts (parsing, formatting etc) of the simulation (since CPU dedicates more transistors for these purposes) while the parts which can be done in parallel are sent to the GPU (since GPUs have more transistors for arithmetic operations).

The simulation algorithm for *snpgpu-sim1* is given in Algorithm 1, with 2 *Stopping criteria*: (I) if a *zero vector* (vector of zeros) is encountered, (II) if the succeeding C_ks have all been produced in previous computations. Both (I) and (II) make sure that the simulation *halts* and does not enter an infinite loop. Algorithm 1 points out which device/s (either **HOST**/CPU or **DEVICE**/GPU) a certain part of the simulation runs on. The file Ck is the file *counterpart* of the vector C_k, file M contains the matrix M_{Π}, and file r contains the rules from R_i.

Algorithm 1 Overview of SNP system simulation algorithm

Require: Input files: Ck, M, r.

I. **(HOST)** Load input files. M, r are loaded once only. $C0$ is also loaded once, then Cks afterwards.

II. **(HOST)** Determine if a rule in r is applicable based on the number of spikes present in each neuron/σ seen in Ck. Then, generate all valid and possible spiking vectors in Sk, a list of lists, given the 3 inputs.

III. **(DEVICE)** Run kernel function on all valid and possible Sk from the current Ck. Produce the next configurations, Ck +1 and their corresponding Sks.

IV. **(HOST+DEVICE)** Repeat steps I to IV, till at least one of the two *Stopping criteria* is encountered.

C. PyCUDA: CUDA Python wrapper

To further improve the performance of *snpgpu-sim1*, we designate our work in this paper as *snpgpu-sim2* and still involve the Python and C languages. However, even with the compiled part of *snpgpu-sim1* (done in C, as Python is commonly an interpreted language) we still call the C part from the Python part of *snpgpu-sim1*. This Python-C tandem is still used in *snpgpu-sim2* although there are significant changes in order to improve the performance of the simulator: (1) Included C code is very minimal, such that precisely only the parts which involve integer manipulation are handled by C (2) C is no longer called from outside the Python code since the C code is now *embedded* within the Python code, existing only as purely the kernel function and nothing else (3) the use of *PyCUDA* [7] to further abstract the memory allocation, cleanup and kernel function calls within Python.

PyCUDA was developed by mathematician Andreas Klöckner, which he used in his dissertation, in order to create safer (e.g. memory handling) and faster (in terms of development time) software using an OOPL such as Python [7]. *PyCUDA* is a wrapper or a programming interface of CUDA for Python, and has been used in numerous researches and real world applications [7]. Lines of code usually written in C simply become function calls in PyCUDA. Another reason for performance improvement using PyCUDA is that it uses *NumPy* to represent the data it manipulates. NumPy is a Python extension for large multi-dimensional arrays, matrices, and high-level mathematical functions and operations, and is an *open source* alternative to similar software such as *MATLAB*.

IV. Improved SNP system GPU simulator: *snpgpu-sim2*

The simulation setup for *snpgpu-sim2* (the same setup used in *snpgpu-sim1*) is an Apple iMac running Mac OS X 10.5.8, with an Intel Core2Duo CPU at 2.66GHz and with a 6MB L2 cache. The iMac has a CUDA enabled NVIDIA GeForce 9400 graphics card at a clock rate of 1.15 GHz, with 256MB VRAM, 16 cores and with CUDA version 3.1. The simulation algorithm used for *snpgpu-sim1* is the one used for *snpgpu-sim2* (Algorithm 1) although the reasons for performance increase have been mentioned in Subsection III-C.

The inputs for Algorithm 1 are loaded from *text files* (Ck, r, M, Sk) containing the elements of the vectors (C_k and S_k), rule list (R_i), and M_Π delimited by either white spaces or the '$' symbol. Elements of M_Π are entered in a *row-major order* (linear array of all the elements of the matrix, where rows are appended one after the other). The row-major ordering of M_Π is thus: 1, 1, 1, 2, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 2. Another common practice in CUDA programming (and in parallel computing in general) is the following: inputs are loaded first in the host, memory is allocated in the device (to receive the loaded inputs), inputs are then moved from the host to the device, the device executes parallel computations on the input data, the data is moved from the device back to the host, and the host outputs the data for other purposes. This practice is observed with the implementation of Algorithm 1.

Recalling equation (2), multiplication of M_Π to S_k is done by assigning each *element* of the two multiplicands to a *thread* in the device and then performing multiplication. This multiplication operation between S_k and M_Π (represented as a linear array) is essentially a dot product, although each element multiplication is done in parallel by the device i.e. all multiplications are done at the same time. Once the product is obtained, it is then added (again done in parallel, with one element addition per thread) to the elements of C_k. After equation (2) has been executed for all possible and valid S_ks for a given C_k, the C_{k+1}s are moved from the device back to the host to be checked again for the stopping criteria mentioned in Subsection IV (also shown in Algorithm 1). Current CPUs can certainly do dot products efficiently, but what sets GPUs apart is that they can efficiently and simultaneously compute dot products (among other operations) using tens of thousands of threads over hundreds of cores at greater speedups, and at the fraction of the cost and energy consumption of clusters and similar CPU only setups.

V. Simulation results and observations

Using *snpgpu-sim2* we simulate the SNP system Π shown in Figure 1. Given $C_0 = (2, 1, 1)$, Figure 2 shows the configuration tree for Π. C_ks followed by a (. . .) go deeper (i.e. produce more C_ks) than what is shown in the figure. It can be observed that the $C_k = (2, 0, 1)$ ends the configuration tree at that point ($C_k = (2, 0, 1)$ is thus a terminal or *leaf node* in the tree) because $C_k = (2, 0, 1)$ has already been produced from $C_0 = (2, 1, 1) \rightarrow (1, 1, 2) \rightarrow (2, 0, 1)$. Hence by the second stopping criteria we do not proceed after encountering $C_k = (2, 0, 1)$ the second time around.

To compare the performance of the GPU based SNP system simulator, Algorithm 1 was used to create a Python-C simulator but which is solely executed in the host/CPU. Thus, the parallel parts in *snpgpu-sim2* written using PyCUDA (Python + C) were written to run in the CPU only. The simulation comparison is as follows: The CPU version of the simulator (we designate this as *snpcpu-sim*) and *snpgpu-sim2* are given similar C_0 (initial configuration vector) as inputs, and are run at least three times (three trials) per C_0. The average of the three trials is then taken and the bar chart from the average of the trials are shown in Figure 3.

As seen in Figure 3, the five C_0 values used for the comparison are (2,1,2), (3,1,3), (4,1,4), (6,1,6), and (9,1,9). In Figure 3 we have as the horizontal axis the C_0 values, while the vertical axis is in seconds i.e. the running time of *snpcpu-sim* and *snpgpu-sim2*. As we increase the C_0 values for Π, it can be seen from Figure 3 that *snpgpu-sim2* performs better (i.e. simulation time average is lower) compared to *snpcpu-sim*. In fact, a *speedup* of up to 1.43 times is seen if Π is simulated using *snpgpu-sim2* over *snpcpu-sim*. It is worth noting that although the decrease in running time of *snpgpu-sim2* over *snpcpu-sim* (and

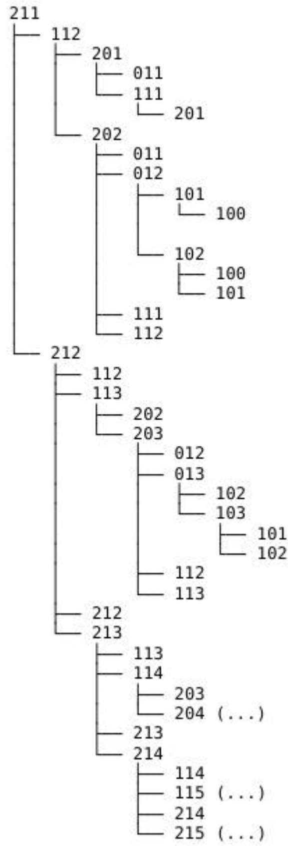

Figure 2. C_k tree for SNP system Π with $C_0 = (2, 1, 1)$.

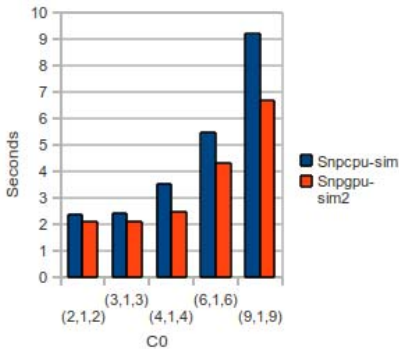

Figure 3. Graph of the average runtime of the CPU only simulator (*snpcpu-sim*) versus the CPU-GPU simulator (*snpgpu-sim2*) over different C_0 values.

hence the performance speedup) is obvious from Figure 3, the speedup wasn't very large because Π only has three neurons. The simulation done with Π was with increasing values of C_0 only.

The runtime charts for *snpgpu-sim1* (not shown in Figure 3) as compared to *snpcpu-sim* and *snpgpu-sim2* take longer times to finish compared to *snpcpu-sim*. The slowdown of *snpgpu-sim1* over *snpcpu-sim* is present even though the C_{k+1}s were computed in parallel (following in Algorithm 1) because of the following implementation reasons: (i) as mentioned in an earlier section, Python was calling the CUDA code written in C outside of the Python host code (ii) the C code for *snpgpu-sim1* did more than integer computations, as it was also tasked to write the output C_{k+1}s to files after allocating memory in the device for itself, among other things it did (recall improvements made to *snpgpu-sim2* from *snpgpu-sim1* in subsection III-C).

VI. Conclusions and Future Work

In this paper we have presented an improved SNP system simulator based on the GPU, using NVIDIA CUDA. The algorithm and the hardware-software design considerations used to improve the simulator *snpgpu-sim2* (using both the CPU and the GPU) over the CPU only simulator *snpcpu-sim* were also discussed. The results show that we have simulated the workings of the computing model (an SNP system without delays) and that *snpgpu-sim2* has significant performance speedups over *snpcpu-sim* especially for higher values of C_0. The use of the OOPL Python via the PyCUDA wrapper has sped up the simulator development and lowered the simulation time. Because SNP systems manipulate *spikes* as objects (represented as *strings* in Python) Python provided the necessary expressivity for string manipulation. The matrix representation however of the SNP system involved the manipulation of vector and matrix values as *integers* and this has also been successfully implemented by embedding minimal C code within the PyCUDA simulator.

For our future work, we would like to implement other variants of SNP systems, including those with delays, as well as more general regular expressions (those of the form (b-1)). The use of sparse matrix-vector implementations for the inputs (C_k, M, r) should also be included (since at the moment C_k, S_k and M are transformed into square matrices). Further understanding of the CUDA architecture (inter-thread communication and memory management for very large inputs/matrices) and the execution of the simulator in GPU clusters with newer GPUs and with more cores are planned. Finally, larger SNP systems (i.e. more neurons than Π) are also to be simulated in order to fully utilize the speedup and performance increase attributed to the massively parallel execution in GPUs.

Acknowledgments

Francis Cabarle is supported by the DOST-ERDT program. Henry Adorna is funded by the DOST-ERDT research grant and the Alexan professorial chair of the UP Diliman Department of Computer Science. Miguel A. Martínez–del–Amor is supported by "Proyecto de Excelencia con Investigador de Reconocida Valía" of the "Junta de Andalucía" under grant P08-TIC04200, and by the project TIN2009–13192 of the "Ministerio de Educación y Ciencia" of Spain, both co-financed by FEDER funds. We thank and acknowledge the anonymous referees for their detailed comments on this paper which helped improve it

References

[1] M. Harris, "Mapping computational concepts to GPUs", *ACM SIGGRAPH 2005 Courses*, NY, USA, 2005.

[2] M. Gross, “Molecular computation”, *Chapter 2 of Non-Standard Computation*, (T. Gramss, S. Bornholdt, M. Gross, M. Mitchel, Th. Pellizzari, eds.), Wiley-VCH, Weinheim, 1998.

[3] M. Ionescu, Gh. Păun, T. Yokomori, “Spiking Neural P Systems”, *Journal Fundamenta Informaticae* , vol. 71, issue 2,3 pp. 279-308, Feb. 2006.

[4] H. Chen, M. Ionescu, T.-O. Ishdorj, A. Păun, Gh. Păun, M. Pérez-Jiménez, “Spiking neural P systems with extended rules: universality and languages”. *Natural Computing: an international journal*, vol. 7, issue 2, pp. 147166, Jun. 2008.

[5] X. Zeng, H. Adorna, M. A. Martínez-del-Amor, L. Pan, M. Pérez-Jiménez, “Matrix Representation of Spiking Neural P Systems”, *11th International Conference on Membrane Computing*, Jena, Germany, Aug. 2010 and *Lecture Notes in Computer Science*, Springer, vol. 6501, pp. 377-39, 2011.

[6] Gh. Păun, G. Ciobanu, M. Pérez-Jiménez (Eds), *Applications of Membrane Computing*, Natural Computing Series, Springer, 2006.

[7] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA: GPU Run-Time Code Generation for High-Performance Computing”, Scientific Computing Group, Brown University, no. 2009-40, RI, USA, Nov. 2009

[8] J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, “Simulating a P system based efficient solution to SAT by using GPUs”, *Journal of Logic and Algebraic Programming*, Vol 79, issue 6, pp. 317-325, Apr. 2010.

[9] J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, “Simulation of P systems with active membranes on CUDA”, *Briefings in Bioinformatics*, Vol 11, issue 3, pp. 313-322, Mar. 2010.

[10] F. Cabarle, H. Adorna, M.A. Martínez-del-Amor, “Simulating Spiking Neural P systems without delays using GPUs”, *Proceedings of the 9th Brainstorming Week on Membrane Computing*, Sevilla, Spain, Feb. 2011.

[11] D. Díaz, C. Graciani, M.A. Gutiérrez, I. Pérez-Hurtado, M.J. Pérez-Jiménez. “Software for P systems”. In Gh. Păun, G. Rozenberg, A. Salomaa (eds.) *The Oxford Handbook of Membrane Computing*, Oxford University Press, Oxford (U.K.), Chapter 17, pp. 437-454, 2009.

[12] G. Ciobanu, G. Wenyuan. “P Systems Running on a Cluster of Computers”. *Lecture Notes in Computer Science*, 2933, 123-139, 2004.

[13] V. Nguyen, D. Kearney, G. Gioiosa. “A Region-Oriented Hardware Implementation for Membrane Computing Applications and Its Integration into Reconfig-P”. *Lecture Notes in Computer Science*, 5957, 385-409, 2010.

[14] D. Kirk, W. Hwu, *Programming Massively Parallel Processors: A Hands On Approach*, 1st ed. MA, USA: Morgan Kaufmann, 2010.

[15] NVIDIA corporation,*“NVIDIA CUDA C programming guide”*, version 3.0, CA, USA: NVIDIA, 2010.

[16] V. Volkov, J. Demmel, “Benchmarking GPUs to tune dense linear algebra”, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, NJ, USA, 2008.

[17] K. Fatahalian, J. Sugerman, P. Hanrahan, “Understanding the efficiency of GPU algorithms for matrix-matrix multiplication”, *In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS ’04)* , ACM, NY, USA, pp. 133-137, 2004