ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **15**, Number 1, 2012, 5–20

Improving GPU Simulations of Spiking Neural P Systems

Francis George C. CABARLE[1], Henry N. ADORNA[1],
Miguel A. MARTÍNEZ-DEL-AMOR[2], Mario J. PÉREZ-JIMÉNEZ[2]

[1] Algorithms and Complexity lab
Department of Computer Science
University of the Philippines Diliman
E-mail: fccabarle@up.edu.ph, ha@dcs.upd.edu.ph

[2] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville, Spain
E-mail: mdelamor@us.es, marper@us.es

Abstract. In this work we present further extensions and improvements of a Spiking Neural P system (for short, SNP systems) simulator on graphics processing units (for short, GPUs). Using previous results on representing SNP system computations using linear algebra, we analyze and implement a computation simulation algorithm on the GPU. A two-level parallelism is introduced for the computation simulations. We also present a set of benchmark SNP systems to stress test the simulation and show the increased performance obtained using GPUs over conventional CPUs. For a 16 neuron benchmark SNP system with 65536 nondeterministic rule selection choices, we report a 2.31 speedup of the GPU-based simulations over CPU-based simulations.

Key-words: Membrane computing, spiking neural network, Spiking Neural P systems, GPU computing, CUDA

1. Introduction

Membrane Computing uses *P systems* (named after their inventor, Gheorghe Păun) as computing models and was introduced in 1998 [18]. 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 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 by some rules and environmental stimuli, and is written back to memory for use in future processes [10]. Another objective is to be able to solve presumably computationally hard problems (e.g. **NP**-complete) in an efficient way and perhaps even go beyond classical models of computation e.g. Turing machine. A loftier goal is to lay the theoretical foundations for future computations with cells (or specifically, neurons, in the case of SNP systems) as the medium of computation. 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 [13, 20]. Membrane Computing has been inspired by the work on DNA computing 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 [18].

P systems (most variants at least) compute in a nondeterministic and maximally parallel manner, oftentimes requiring exponential space as trade off to solve hard problems in polynomial, often linear, time [18, 21, 26]. 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 as one of the ways to further study them. Since P systems were introduced, many simulators using different parallel devices have been produced [8], including CPU clusters [6], reconfigurable hardware as in FPGAs [17], as well as GPUs [3, 4]. 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 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 computations on the GPU), their architecture which is specifically designed for massively parallel computations, are laid bare to programmers [11]. 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 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, 25]. Parallel computing concepts such as hardware abstraction, scaling, and so on are also handled efficiently by current GPUs [14, 25].

Given that SNP systems have already been represented as matrices [24], simulating them in parallel devices such as GPUs is the next natural step. Matrix algorithms are well known in parallel computing literature, including GPUs [9, 23], 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 [1, 2]. In this work we further extend and improve the performance of the previous simulators. We give definitions

and notations for the detailed analysis of the algorithms and simulations. We also create bechmarks to further emphasize the performance increase with GPUs. Limitations of the implementation on GPUs will be pointed out, as well as work for further research.

This paper is organized as follows: Section 2 provides the preliminaries needed for the simulation algorithm and implementation. SNP systems and their matrix representations are formally defined, together with some notions and notations from formal language theory used throughout the paper. Section 3 presents the simulation algorithm and the algorithm analysis. Section 4 presents the benchmark SNP systems used to stress test the simulation. The test hardware setup are also presented in Section 4, together with the results of simulating the benchmark SNP systems. Lastly, we provide conclusions and our future work.

2. Preliminaries

It is assumed that the readers are familiar with the basics of Membrane Computing (a good introduction can be found in [18, 21] while recent online results and information can be found in [26]) and formal language theory (widely available in print and online, e.g. [22]). We only briefly mention notions and notations which will be useful throughout the paper, as was done in the seminal paper for SNP systems [12].

Let V be an alphabet, V^* is the free monoid over V with respect to concatenation and the identity element λ (the empty string). The set of all non-empty strings over V is denoted as V^+ so $V^+ = V^* - \{\lambda\}$. We call V a *singleton* if $V = \{a\}$ and simply write a^* and a^+ instead of $\{a^*\}$ and $\{a^+\}$. The length of a string $w \in V^*$ is denoted by $|w|$.

Regular languages can be defined (among others) by *regular expressions*. A regular expression over an alphabet V is constructed starting from λ and the symbols of V using the operations union, concatenation, and Kleene $+$, using parentheses when necessary to specify the order of operations. Specifically, (i) λ and each $a \in V$ are regular expressions, (ii) if E_1 and E_2 are regular expressions over V then $E_1 \cup E_2$, E_1E_2, and E_1^+ are regular expressions over V, and (iii) nothing else is a regular expression over V. With each expression E we associate a language $L(E)$ defined in the following way: (i) $L(\lambda) = \{\lambda\}$ and $L(a) = \{a\}$ for all $a \in V$, (ii) $L(E_1 \cup E_2) = L(E_1) \cup L(E_2)$, $L(E_1E_2) = L(E_1)L(E_2)$, and $L(E_1^+) = L(E_1)^+$, for all regular expressions E_1, E_2 over V. Unnecessary parentheses are omitted when writing regular expressions, and $E^+ \cup \{\lambda\}$ is written as E^*. A language $L \subseteq V^*$ is regular if there is a regular expression E over V such that $L(E) = L$.

Next we provide the definition of a SNP system from [12, 19].

Definition 1. A *SNP system without delay of degree* $m \geq 1$ is a tuple 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 neurons of the form

$$\sigma_j = (\alpha_j, R_j), 1 \leq j \leq m,$$

where:

(a) $\alpha_j \geq 0$ gives the initial number of spikes contained in neuron σ_j

(b) R_j 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 $c \geq p \geq 1$;

(b-2) $a^s \rightarrow \lambda$, are known as *forgetting rules*, for $s \geq 1$, such that for each rule $E/a^c \rightarrow a^p$ of the previous type from R_j, $a^s \notin L(E)$.

3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$, $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses* among neurons).

4. $in, out \in \{1, 2, \ldots, m\}$ are the indices of 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 $k \geq c$. The regular expression E therefore covers *exactly* the number of spikes in σ_i. Using this type of rule consumes c spikes from the neuron, producing p spikes which are sent to each of the neuron/s connected to σ_i via a synapse in syn. 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. Whenever $E = a^c$ we write (b-1) in shorthand notation as $a^c \rightarrow a^p$.

An *instantaneous description* or a *configuration* at any instant t of a SN P system is described by the number of spikes in each neuron at t. The *initial configuration* is described by the number of spikes initially placed in each neuron, $\alpha_1, \alpha_2, \ldots, \alpha_m$. A configuration is a *halting configuration* if no rule of the system can be applied anymore. Using the rules described above, one can define *transitions* among configurations. We say that configuration C_1 yields configuration C_2 in one *transition step*, denoted by $C_1 \Rightarrow_\Pi C_2$, if we can pass from C_1 to C_2 by applying the rules from the system following the previous remarks.

A *computation* of Π is a (finite or infinite) sequence of configurations such that:

1. The first term of the sequence is the initial configuration of the system;

2. Each non-initial configuration of the sequence is obtained from the previous configuration by a transition step; and

3. If the sequence is finite (called *halting computation*) then the last term of the sequence is a halting configuration.

A computation in a system as above starts at the initial configuration. For *accepting* SNP systems one way to interpret a computation is as follows: In order

to compute a function $f : \mathbf{N}^k \to \mathbf{N}$ we introduce k natural numbers $q_1, \ldots, q_k$ in the system using σ_{in}, by "reading" from the environment a binary sequence $z = 10^{q_1-1}10^{q_2-1}1 \ldots 10^{q_k-1}1$. This means that the input neuron of Π receives a spike in each step corresponding to a digit 1 from the string z and no spike otherwise. Note that we input exactly $k+1$ spikes, *i.e.*, after the last spike we assume that no further spike is coming to the input neuron.

One way to interpret a result of a computation for a *generative* SNP system (*i.e.* one that produces spikes to the environment: if σ_{out} spikes, the spike is sent to the environment) is to check the time difference between the first spike at time t and the second one at time $t+k$. In this sense we say that the system computes the value k. Another way is to treat, with respect to the output neuron, a time when a spike is produced as '1' and a time when no spike is produced as '0', effectively producing a binary spike train of spikes.

The neurons in an SNP system operate in parallel and synchronously, under a *global clock* [12]. However, only one rule can be applied at a given time in each neuron [12, 24]. The *nondeterminism* of SNP systems comes with this fact: if two spiking (or forgetting) rules exist in σ_i, each with regular expressions E_1 and E_2 and that both rules are applicable as well as $L(E_1) \cap L(E_2) \neq \emptyset$ then the neuron has to nondeterministically choose one rule to apply.

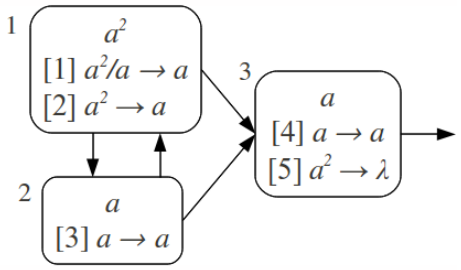


Fig. 1. An SNP system Π_1, generating all numbers in the set $\mathbb{N} - \{1\}$, from [24].

The SNP system shown in Fig. 1 generates all numbers in the set $\mathbb{N} - \{1\}$. The *output* of the computation of the system is the time difference between the first spike produced by the output neuron (to the environment) and the succeeding spikes the output neuron produces. A total system ordering is given to neurons (from (1) to (3)) and rules (from (1) to (5)) of the system in Fig. 1. This SNP system works in a generative way, and it is formally defined as $\Pi_1 = (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, out)$, where:

$\sigma_1 = (2, R_1)$, $\alpha_1 = 2$, $R_1 = \{a^2/a \to a, a^2 \to a\}$,
$\sigma_2 = (1, R_2)$, $\alpha_2 = 1$, $R_2 = \{a \to a\}$,
$\sigma_3 = (1, R_3)$, $\alpha_3 = 1$, $R_3 = \{a \to a, a^2 \to \lambda\}$,
$syn = \{(1,2), (1,3), (2,1), (2,3)\}$,
$out = 3$,
The system has no input neuron.

Now we proceed to represent SNP systems and their computations using matrices. In [24], a matrix representation of SNP systems without delays was introduced. Let

us assume a total order on the rules of the system $R_1 = \{r_{1,1}, \ldots, r_{1,t_1}\}$; $R_2 = \{r_{2,1}, \ldots, r_{2,t_2}\}$; $\ldots$ $R_m = \{r_{m,1}, \ldots, r_{m,t_m}\}$. We have the following definitions.

Definition 2. A *configuration vector* $C_k =< \alpha_1, ..., \alpha_m >$, $\alpha_j \in \mathbb{N}$ for $1 \leq j \leq m$, is a vector containing the number of spikes in every neuron corresponding to the *kth* computation step. C_0 is an initial configuration vector containing the number of spikes in the system at the beginning of the computation.

Definition 3. Given a configuration vector C_k, a *spiking vector* associated with C_k is $S_k =< S_{k_{(1)}}, ..., S_{k_{(n)}} >$, such that

1. for each i $(1 \leq i \leq n)$, $S_{k_{(i)}} = 1$ whenever rule r_i is applicable to C_k, otherwise $S_{k_{(i)}} = 0$;
2. if $\{i_1, \ldots, i_t\}$ verifies $S_{k_{(i_j)}} = 1$ $(1 \leq j \leq t)$ and $S_{k_{(p)}} = 0$ for $p \notin \{i_1, \ldots, i_t\}$, then rules $r_{i_1}, \ldots, r_{i_t}$ are simultaneously applicable to C_k.

For the SNP system in Fig. 1 we have $C_0 =< 2, 1, 1 >$. The initial number of spikes of σ_i is the ith value of the initial configuration vector C_0. With respect to C_0 we have the spiking vector $S_0 =< 1, 0, 1, 1, 0 >$ (if we choose to apply rule (1) and not rule (2)) or $S'_0 =< 0, 1, 1, 1, 0 >$ (we choose to apply rule (2) and not rule (1)). Note that we can have more than one applicable rule in a neuron with respect to a given configuration C_k at the kth step but we choose only one rule to apply. Hence for the SNP system in Fig. 1, $S''_0 =< 1, 1, 1, 1, 0 >$ is an *invalid* spiking vector with respect to C_0.

Definition 4. *Spiking transition matrix* M_Π is an $n \times m$ matrix consisting of elements a_{ij} given as

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

From Definition 4, rows represent rules $(1 \leq i \leq n)$ and columns represent neurons $(1 \leq j \leq m)$. For Π_1 in Fig. 1 the matrix representation M_{Π_1} is as follows:

$$M_{\Pi_1} = \begin{pmatrix} -1 & 1 & 1 \\ -2 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & -2 \end{pmatrix} \tag{1}$$

Finally, the following equation provides the configuration vector at the $(k+1)$th step, given the configuration vector (C_k), the spiking vector (S_k) at the kth step, and the matrix representation (M_Π):

$$C_{k+1} = C_k \ + \ S_k \cdot M_\Pi \tag{2}$$

3. SNP systems simulation algorithm

Now we provide additional notations and definitions for the simulation algorithm. The algorithm is then presented and described. We will take into consideration the parallel implementation of the algorithm in the succeeding section.

Let Π be an SNP system with m neurons and n rules. Recall from Definition 1 that R_j is the set of rules for neuron σ_j, $1 \leq j \leq m$. Recall also that for the matrix representation we define a total ordering of all n rules from $r_1, ..., r_n$ so that we can write $R_j = \{r_{(j,1)}, ..., r_{(j,\beta_j)}\}$ for $(j,1) < (j,2) < \ldots < (j,\beta_j)$, and $\beta_j = |R_j|$. We can label the union of all R_j sets as $R = \bigcup_{j=1}^{m} R_j$. The general expression for a rule (whether spiking or forgetting) is $E/a^c \rightarrow a^p$ a rule $r_i \in R_j$ can be split into three parts: the regular expression E, the consumed number of spikes c, and the produced spikes p. Note that for a forgetting rule (Definition 1), $E = a^c$, $c = s$, and a^p for $p = 0$ ($a^0 = \lambda$). Hence a rule r_i can be written as $r_i =< E, c, p >$ where $r_{i_1} = E$, $r_{i_2} = c$, and $r_{i_3} = p$.

Also note that the n number of rules in an SNP system of degree m can be expressed as $n = \sum_{j=1}^{m} |R_j|$.

In order to simulate the computation of an SNP system Π the simulation algorithm must be initially provided with C_0, M_Π and R as the three initial inputs to the simulation algorithm. We now provide additional definitions for SNP system computations:

Definition 5. We say that $C' = C_k$ is a *reachable configuration* from C_0 after k computational steps if there is a sequence of configuration vectors C_0, C_1, ..., C_{k-2}, C_{k-1} and spiking vectors S_0, S_1, ..., S_{k-2}, S_{k-1} leading to C_k. We simply write $C_0 \overset{k}{\Rightarrow} C_k$.

Definition 6. The set $\overline{S}(C_k)$ is the set of all spiking vectors for a given C_k. The set $S = \bigcup_{C_0 \overset{k}{\Rightarrow} C_k} \overline{S}(C_k)$, $k \geq 0$, is the set of all spiking vectors for every reachable C_k from C_0.

Definition 7. The set $\overline{C}(C_{k+1})$ is the set of all C_{k+1} given C_k and $\overline{S}(C_k)$. The set $C = \bigcup_{C_0 \overset{k}{\Rightarrow} C_k} \overline{C}(C_{k+1})$, $k \geq 0$, is the set of all reachable configuration vector C_k from C_0.

From the first three inputs (*i.e.* C_0, M, R) we can obtain the sets S and C. Using Equation (2) we can determine the next configuration vector C_{k+1} and so the simulation of Π must be able to produce all reachable C_{k+1} given a C_0.

Given a set of rules R and a C_k at the kth step, recall that we only choose one among several applicable (due to nondeterminism) spiking or forgetting rules in a neuron. We can compute q, the maximum number of nondeterministic choices, and hence the maximum number of spiking vectors for any computational step of an SNP system using Lemma 1. The number q will be used in the succeeding section for the computation of $q_{k+1} \leq q$ number of C_{k+1} configurations in parallel.

Lemma 1. *The maximum number of nondeterministic choices at any step for SNP system* Π *is given by*

$$q = \prod_{j=1}^{m} max(\#_{spik}(R_j), \#_{forg}(R_j)) \tag{3}$$

where $\#_{spik}(R_j)$ *and* $\#_{forg}(R_j)$ *denote the number of spiking and forgetting rules, respectively, in the set* R_i *of neuron* σ_j*.*

Proof. Recall that whenever a spiking rule is applicable a forgetting rule cannot be applicable, and vice versa (Definition 1). Additionally, it is possible to have more than one applicable spiking (or forgetting) rule for a given C_k. In order to obtain q we need to obtain the larger value between $\#_{spik}(R_j)$ and $\#_{forg}(R_j)$ of R_j since we are interested in the *maximum number* of nondeterministic choices that could be made between rules of Π assuming *all* spiking (or forgetting) rules are applicable at step k. For a neuron σ_j what determines the maximum number of rules σ_j has to nondeterministically choose from is whether R_j has more spiking or forgetting rules. We denote as q_k the nondeterministic rule choices at step k. We denote by q_{j_k} the number of nondeterministic choices for σ_j at step k, and we multiply each q_{j_k} for all m neurons to obtain q_k so that for any step k, $q_k \leq q$.

□

Observation 1. *For any* C_k *such that* $C_0 \overset{k}{\Rightarrow} C_k$, $|\overline{C}(C_{k+1})| = |\overline{S}(C_k)| \leq q$.

Now we present and describe Algorithm 1 to simulate an SNP system Π's computations. First we define two stopping criteria in the simulation of an SNP system's computation. We halt the simulation once one of the two *stopping criteria* are satisfied: (1) all unique configuration vectors have been produced, or (2) a counter variable $\#_{C_k}$ to count the number of C_k's to produce for a given run of the simulation to set the simulation limit.

Algorithm 1 Overview of SNP system simulation algorithm. **SEQ** and **PAR** indicate which step is done sequentially or in parallel, respectively.

Require: Inputs: C_0, M_Π, R, $\#_{C_k}$ of SNP system Π for $1 \leq i \leq n$, $1 \leq j \leq m$.

0. **(SEQ)** Load inputs: M, R, and C_0 are loaded once only;
I. **(SEQ)** Load every C_{k+1} afterwards for $k \geq 0$;
II. **(SEQ)** With respect to current C_k, determine if a rule $r_i \in R_j$ is applicable by checking if $a^{\alpha_j} \in L(E)$ for E associated with the rule r_{i_1}, $\alpha_j \in C_k$, $R_j \in \sigma_j$. Then generate all possible S_k from all applicable rules;
III. **(PAR)** Produce all C_{k+1} from all possible S_k with respect to current C_k;
IV. **(SEQ)** Repeat steps I to IV, till all unique C_k are produced (stopping criterion (1)) or the number of computed C_k is equal to $\#_{C_k}$ (stopping criterion (2));

Algorithm 1 starts by loading the initial inputs C_0, M_Π, R, and a desired $\#_{C_k}$ (step I). In step II, the number of spikes in a neuron σ_j are checked if they satisfy

the regular expression E_i of a rule $r_i \in R_j$. The simulation 'implements' the nondeterminism by producing all S_k's for a given C_k, and proceeds to compute each of the C_{k+1} from these. The simulation essentially does a *breadth-first search* in producing all C_k from a C_0. The set $\overline{C}(C_{k+1})$ is the union of all the configuration vectors at the $(k+1)$th level of the C_k tree. The set C is the union of all the configuration vectors of the configuration tree where C_0 is the root node.

The process by which all possible and applicable S_k's are produced is as follows: Once all the rules in the system are identified, given the current α_j's (number of spikes present in each of the σ_j's), the {*1,0*} strings (at the moment they are treated as strings, and then as integral values later on during the computation of Equation (2)) are produced on a per neuron level. As an example, given C_0 for Π_1 and $\alpha_1 = 2$ in Figure **??**, and σ_1 has two rules r_1 and r_2, we have the neuron-level strings *'10'* (we choose to use r_1 instead of r_2) and *'01'* (use r_2 instead of r_1). For σ_2 we only have $\alpha_2 = 1$ (r_3 of σ_2 has the needed single spike in σ_2, and it has only one rule) while σ_3 gives us *'10'* since its single spike enables r_4 only and not r_5. After producing the neuron-level {*1,0*} strings, the strings are exhaustively paired up and concatenated with each other, from left (first neuron) to right (the last neuron, since there is a need for ordering), until finally all the applicable S_k's (i.e. $(1,0,1,1,0)$ and $(0,1,1,1,0)$) from the current C_k are produced.

Step III performs Equation (2) in parallel. The *strings* (for the purposes of concatenation, regular expression checking, among others) are now treated as integral values. Step IV then checks whether to proceed or to stop based on 2 *stopping criteria* for the simulation given the produced C_{k+1}.

Lemma 2. *Algorithm 1, for an SNP system* Π *with* m *neurons and* n *rules, computes the sets* C *and* S*.*

Proof. We have as input an SNP system Π with finitely many neurons and rules. Step II checks all $|R| = n$ rules for applicability. At a computational step k, rule $r_i \in R$, $1 \leq i \leq n$, is applicable if $a^{\alpha_j} \in L(E)$ and $\alpha_j \geq c$, where $c = r_{i2}$, $E = r_{i1}$ and α_j is the number of spikes for neuron j in each C_k. Once all applicable rules are known, step II is performed, which from Lemma 1 produces $q_k \leq q$ applicable spiking vectors. Step II then produces the set $\overline{S}(C_k)$. Step III performs Equation (2) over the q_k elements of $\overline{S}(C_k)$ producing q_k number of configuration vectors added to the set $\overline{C}(C_{k+1})$. In step II we add the elements of $\overline{S}(C_k)$ to S (S is initially set to $\emptyset$) and the elements of $\overline{C}(C_{k+1})$ to C (C is initially set to $\emptyset$) by the union operation. We then check if one of the stopping criteria is satisfied to make sure that Algorithm 1 halts. If not one of the stopping criteria is satisfied, we repeat from step I. Once one of the stopping criteria is satisfied Algorithm 1 halts, in effect we have traversed each computation branch of the C_k tree of Π starting from C_0 as root of the tree (where the branches are due to nondeterminism) and we produce the sets S and C.

□

Theorem 1. *Algorithm 1 simulates the computation of an SNP system* Π*.*

Proof. The proof follows directly from Lemma 2. □

4. SNP system simulations in GPUs

NVIDIA, a 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 [25]. 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 [25].

GPUs introduce increased performance speedups over CPU only implementations with 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] (see Fig. 2 for an illustration). 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. However if the problem to be solved cannot be organized in a data parallel form (a task performing computations on data need not depend heavily on other task's results) then the performance of GPUs over CPUs will not be fully utilized.

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) acts as *co-processor* 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/GPU. Each device has multiple cores, each capable of running its own threads. Each core in the device is able to run a set of threads (8 blocks/core $\times$ 4 threads/block equal to 32 threads per core which is known as the *warp size*). A thread block is assigned to each multiprocessor, where each processor is made up of several cores [14, 25]. 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) in a thread block, and the thread blocks (from one to two dimensions) in a grid. Table 1 shows the resources of current CUDA enabled NVIDIA GPUs.

From Algorithm 1, step III is executed in the device and done in parallel. At step III there are two levels of parallelism. The first level is where C_{k+1} is computed in parallel using Equation (2). The second level of parallelism comes from computing all q_k number of C_{k+1_i} (ith configuration at $(k+1)$th step) given M, C_k, and S_{ki}

(ith spiking vector at kth step), for $1 \leq i \leq q_k$. The parallel implementation is shown graphically in Fig. 3. From Fig. 3 the length of the linear array containing all possible spiking vectors S_k is $q \cdot n$ because each spiking vector is of length $|R| = n$ (the number of rules in the system). Note that at this point we use q and q_k interchangeably, since we are testing for the maximum limit of the algorithm and the GPUs. Similarly, from the $q \cdot n$ number of spiking vectors we will obtain $q \cdot m$ (we have m neurons in the system) number of C_{k+1}, which is the length of the linear array of all C_{k+1} in Fig. 3. We then launch $q \cdot m$ number of threads, distributed in q thread blocks each having m threads.

Table 1. Typical resources for CUDA enabled GPUs (from [14, 25])

GPU resources	**Values**
Global memory	Depends on GPU model
Max number of threads per dimension (x, y, z)	$(512, 512, 64)$
Max number of thread blocks per grid (x, y, z)	$(65535, 65535, 1)$

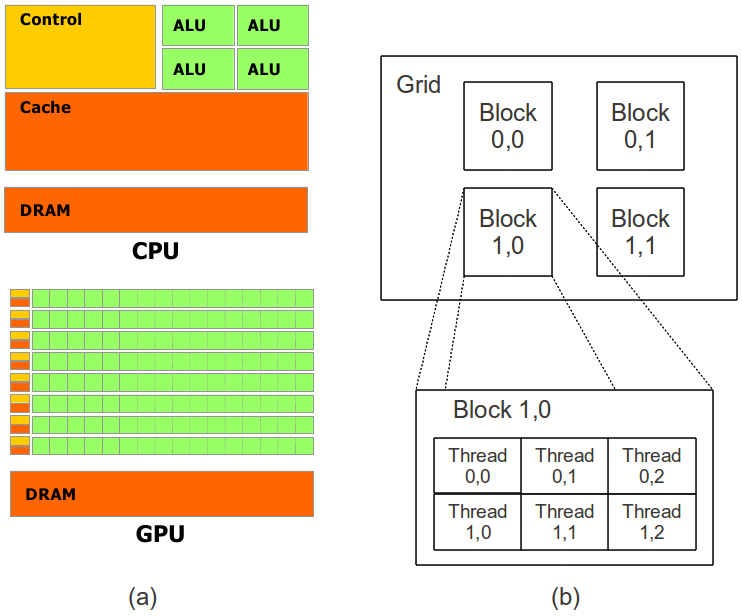


Fig. 2. (a) Common transistor allocation of CPUs and GPUs; (b) Computing unit hierarchy of GPUs, from [25] and [7].

Observation 2. *For an SNP system* Π *with* m *neurons and* n *rules, if* $n < m$ *then* Π *has neurons without rules and hence can be ignored without affecting the system since these neurons only act as spike repositories.*

Observation 3. *For an SNP system* Π *with* m *neurons and* n *rules, if* $m = n$*, then* Π *is a deterministic system.*

From Observations 3 and 2 we base the reason for launching $m \cdot q$ number of threads since n will be at least m in value. A thread block therefore in Fig. 3 is a linear arrangement of threads. Similarly, the grid is a linear arrangement of thread blocks.

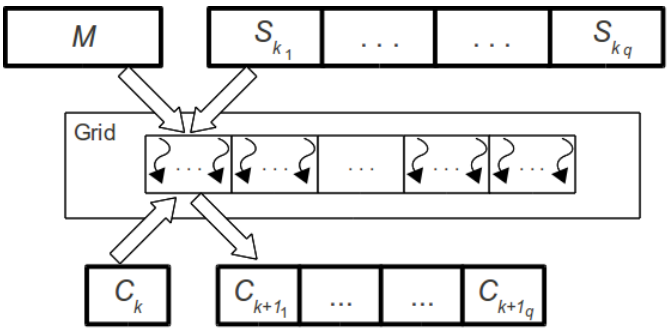


Fig. 3. Parallel computation of next configuration C_{k+1} given q number of nondeterministic choices: Blocks with thicker borders represent memory allocation in the GPU. The figure shows a single instance of the parallel memory access and computation of Equation (2). A curvy arrow represents a thread (ranging from 0 to $m-1$) in a thread block (ranging from 0 to $q-1$). Here $|M| = m \times n$, $|S_{kl}| = |R|$, $|C_k| = |C_{k+1_l}| = m$ for $1 \leq l \leq q$, and q is from Equation (3).

Based on Algorithm 1 and the implementation using *PyCUDA* (a Python wrapper for CUDA, see [15]) and CUDA C, the amount of device global memory (simply, G) used is calculated (in terms of m, n, and q_k) to be :

$$G \geq q_k(n + 2m) + m(n + 1) \tag{4}$$

Equation (4) was derived by expressing the computation of Equation (2) (performed q_k times) in terms of m, and n, including another $q_k \cdot m$ linear array S_kM to temporarily hold the products of all the spiking vectors to M. M is only loaded onto the device once. At step III, q_k number of thread blocks are launched, each containing m threads. A thread block computes Equation (2) in parallel with respect to the $q-1$ other thread blocks. A thread in a thread block computes a single element of C_{k+1_i} in parallel with respect to the $m-1$ other threads in the same thread block.

In order to "stress test" the parallel implementation of Algorithm 1 we produce *complete graphs* with neurons as nodes and directed edges as synapses. We refer to these SNP systems as *benchmark SNP systems* or simply B_{mx} having 2^x neurons. The number of neurons will be a power of 2 so that for B_{m4} will have $2^4 = 16$ neurons, where each neuron (or node) has an outdegree of 15. Figure 4 shows a B_{m2} where each neuron has two spiking rules (no forgetting rules) of the form $(a^2)^*/a \rightarrow a$ and $(a^2)^*/a^2 \rightarrow a^2$ so that whenever the number of spikes in a neuron is a multiple of 2 then each neuron has to nondeterministically choose between the two rules. In Fig. 4 we see that B_{m2} has a constant *rule per neuron density* $d_{r/m} = 2$ so that for B_{m2} we can calculate q to be equal to $2^4 = 16$. For a benchmark SNP system B_{mx} with $d_{r/m} = 2$, $q = q_k = (d_{r/m})^{2^x}$.

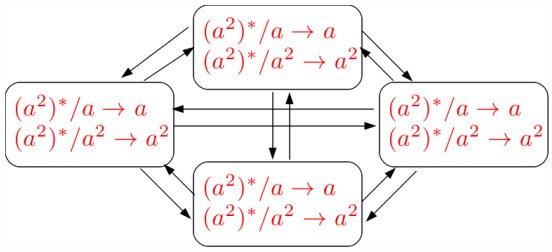


Fig. 4. A benchmark SNP system B_{m2} with $2^2 = 4$ neurons, where neurons are placed on a complete graph with the outdegree of each neuron (node) equal to 3.

Table 2 shows the setup used to run B_{m1} to B_{m4}. From Table 2 we can see that although the GPU has 4GB of memory, its memory is still far less than the CPU. Current GPUs have far more computational units than CPUs but they usually have smaller amount of memory compared to CPUs. For the simulation, once all q_k number of spiking vectors have been produced in the host, we move all the spiking vectors to the device, compute Equation (2) producing q_k number of C_{k+1}, and then moving all the C_{k+1} back to the host from the device. Note that, as with GPU (and oftentimes in parallel computing as well) before moving all q_k number of spiking vectors, $q_k \cdot m$ space has to be allocated at the receiving end (the device, from the host) for the expected q_k number of C_{k+1}.

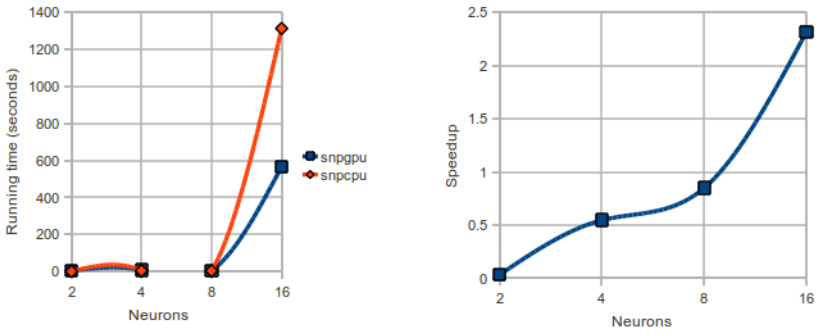
Table 2. Setup used to run the parallel implementation of Algorithm 1 for B_{m1} to B_{m4}

System specifications	**Values**
Processors	2 × Intel Xeon E554 @ 2GHz
Cores/processor	4
Host RAM	12GB
GPUs	2 × Tesla C1060 @ 1.3GHz
CUDA Cores/GPU	30 (multiprocessors) × 8 (cores/processor) = 240
Device global memory	4GB
Operating system	Ubuntu 10.04 server edition (64bit)
CUDA version	3.2

Table 3. Table showing the number of neurons and values of $q = q_k$ for B_{m1} to B_{m4}

Benchmark SNP system ($d_{r/m} = 2$)	**neurons**	$q = q_k$
B_{m1}	2	4
B_{m2}	4	16
B_{m3}	8	256
B_{m4}	16	65536

A CPU only version of Algorithm 1 is created so that step III of the algorithm is done sequentially and we designate this as *snpcpu-sim* while the CPU-GPU simulator is designated as *snpgpu-sim*.



(a) (b)

Fig. 5. (a) The running time of *snpcpu* versus *snpgpu* for B_{m1} to B_{m4}; (b) The speedup ($snpcpu_{runtime}/snpgpu_{runtime}$) is up to 2.31 .

Figure 5 shows the running times for both *snpcpu-sim* and *snpgpu-sim* simulators regarding the parameter m (number of neurons). As shown in Table 3, q and m are exponentially increased, and so are the number of threads ($q \cdot m$) and amount of device global memory (Equation 4). The larger the number of neurons, the resources in the GPU are better fulfilled. Therefore, we report the maximum value of speedup for 16 neurons in our benchmark, which is 2.31x.

5. Conclusions and future work

We've presented the simulation (theoretical and practical) details of an SNP system simulator. The simulation algorithm was shown to simulate the computation of an SNP system given initial configurations and inputs. The algorithm was implemented on a CPU-GPU setup and was tested using "stress test" benchmarking SNP systems which are complete graphs where a node is a neuron. The neurons in the benchmark SNP systems had a fixed rule per neuron density $d_{r/m}$ and were simulated from 2 ($q = 4$) neurons up to 16 neurons ($q = 2^{16} = 65536$). The sequential implementation of Algorithm 1 (*snpcpu*) as well as the parallelized CPU-GPU version of Algorithm 1 (*snpgpu*) were run on a test setup. The speedup for *snpgpu* given the benchmark SNP systems as inputs was found to be up to 2.31 times (over *snpcpu*). The running time for *snpgpu-sim* is increased (thus lowering the speedup) because of the continuous transfer of data (between host and device) for each C_k in step III from Algorithm 1.

For our future work, further tests for *snpgpu* will be performed for stress testing, such as benchmark SNP systems with a fixed m (number of neurons) but where the $d_{r/m}$ increases, thus effectively increasing the number of rules as well as the nondeterministic rule choices (q). We plan to include other steps of Algorithm 1 in the GPU, avoiding host-device data transfers as much as possible in order to increase performance. Running the simulator for a multi-GPU setup is also one of the future works. Additionally, combining the GPU simulator for SNP systems with *P-Lingua*, a CPU-based framework for simulating P systems written in Java by the Natural

Computing research group from Sevilla is also a future work [16].

Acknowledgements. Francis George C. Cabarle is supported by the Engineering Research and Development for Technology (ERDT) program of the Department of Science and Technology (DOST) of the Philippines. Henry N. 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 and Mario J. Pérez–Jiménez are 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 Ciencia e Innovación" of Spain, both co-financed by FEDER funds.

References

[1] Cabarle F.G.C., Adorna H., Martínez-del-Amor, M.A., *Simulating spiking neural P systems without delays using GPUs*, International Journal of Natural Computing Research, **2**, 2 (2011), pp. 19–31.

[2] Cabarle F.G.C., Adorna H., Martínez-del-Amor M.A., *An Improved GPU Simulator For Spiking Neural P Systems*, Proceedings of the IEEE Sixth International Conference on Bio-Inspired Computing: Theories and Applications, Penang, Malaysia, 2011, pp. 262–267.

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

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

[5] Chen H., Ionescu M., Ishdorj T.-O., Păun A., Păun Gh., Pérez-Jiménez M., *Spiking neural P systems with extended rules: universality and languages*, Natural Computing, 7, 2 (2008), pp. 147–166.

[6] Ciobanu G., Wenyuan G., *P Systems Running on a Cluster of Computers*, Workshop on Membrane Computing 2003 (C. Martín-Vide et al., eds.), Lecture Notes in Computer Science, 2933, Springer, Berlin, 2004, pp. 123–139.

[7] Clemente J., Cabarle F.G.C., Adorna H., *PROJECTION Algorithm for Motif Finding on GPUs*, Workshop on Computation: Theory and Practice 2011, In Proceedings of Information and Communications Technology (J. Caro et al., eds.), Springer, 2012, in press.

[8] Díaz D., Graciani C., Gutiérrez M.A., Pérez-Hurtado I., Pérez-Jiménez M.J., *Software for P systems*, The Oxford Handbook of Membrane Computing (Gh. Păun et al., eds.), Oxford University Press, Oxford (U.K.), Chapter 17, 2009, pp. 437–454.

[9] Fatahalian K., Sugerman J., Hanrahan P., *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, 2004, pp. 133–137.

[10] Gross M., *Molecular computation*, Non-Standard Computation (T. Gramss et al., eds.), Chapter 2, Wiley-VCH, Weinheim, 1998.

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

[12] Ionescu M., Păun Gh., Yokomori T., *Spiking Neural P Systems*, Fundamenta Informaticae, **71**, 2-3 (2006), pp. 279–308.

[13] Kari L., Rozenberg G., *The many facets of natural computing*, Communications of the ACM, **51**, 10 (2008), pp. 72–83.

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

[15] Klöckner A., Pinto N., Lee Y., Catanzaro B., Ivanov P., Fasih A., *PyCUDA: GPU Run-Time code generation for High-Performance Computing*, Scientific Computing Group, Brown University, no. 2009–40, RI, USA, 2009

[16] Macías-Ramos L.F., Pérez-Hurtado I., García-Quismondo M., Valencia-Cabrera L., Péerez-Jiménez M.J., Riscos-Núñez A., *A PLingua based Simulator for Spiking Neural P Systems*, 12th Int'l Conference on Membrane Computing 2011 (M. Gheorghe et al., eds.), Lecture Notes in Computer Science, 7184, Springer, Berlin, 2012, pp. 257–281.

[17] Nguyen V., Kearney D., Gioiosa G., *A Region-Oriented Hardware Implementation for Membrane Computing Applications*, 10th Workshop on Membrane Computing 2009 (Gh. Păun et al., eds.), Lecture Notes in Computer Science, 5957, Springer, Berlin, 2010, pp. 385–409.

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

[19] Păun Gh., *Spiking Neural P Systems: A Tutorial*, Journal Bulletin of the European Association for Theoretical Computer Science, **91**, 2007, pp. 145–159.

[20] Păun Gh., *From Cells to (Silicon) Computers, and Back*, New Computational Paradigms: Changing Conceptions of What Is Computable, Springer, 2008, pp. 343–371.

[21] Păun Gh., Rozenberg G., Salomaa A. (eds.), *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.

[22] Shallit J., *A Second Course in Formal Languages and Automata Theory*, Cambridge University Press, 2008.

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

[24] Zeng X., Adorna H., Martínez-del-Amor M.A., Pan L., Pérez-Jiménez M., *Matrix Representation of Spiking Neural P Systems*, 11th International Conference on Membrane Computing 2010 (M. Gheorghe et al., eds.), Lecture Notes in Computer Science, 6501, Springer, 2011, pp. 377–391.

[25] NVIDIA corporation, *"NVIDIA CUDA C programming guide"*, version 3.2, CA, USA, 2010.

[26] P systems resource website. (2011, Nov) [Online].
Available: `http://ppage.psystems.eu/`.