Improving Simulations of Spiking Neural P Systems in NVIDIA CUDA GPUs: CuSNP

[1]Jym Paul Carandang, [1]John Matthew B. Villaflores, [1]Francis George C. Cabarle, [1]Henry N. Adorna, [2]Miguel Ángel Martínez-del-Amor

[1]Algorithms and Complexity
Department of Computer Science
University of the Philippines Diliman
Diliman 1101 Quezon City, Philippines;
email: jacarandang@gmail.com, matthewvillaflores@gmail.com, fccabarle@up.edu.ph
hnadorna@up.edu.ph
[2]Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: mdelamor@us.es

Summary. Spiking neural P systems (in short, SN P systems) are parallel models of computations inspired by the spiking (firing) of biological neurons. In SN P systems, neurons function as spike processors and are placed on nodes of a directed graph. Synapses, the connections between neurons, are represented by arcs or directed endges in the graph. Not only do SN P systems have parallel semantics (i.e. neurons operate in parallel), but their structure as directed graphs allow them to be represented as vectors or matrices. Such representations allow the use of linear algebra operations for simulating the evolution of the system configurations, i.e. computations. In this work, we continue the implementations of SN P systems with delays, i.e. a delay is associated with the sending of a spike from a neuron to its neighbouring neurons. Our implementation is based on a modified representation of SN P systems as vectors and matrices for SN P systems without delays. We us massively parallel processors known as graphics processing units (in short, GPUs) from NVIDIA. For experimental validation, we use SN P systems implementing generalized sorting networks. We report a speedup, i.e. the ratio between the running time of the sequential over the parallel simulator, of up to approximately 51 times for a 512-size input to the sorting network.

Key words: Membrane computing; Spiking neural P system; NVIDIA CUDA; graphics processing units

1 Introduction

Membrane computing, initiated in [15], involves models of computations inspired by structures and functions of various types of biological cells. Models in membrane computing are known as membrane or P systems. The specific type of P system we consider in this work are spiking neural P systems, in short, SN P systems. SN P systems, first introduced in [7], are inspired by the pulse coding of information that occur in biological neurons. In pulse coding from neuroscience, pulses known as *spikes* are indistinct, so information is instead encoded in their multiplicity or the time step(s) they are emitted.

SN P systems are known to be computationally universal (i.e. equivalent to Turing machines) in both generative (an output is given, but not an input) and accepting (an input is given, but not an output) modes. SN P systems can also solve hard problems in feasible (polynomial to constant) time. Another active line of investigation on the computability and complexity of SN P systems is taking mathematical and biological inspirations in order to create new variants, e.g. asynchronous operation, weighted synapses, rules on synapses, structural plasticity. We do not go into details, and we refer to [7, 9, 14, 17, 5] and references therein.

Software simulators for P systems, whether sequential or parallel, have been provided. Sequential simulators include for example those implemented using PLingua, a programming language designed for P systems, e.g. [11]. Simulators using massively parallel processors known as graphics processing units (in short, GPUs) for cell-like P systems as well as SN P systems include [13], a comprehensive survey in [12], and [4, 10].

In this work, we report our ongoing efforts to simulate SN P systems on GPUs manufactured by NVIDIA. In particular, our contributions in this report are as follows: (a) modified matrix representation of [18] in order to be able to simulate SNP systems with delays, (b) the entire simulation of SN P systems with delays is now performed in the GPU, compared to a small portion of the simulation in [4], and (c) using generalized sorting network of SN P systems, we report up to 51 times speedup in our experiments with a 512 input size network. A preliminary version of this work is available in [6].

This work is organized as follows: Section 2 provides preliminaries for the remainder of this work; Section 3 provides the definition of SN P systems as well as their linear algebra representations; Section 4 provides an overview of the NVIDIA CUDA architecture; Section 5 provides the simulation algorithm for our work; Section 6 provides experimental results for the sequential and parallel simulators; Finally, Section 7 provides conclusions from our work as well as future research directions.

2 Preliminaries

We recall some formal language theory (available in many monographs). We only briefly mention notions and notations which will be useful throughout the paper.

We denote the set of natural (counting) numbers as $\mathbb{N} = \{0, 1, 2, \ldots\}$, where $\mathbb{N}^+ = \mathbb{N} - \{0\}$. Let V be an alphabet, V^* is the set of all *finite* strings 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 $V^+ = V^* - \{\lambda\}$.

A language $L \subseteq V^*$ is *regular* if there is a regular expression E over V such that $L(E) = L$. A regular expression over an alphabet V is constructed starting from λ and the symbols of V using the operations union, concatenation, and $+$. 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. If $V = \{a\}$, we simply write a^* and a^+ instead of $\{a\}^*$ and $\{a\}^+$. If $a \in V$, we write $a^0 = \lambda$.

3 Spiking Neural P Systems

We assume some familiarity with membrane computing concepts, widely available online (e.g. [1]) or in print (e.g. [16]). First, we formally define SN P systems, followed by linear algebra representations of their computations.

3.1 Spiking Neural P System

A Spiking Neural P system Π is of the form:

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

1. $O = \{a\}$ is the alphabet containing a single symbole (the spike);
2. $\sigma_1, ..., \sigma_m$ are neurons, of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$ where:
   a) $n_i \geq 0$ is the initial number of spikes contained in σ_i.
   b) R_i is a finite set of rules of the following two forms:
      i. $E/a^c \rightarrow a^p; d$ where E is a regular expression over O and $c \geq p \geq 1, d \geq 0$.
      ii. $a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a^p; d$ of type (i) from R_i, we have $a^s \notin L(E)$;
3. $syn \subseteq \{1, 2, ..., m\}x\{1, 2, ..., m\}$ with $i \neq j$ for all $(i, j) \in syn, 1 \leq I, j \leq m$ (synapses between neurons);
4. $in, out \in \{1, 2, ..., m\}$ indicate the input and the output neurons, respectively.

The rules of type (i) as mentioned in the construct of neurons are firing (or spiking) rules while the type (ii) are called forgetting rules. An SN P system whose firing rules have $p = 1$ is said to be of the standard type (non-extended). Given a spiking rule, it is applied as follows. If a neuron σ_i contains k spikes, and

$a^k \in L(E), k \geq c$, then the rule $E/a^c \rightarrow a^p; d \in R_i$ can be applied. This means we remove c spikes so that $k - c$ spikes remain in σ_i, the neuron is then fired and produces p spikes (1 in the case of standard SN P systems) after d time units.

Spikes are fired after $t + d$ where t is the current time step of the computation. For the case that $d = 0$, the spikes are fired immediately. When the time step of the computation is between t and $t + d$, we say that the neuron σ has not fired the spike yet and σ is closed, meaning it cannot receive spikes from other neuron connected to it. In the case that a neuron with an in-going synapse to σ fires, the spike(s) is(are) lost. During the time step $t+d$, the spikes are fired, and the neuron is now open to receive spikes. At $t + d + 1$ the neuron can begin applying rules. When neuron σ_i emits the spike, the spikes reach immediately all neuron σ_j such that $(i, j) \in syn$ and σ_j is open.

A forgetting rule is applied as follows. If the neuron σ_i contains exactly s spikes, then the rule $a^s \rightarrow \lambda$ from R_i can be applied, meaning all of the s spikes are removed from σ_i.

For rules of the form $E/a^c \rightarrow a^p; d$ of type (1) where $E = a^c$, we write it in the shortened form $a^c \rightarrow a^p; d$. There are cases when two or more rules are applicable in a step of the computation, at these cases, only one rule is applied and is non-deterministically chosen. However, by definition, it is impossible to have a spiking rule and a forgetting rule to be applied at the same time. In short, for each neuron, at most one rule will be applied at a time unit.

A *configuration* or state of the system at time t can be described by $C_t = \langle r_1/k_1, \ldots, r_m/k_m \rangle$ for $1 \leq i \leq m$, where neuron i contains $r_i \geq 0$ spikes and remains closed for k_i more steps. The initial configuration of the system is therefore $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$. Rule application provides us a *transition* from one configuration to another. A computation is any (finite or infinite) sequence of configurations such that: (a) the first term is the initial configuration C_0; (b) for each $n \geq 2$, the nth configuration of the sequence is obtained from the previous configuration in one transition step; and (c) if the sequence is finite (called *halting computation*) then the last term is a *halting configuration*, i.e. a configuration where all neurons are open and no rule can be applied.

Two common ways to interpret output of an SN P system are as follows: (1) obtaining the time interval between exactly the first two steps when the output neuron σ_{out} spikes, e.g. number $n = t_n - t_1$ is computed, where σ_{out} produced its first two spikes at steps t_1 and t_n; (2) counting the number of spikes produced by σ_{out} until the system halts. Note that for (1) the system need not halt, since we only consider the first two steps when σ_{out} spikes. In this work, we consider systems that produce their output using the manner given in (2).

Spiking Neural Systems are usually represented as a directed graphs. Figure 1 shows an example of an SN P system with 3 neurons. This system is formally defined as:

$$\Pi = (\{a\}, \sigma_1, \sigma_2, \sigma_3, \{(1, 2), (2, 1), (1, 3), (3, 1)\}, 1) \text{ where:}$$

1. $\sigma_1 = (0, \{a/a \rightarrow a; 0, a^2/a^2 \rightarrow \lambda\})$

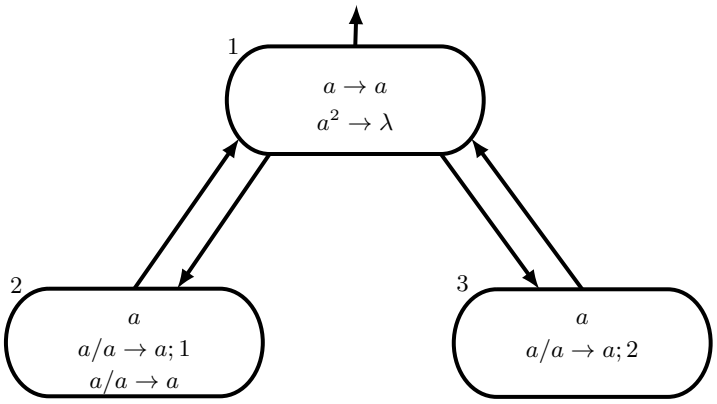

Fig. 1: Example of an SN P System

2. $\sigma_2 = (1, \{a/a \to a; 1, a/a \to a\})$
3. $\sigma_3 = (1, \{a/a \to a; 2\})$

Notice that in σ_1, the rule $a/a \to a; 0$ was written $a \to a$ which is convention when $L(E) = a^p$, we can only write a^p. Also, we do not write the delay d if it is equal to 0. In this example, some rules used this convention while others did not to show the interchangeability between the two.

In Figure 1, there are three neurons, two of which have initial spikes (neurons σ_2 and σ_3). Neurons 1 and 3 have synapses between each other, likewise between σ_1 and σ_2. A synapse exist from σ_1 to the environment to indicate it is the output neuron. In the first step of computation, σ_1 will not fire since it does not contain any spike. Neuron 3 will fire and consume its spike but it will close and will not transmit a spike yet since it has a delay. For σ_2, we non-deterministically choose which rule to apply. Assuming we choose the rule $a/a \to a; 1$, the neuron will consume all of its spikes and similar to σ_3, it will close and will not send any spikes yet.

Therefore, after the first step of computation, there are no spikes in the system. In the next step of computation, no rules will spike since there are no spikes in the system but σ_2 will release a spike since the delay is done. After this step, σ_1 will contain a spike. At the third step of computation, σ_3 will release a spike that was delayed and σ_3 becomes open also. Neuron 1 will send a spike to both σ_2 and σ_3. After this step, each neuron will have one spike. Assuming we always choose to apply the first rule of σ_2, this SN P system will cycle every three steps of computation.

Going back to the first step of computation, assuming we choose to apply the second rule in σ_2, in the next configuration, σ_1 will have one spike while σ_2 and σ_3 will have zero. Neuron 1 will then fire, sending a spike to σ_2. Neuron 3 will not receive a spike since it is currently closed. In the third step of computation, assuming we selected the second rule in σ_2 again, σ_3 will send the delayed spike and

σ_2 will spike immediately. After this step, σ_1 will have two spikes. The forgetting rule will be applied and the computation halts.

3.2 Matrix Representation of Spiking Neural P systems with Delay

In [18], a matrix representation of SN P system without delay was introduced. In this work we introduce modifications to this representation which allows us to devise simulations for SN P systems with delay. Let Π be an SN P system with delay having m neurons and n rules. We use the following definitions, modified from [18], to represent our simulation algorithm:

Definition 1: (Configuration Vector). The vector $C^{(k)} = \langle c_1, c_2, ..., c_m \rangle$ is called the configuration vector at the kth step of computation where each $c_i, i = 1, 2, ..., m$, is the amount of spikes neuron i contains.

Specifically, the vector $C^{(0)} = \langle c_1, c_2, ..., c_m \rangle$ is called the initial configuration vector of Π, where c_i is the amount of the initial spikes present in neuron $\sigma_i, i = 1, 2, ..., m$ before the computation starts.

Definition 2: (Spiking Vector). Let $C^{(k)} = \langle c_1, c_2, ..., c_m \rangle$ be the kth configuration vector of Π. Assume a total order $d : 1, ..., n$ is given for all the n rules, so the rules can be referred to as $s_1, ..., s_n$. A spiking vector $S^{(k)}$ is defined as follows:

$$S^{(k)} = \langle s_1^{(k)}, s_2^{(k)}, ..., s_n^{(k)} \rangle$$

$$s_i^{(k)} \begin{cases} 1, \text{if the regular expression } E_i \text{ of rule } r_i \text{ is satisfied by} \\ \text{the numbers of spikes } c_j \text{ (rule } r_i \text{ is in neuron } \sigma_j \text{) and} \\ \text{rule } r_i \text{ is chosen and applied;} \\ 0, \text{otherwise} \end{cases}$$

Definition 3: (Status Vector). The vector $St^{(k)} = \langle st_1, st_2, ..., st_m \rangle$ is called the status vector at the kth step of computation where each $st_i, i = 1, 2, ..., m$, determines the status of the neuron m.

$$st_i = \begin{cases} 1, \text{if neuron } m \text{ is open} \\ 0, \text{if neuron } m \text{ is closed} \end{cases}$$

Note that a neuron is said to be closed when a rule with a delay is activated and is waiting for that delay to become zero. A neuron that is closed may not receive any incoming spikes.

Definition 4: (Rule Representation). The set $R = \{r_1, r_2, ..., r_n\}$ is the set of rules where each r_i, $i = 1, 2, ..., n$ is a vector representing each rule in Π. Each r_i is defined as follows.

$$r_i = \langle E, j, d', c \rangle$$

where:

1. E is the regular expression for rule i
2. j is the neuron that contains the rule r_i
3. $d' = \begin{cases} -1, \text{if the rule is inactive (i.e. not applied)} \\ 0, \text{if the rule is fired} \\ \geq 1, \text{if the rule is currently on delay (i.e. } \sigma_j \text{ is closed)} \end{cases}$
4. c is the number of spikes that neuron σ_j will consume if it applies r_i.

Definition 5: (Delay Vector). The delay vector $D = \langle d_1, d_2, ..., d_n \rangle$ contains the delay value for each rule $r_i, i = 1, 2, ..., n$ in Π.

Definition 6: (Loss Vector). The vector $LV^{(k)} = \langle lv_1, lv_2, ..., lv_m \rangle$ is the loss vector where each lv_i, for each neuron $\sigma_i, i = 1, 2, ..., m$, contains the number of spikes consumed, c, if σ_i applies r_i at step k.

Definition 7: (Gain Vector). The vector $GV^{(k)} = \langle gv_1, gv_2, ..., gv_m \rangle$ is the gain vector which contains the total number of spikes gained, gv_i, for each neuron $\sigma_i, i = 1, 2, ..., m$, at the kth step of computation not considering whether the neuron is open or closed.

Definition 8: (Transition Vectors). Given the total order $d : 1, ..., n$ for all the n rules, the transition vector Tv of the system Π, is a set of vectors defined as follows:

$$Tv = \langle tv_1, ..., tv_n \rangle$$

$$tv_i = \langle p_1, ..., p_m \rangle$$

$$p_j = \begin{cases} p, \text{ if rule } r_i \text{ is in neuron } \sigma_s (s \neq j \text{ and } (s, j) \in syn) \\ \text{ and it is applied producing } p \text{ spikes;} \\ 0, \text{if rule } r_i \text{ is in neuron } \sigma_s (s \neq j \text{ and } (s, j) \notin syn). \end{cases}$$

The set Tv replaces the spiking transition matrix used in [18] since in [18], each matrix entry can contain values either from spikes consumed or produced by each neuron with respect to rules of other neurons. Transition vectors, however, contain only the p spikes gained from other neurons, otherwise 0.

Definition 9: (Indicator Vector) The indicator vector $IV^k = \langle iv_1, iv_2, ..., iv_m \rangle$ will be multiplied to Transition Vector, $Tv \cdot IV^k$, in order to get the net number of spike a neuron will get not considering a neuron's status.

Definition 10: (Net Gain Vector). Let $LV^{(k)} = \langle lv_1, lv_2, ..., lv_m \rangle$ be the kth loss vector vector, $GV^{(k)} = \langle gv_1, gv_2, ..., gv_m \rangle$ is the kth gain vector vector, and $St^{(k)} = \langle st_1, st_2, ..., st_m \rangle$ is the kth status vector vector. The net gain vector at step k is defined as

$$NG^{(k)} = GV^{(k)} \otimes st^{(k)} + LV^{(k)}$$

4 NVIDIA CUDA

CUDA or Compute Unified Device Architecture is a parallel programming computing platform and application programming interface model developed by NVIDIA

[3]. CUDA allows software developers to use a CUDA enabled graphics processing unit(GPUs) for general purpose processing, an approach known as GPGPU.

Functions that execute in the GPU, known as kernel functions, are executed by one or more threads arranged in thread blocks. In the CUDA programming model, the GPU is often referred to as the device, while the CPU is referred to as the host. The CPU (the host) is the one performing kernel function calls to be executed on the device (the GPU). CUDA works on an SPMD principle or the single program multiple data principle. That is, similar code runs on the threads, and the threads can be accessing multiple (possibly different values of) data. CUDA also has implements a memory hierarchy, similar to how there exist memory hierarchies and cache organizations within the CPU.

The host and the device have a separate memory space so copying data from the host and device memory may be necessary. The memory hierarchy for the device includes its global memory, shared memory, and constant memory. Each memory type has its own advantage such as bandwidth size, access speed and control. The developer has the ability to fine tune the memory use and type for kernel functions in order to optimize computations. Poor memory management can cause bottlenecks, e.g. when copying memory from device to host and vice-versa often. A good memory access pattern would be transferring all the required data to the device and do all the processing within the device before returning the computation result to the host. This access pattern prevents the high-latency transfers between the device and the host.

Optimizing block structure is also important to maximize the parallel structure of the GPU. The physical execution of threads occur in warps of 32 and a not optimal block structure could result in serializing of execution. Lastly, the kernel code itself must be optimized to maximize the use of the threads. GPUs are often used to accelerate computations involving highly parallelizable tasks such as linear algebra operations, while the CPU is more efficient with highly sequential tasks.

5 Algorithm for simulating SN P Systems with delay

In our simulation of SN P systems with delays, we consider the following *cases* when applying rules in the system:

1. Trivially, the spikes contained in the neuron do not satisfy the regular expression E of a rule, hence the rule is not applied.
2. When E of a rule is satisfied and the rule applied with a delay $d > 0$, hence c spikes are consumed (the neuron becomes closed) and begin the countdown of the delay until delay becomes 1.
3. When the countdown for the delay in Case 2 reaches 0 (neuron becomes open), we consider the *net* number of spikes: those spikes produced to other neurons and received from other neurons, possibly including previous spikes in the neuron before the neuron closed.

4. When a rule applied has $d = 0$ which is similar to Case 2 and 3 except that we do not perform any countdown and the neuron for the rule does not become closed.

We also introduce the operation $\otimes$, where $C = A \otimes B$ is the element-wise multiplication of vectors A and B, i.e., for each $x_i \in A$, $y_i \in B$, $z_i \in C$, $z_i = x_i * y_i$. For example, given vector $A = \langle 2, 6, -4, 3\rangle$ and $B = \langle 5, 6, 1, 2\rangle$, we have $A \otimes B = \langle 10, 36, -4, 6\rangle$.

The main simulation algorithm is given in Algorithm 1, which refers to definitions given in Section 3.2. The algorithm is devised to return or produce the $(k+1)$th configuration of an SN P system, given the current step k.

1: **procedure** SIMULATE SNP $(C^{(k)}, R, Tv, St^{(k)})$
2: Reset$(Lv^{(k)})$
3: Reset$(Gv^{(k)})$
4: Reset$(NG^{(k)})$
5: Reset$(Iv^{(k)})$
6: Compute $S^{(k)}$
7: **for** $r_i = \langle E, j, d', c\rangle \in R$ **do** ▷ Check for the cases
8: **if** $S_i^{(k)} = 1$ **then** ▷ Case 2
9: $Lv_j^{(k)} \leftarrow c$
10: $d' \leftarrow d_i$
11: $St_j^{(k)} \leftarrow 0$
12: **if** $d' = 0$ **then** ▷ Case 4
13: $Iv_j^{(k)} \leftarrow 1$
14: $St_j^{(k)} \leftarrow 1$
15: **end if**
16: **else if** $d' = 0$ **then** ▷ Case 3
17: $Iv_j^{(k)} \leftarrow 1$ ▷ Set indicator bit to 1
18: $St_j^{(k)} \leftarrow 1$
19: **end if**
20: **end for**
21: $Gv^{(k)} \leftarrow Tv * Iv^{(k)}$
22: $NG^{(k)} \leftarrow Gv^{(k)} \otimes St^{(k)} + Lv^{(k)}$
23: $C^{(k+1)} \leftarrow C^{(k)} + NG^{(k)}$
24: **for** $r_i = \langle E, j, d', c\rangle \in R$ **do** ▷ Countdown
25: **if** $d' \neq -1$ **then**
26: $d' \leftarrow d' - 1$
27: **end if**
28: **end for**
return $C^{(k+1)}$
29: **end procedure**

Algorithm 1: Simulation of Π from $C^{(k)}$ to $C^{(k+1)}$.

Note that Reset(X) for some vector X resets the vector to a 0 vector to prevent its previous values from interfering with the next iteration of the simulation (i.e. the next step of the computation). Also, the algorithm does not discriminate between the firing and the forgetting rule. That is, a forgetting rule is simply treated as a firing rule that doesn't produce any spikes.

Algorithm 1 takes in an SN P system Π represented using definitions in Section 3.2. The algorithm accepts the initial configuration vector $C^{(0)}$, the rules representation R, the transition vectors Tv, and the status vector $St^{(k)}$. After determining the Spiking Vector $S^{(k)}$, details provided below in Algorithm 2, we check the three cases defined previously (except the trivial case 1). If case 2 applies, where a rule is applied, we set the Loss Vector $Lv^{(k)}$ to c (for the corresponding neuron that contains the rule). Then the counter is started by setting the $d' = d_i$. We then make sure only one applied rule in a neuron will modify a single element of $Lv^{(k)}$, based on the semantics of rule application of SN P systems. The element of $St^{(k)}$ corresponding to the neuron is set to 0, hence closing the neuron.

If the delay of the rule is 0 (case 4), we set the corresponding $Iv_j^{(k)}$ to 1 and open the neuron. Also, the corresponding Status vector element $St_j^{(k)}$ is set to 1. For case 3, we set Tv_j to 1 and open the neuron again by setting $St_j^{(k)}$ to 1. We obtain the Gain Vector $GV^{(k)}$ by multiplying the Transistion Vectors Tv (the rules that released their spikes, i.e. rules where case 3 and 4 applies) to $Iv^{(k)}$. We obtain the Net Gain vector $NG^{(k)}$ using element-wise multiplication of $GV^{(k)}$ and $St^{(k)}$, then adding $Lv^{(k)}$.

The Status Vector acts as a selector where a neuron receives spikes based on its status, after consumed spikes are removed. Finally, we compute for $C^{(k+1)}$ by adding $C^{(k)}$ to $NG^{(k)}$. We reduce each d' for $0 \leq i \leq n$ which signifies the count down. On selecting the Spiking Vector $S^{(k)}$, Algorithm 2 is used.

```
1: procedure COMPUTE S^(k) (C^(k), R^(k))
2:     array n_tmp(0 : m)                          ▷ Initialize an array of size m
3:     for r_i ∈ R do
4:         if St_j^(k) == 0 then
5:             S_i^(k) ← 0                         ▷ Neuron that owns the rule is closed
6:         else if n_tmp_j == 1 then
7:             S_i^(k) ← 0                         ▷ Neuron that already has a rule that applied
8:         else
9:             if L(E_i) matches C_j^(k) then
10:                S_i^(k) ← 1                     ▷ E of rule matches with C^(k)
11:                n_tmp_j ← 1
12:            else
13:                S_i^(k) ← 0                     ▷ E does not match with C^(k)
14:            end if
15:        end if
16:    end for
```

17: **end procedure**

Algorithm 2: Computation of Spiking Vector $S^{(k)}$.

Note that Algorithm 2 only computes for one valid spiking vector of the system, since we only simulate deterministic systems.

6 Results

Algorithms 1 and 2 were implemented in both sequential and parallel code. C++ was used for the sequential implementation while CUDA C for the parallel implementation. The regular expression E in the rules are represented as integers: a^k is stored as k and a^* is stored as -1. We are currently at work in order to include simulations of more general regular expressions, e.g. $a^i(a^j)^*, i, j \geq 0$. Software for this work is available at [2]. The sequential and parallel implementations simulate deterministic SN P systems with delays.

For the CUDA implementation, computations in Algorithms 1 and 2 are performed in the GPU, until simulation halts. The recommended technique or workflow for GPU computing is to initialize inputs at the host (i.e. the CPU), copy inputs to the device (i.e. the GPU), finish all computations in the device, and finally copy results back to the host. This workflow for GPU computing is necessary in order to prevent the overhead (time delays incurred) of data transfer from host to device.

SN P systems implementing generalized sorting networks (provided in [8]) were used as inputs. The input sizes for the sorting networks are of the form 2^n for $n = 1, 2, \ldots, 9$, i.e. from 2 up to 512. The values to be sorted are natural numbers between 0 and 99, randomly generated. For the case of input sizes greater than 100, there will be repetitions of several numbers to be sorted.

The machine set-up for the experiments performed in this work runs an Ubuntu 15.04 64-bit operating system, with an Intel Core i7-4790 CPU with maximum frequency of 4 GHz, and 16 GBytes of memory. The GPU of the set-up is an NVIDIA GeForce GTX 750 with 512 CUDA cores (Kepler microarchitecture) with maximum frequency of 1084 MHz and 2047 MBytes of memory.

The SN P systems used as inputs for both sequential and parallel simulators are the systems implementing generalized sorting networks in [8]. In particular, a sorting network has n input neurons in order to sort n natural numbers. A sorting network of input size n has input neurons $\sigma_1, \ldots, \sigma_n$ containing $r_1, \ldots, r_n$ spikes initially, where the values $r_1, \ldots, r_n$ are the numbers to be sorted (delays are not used in this case).

Figures 2 and 3 illustrate the running time (vertical axis) versus input size (horizontal axis) of both the sequential (i.e. C++SNP) and parallel (i.e. CuSNP) simulators. The 9 inputs, from 2 up to 512, were separated into two charts (given by Figures 2 and 3) since the running time of C++SNP for a 512-input sorting network is much greater (approximately 10 minutes) than the remaining smaller input sizes (under 2 minutes).

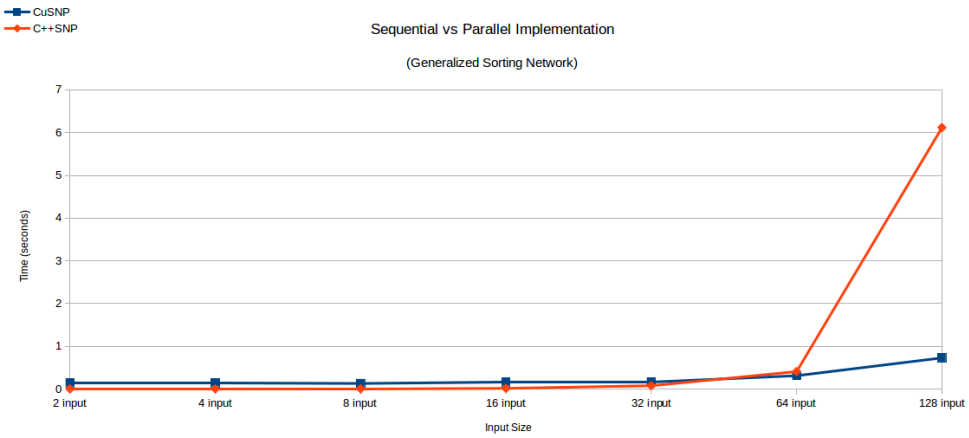

Fig. 2: Runtime Comparison of C++SNP (Sequential) vs CuSNP (Parallel) implementations, simulating SN P systems as generalized sorting networks with input sizes 2 up to 128 (continued in Figure 3). Vertical axis is time, given in seconds.

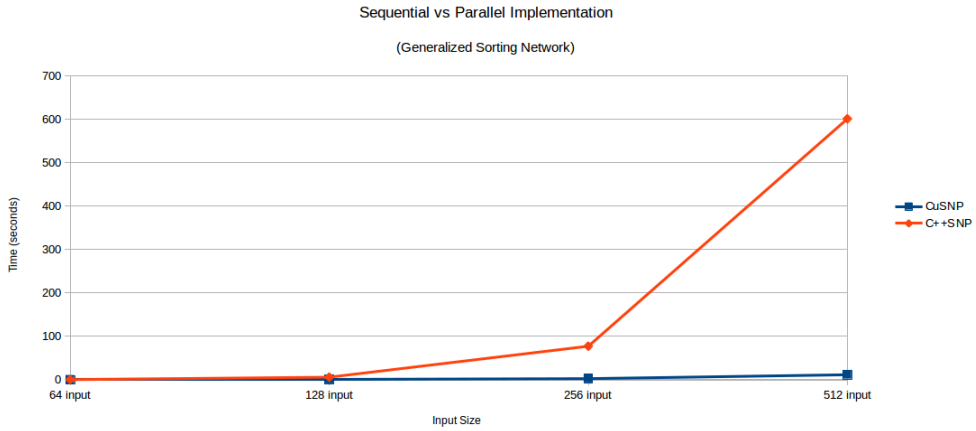

Fig. 3: Runtime Comparison of C++SNP (Sequential) vs CuSNP (Parallel) implementations, simulating SN P systems as generalized sorting networks with input sizes 64 up to 512 (continued from Figure 2. Vertical axis is time, given in seconds.

In Figure 4 we see a chart indicating the speedup, in this case the ratio between the running time of C++SNP over CuSNP, for each input size of the sorting network. Following the GPU computing workflow mentioned above, the larger inputs benefit from being parallelized using CUDA GPUs. It must be noted that for input size 64 and lower, the sequential simulator C++SNP runs faster than CuSNP (see Figure 2, due to the overhead mentioned above. However, for input size 128 and larger, CuSNP overtakes C++SNP in terms of running time as more parallelism is introduced given larger input. This overtaking is also seen in the speedup in Figure 4, where the speedup for input size 64 and lower is less than 1, while speedup for input size 128 and above is greater than 1. In particular, the maximum speedup we obtained is approximately 51 for input size of 512.

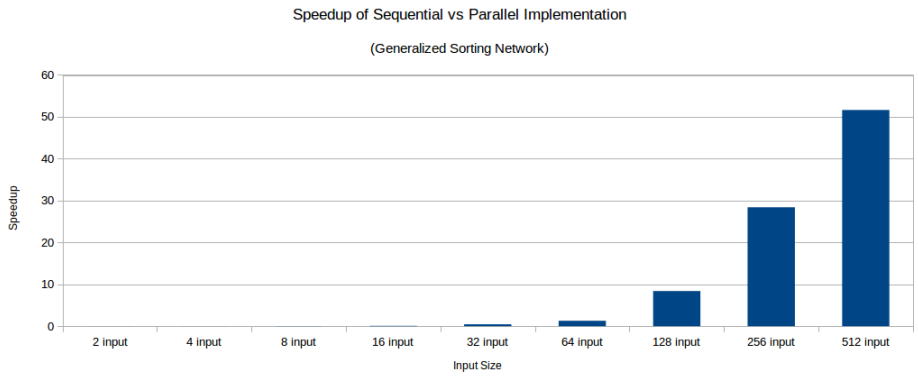

Fig. 4: Runtime speedup of C++SNP (sequential) vs CuSNP (parallel) implementations.

7 Final remarks

In this work we presented our ongoing efforts to simulate SN P systems in NVIDIA CUDA GPUs. In particular, the software available in [2] includes parallel and sequential simulators which simulate SN P systems with delays. We modified the matrix representation in [18] in order to simulate SN P systems with delays. Our experiments were performed using systems implementing generalized sorting networks form [8], and we achieved speedup values of up to 51 times for a 512-size input, i.e. for a 512-size sorting network, CuSNP (parallel simulator) is 51 times faster than C++SNP (sequential simulator). This speedup was obtained, largely in part due to improved memory access pattern between the host (CPU) and the device (GPU).

Much work remains to be done, and we are currently extending the regular expressions available for both simulators to include more general regular expressions (using implementations of finite automata). Also, we are currently optimizing the data types and structures of CuSNP, among other improvements. In a succeeding work, we will also provide detailed profiles of kernel functions (using tools provided by NVIDIA) in CuSNP in order to identify further possibilities for optimizations of simulator performance.

References

1. P systems web page. `http://ppage.psystems.eu/`.
2. CUDA SNP simulators version 06.05.16 (CuSNP v06.05.16). `http://aclab.dcs.upd.edu.ph/productions/software/cusnp_v060516`, 2016.
3. NVIDIA CUDA C programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2016.
4. F. G. C. Cabarle, H. N. Adorna, M. A. Martínez-del Amor, and M. J. Pérez-Jiménez. Improving GPU Simulations of Spiking Neural P Systems. *Romanian Journal of Information Science and Technology*, 15(1):5–20, 2012.
5. F. G. C. Cabarle, H. N. Adorna, M. J. Pérez-Jiménez, and T. Song. Spiking neural P systems with structural plasticity. *Neural Computing and Applications*, 26(8):1905–1917, 2015.
6. J. P. Carandang and J. M. Villaflores. CuSNP: Improvements on GPU Implementation of SNP Systems in NVIDIA CUDA. P*roc. 16th Philippine Computing Science Congress (PCSC2016), Puerto Princesa, Palawan, Philippines*, pages 77–84, 2016.
7. M. Ionescu, G. Păun, and T. Yokomori. Spiking Neural P Systems. *Fundamenta Informaticae*, 71(2,3):279–308, Feb. 2006.
8. M. Ionescu and D. Sburlan. Some Applications Of Spiking Neural P Systems. *Computing and Informatics*, 27(3):515–258, 2008.
9. A. Leporati, C. Zandron, C. Ferretti, and G. Mauri. Solving Numerical NP-Complete Problems with Spiking Neural P Systems. In G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4860 of *LNCS*, pages 336–352. 2007.
10. L. F. Macías-Ramos, M. A. Martínez-del Amor, and M. J. Pérez-Jiménez. Simulating FRSN P systems with real numbers in P-Lingua on sequential and CUDA platforms. *Lecture Notes in Computer Science*, 9504:227–241, 12/2015 2015.
11. L. F. Macías-Ramos, I. Pérez-Hurtado, M. García-Quismondo, L. Valencia-Cabrera, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-Lingua Based Simulator for Spiking Neural P Systems. In M. Gheorghe, G. Păun, G. Rozenberg, A. Salomaa, and S. Verlan, editors, *Membrane Computing*, volume 7184 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2012.
12. M. A. Martínez-del Amor, M. García-Quismondo, L. F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, and M. J. Pérez-Jiménez. Simulating P Systems on GPU Devices: A Survey. *Fundam. Inf.*, 136(3):269–284, July 2015.
13. M. A. Martínez-Del-Amor, L. F. Macías-Ramos, and M. J. Pérez-Jiménez. Parallel simulation of PDP systems: Updates and roadmaps. *13th Brainstorming Week on Membrane Computing, BWMC15, Seville, Spain*, pages 227–244, 2015.

14. L. Pan, G. Păun, and M. J. Pérez-Jiménez. Spiking neural P systems with neuron division and budding. *Science China Information Sciences*, 54(8):1596–1607, 2011.
15. G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108 – 143, 2000.
16. G. Păun, G. Rozenberg, and A. Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford Univeristy Press, 2010.
17. T. Song, L. Pan, and G. Păun. Spiking neural P systems with rules on synapses. *Theoretical Computer Science*, 529:82–95, 2014.
18. X. Zeng, H. Adorna, M. Á. Martínez-del Amor, L. Pan, and M. J. Pérez-Jiménez. *Membrane Computing: 11th International Conference, CMC 2010, Jena, Germany, August 24-27, 2010. Revised Selected Papers*, chapter Matrix Representation of Spiking Neural P Systems, pages 377–391. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.