ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **20**, Number 1, 2017, 57–70

CuSNP: Spiking Neural P Systems Simulators in CUDA

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

[1] Department of Computer Science, University of the Philippines Diliman
Diliman 1101 Quezon City, Philippines;

[2] Department of Computer Science and Artificial Intelligence
University of Seville, Avda. Reina Mercedes s/n, 41012 Sevilla, Spain

Abstract. Spiking neural P systems (in short, SN P systems) are models of computation inspired by biological neurons. CuSNP is a project involving sequential (CPU) and parallel (GPU) simulators for SN P systems. In this work, we report the following results: a P-Lingua file parser is included, for ease of use when performing simulations; extension of the matrix representation of SN P systems to include delay; comparison and analysis of our simulators by simulating two types (bitonic and generalized) of parallel sorting networks; extension of supported types of regular expressions in SN P systems. Our GPU simulator is better suited for generalized sorting as compared to bitonic sorting networks, and the GPU simulators run up to 50× faster than our CPU simulator. Finally, we discuss our experiments and provide directions for further work.

Key-words: Membrane computing; SN P systems; CUDA; GPU

1. Introduction

Spiking neural P systems (in short, SN P systems) are parallel models of computation inspired by the functioning and structure of neurons that was introduced in 2006 in [1]. In this work we report our ongoing efforts to improve *CuSNP*, which is a collection of simulators (sequential and parallel). Many simulators and published works investigate the simulation of membrane models or P systems in software and hardware, whether sequential or in parallel, e.g. [2] and [3]. A survey of simulations of P systems in graphics processing units (in short, GPUs) is [4]. More recently, [5]

includes simulations on a variant of SN P systems known as fuzzy reasoning SN P systems. In order to standardize the simulations of P systems (e.g. in terms of input format) the P-Lingua project was introduced. P-Lingua has also been used to simulate SN P systems in a sequential manner in [6]. Due to the ease of use of P-Lingua, users do not need to know the in-depth details of the simulated P system: as long as the P system for a problem is given, the users can run their simulations using P-Lingua syntax.

Much have been investigated in SNP systems theory, e.g. computability in terms of generating or accepting numbers or languages, as in [1, 7–12], and computational efficiency as in [13–15]. These investigations have also been applied to several variants of SNP systems, where the variants take various inspirations from biology, mathematics, or computer science, e.g. [16–20]. SNP systems were also used to solve combinatorial optimization problems [21] and to diagnose faults in power systems [22].

The following are our contributions in this work: (1) we included in CuSNP the feature of allowing `.pli` files or input files for P-Lingua simulator as input to our simulators; (2) we modify the matrix representation of SN P systems in [23] in order to simulate SN P systems with delays, and we prove that our algorithms indeed simulate SN P systems computations; (3) we test our sequential and parallel simulators using two types of sorting networks: generalized and bitonic, provided in more detail in [24] and [25], respectively. At present, the former type performs better in our simulators due to rule density (more details later). We also analyze and profile our simulators in order to gain insights on how to better improve them; (4) We provide an algorithm based on finite automata, different from [26], to allow more forms of regular expressions.

This work is organized as follows: Section 2 provides syntax and semantics for SN P systems; Section 3 discusses the main elements of CuSNP; Section 4 provides the matrix definitions that allow us to simulate SN P systems with delays. The definitions are then used to create the algorithms used by CuSNP. Section 5 provides experimental results and analyses of our experiments. We end with conclusions and future research directions in Section 6.

2. Spiking Neural P Systems

The reader is assumed to be familiar with basics of membrane computing and formal language theory. 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 symbol (the spike);
2. $\sigma_1, ..., \sigma_m$ are neurons, of the form $\sigma_i = (\alpha_i, R_i), 1 \leq i \leq m$, where:
   (a) $\alpha_i \geq 0$ 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) (Spiking Rule) $E/a^c \rightarrow a^p; d$ where E is a regular expression over O and $c \geq p \geq 1, d \geq 0$.

(*ii*) (Forgetting Rule) $a^s \to \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \to a^p; d$ of type (i) from R_i, we have $a^s \notin L(E)$;

3. $syn \subseteq \{1, 2, ..., m\} \times \{1, 2, ..., m\}$ 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.

An SN P system whose spiking rules have $p = 1$ is said to be of the standard type (non-extended). A spiking rule is applied as follows: if a neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \to a^p; d \in R_i$ 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 after d time steps. Spikes are fired after $t + d$ steps where t is the current time step of the computation. If $d = 0$, the spikes are fired immediately. Between step t and $t + d$, we say σ has not fired the spike yet and is *closed*, i.e. σ_i cannot receive spikes from other neurons connected to it. If neurons with a synapse to σ fire, the spikes are lost. At step $t + d$, the spikes are fired, and σ_i is now *open* to receive spikes. At $t + d + 1$, σ_i can apply a rule.

A forgetting rule is applied as follows: If σ_i contains exactly s spikes, then the rule $a^s \to \lambda$ from R_i can be applied, meaning all of the s spikes are removed from σ_i. Rules of type (1) where $E = a^c$ can be written in the short form of $a^c \to a^p; d$. In the case two or more rules of σ_i are applicable at the same step, only one rule is applied and is non-deterministically chosen. A *configuration* of the system at step t is denoted as $C_t = \langle r_1/k_1, \ldots, r_m/k_m \rangle$ for $1 \leq i \leq m$, where σ_i contains $r_i \geq 0$ spikes and remains closed for k_i more steps. The initial configuration of the system is therefore $C_0 = \langle r_1/0, \ldots, r_m/0 \rangle$. 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 obtain the output of an SN P system are as follows: (1) as the time interval between 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. In this work, we consider systems that produce their output as given in (2). SN P systems are represented graphically as directed graphs: the neurons are represented by oval vertices, and synapses are the arcs between the vertices.

3. Technologies for CuSNP

The following discussion provides details of the `.pli` file parser included in CuSNP. We follow the P-Lingua syntax provided in [6] and identify the important reserved keywords that will be relevant for CuSNP, e.g. `@mu = m1, m2, ..., mN; @marcs = (m1, m2); @ms(m1) = a*k;` where `@mu, @marcs`, and `@ms` define the neurons, the

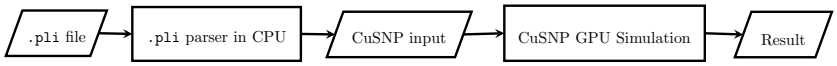

Figure 1: Workflow diagram for `.pli` parser in CuSNP.

synapses between neurons, and the initial spikes contained in each neuron, respectively. Rules are strictly limited to spiking and forgetting rules, since other types of rules are also not relevant to CuSNP. Rules can be defined as `[x]'n --> [y]'n "r"::d;` where `x` is the number of spikes consumed, `y` is the number of spikes produced (that is given a rule $E/a^c \rightarrow a^p; d,\ x = c\ and\ y = p$), `n` is the label of the neuron containing the rule, `r` is the regular expression E, and `d` is the delay count for the rule. An example of a P-Lingua code that defines a rule is `[a --> a]'in{i} "a*": 1<=i<=n;`. In this example, an optional range of values for expression `i` can be appended at the end of each line of code. Using the `.pli` parser, we have the workflow illustrated in Figure 1. The workflow for CuSNP allows users with minimal technical knowledge of SN P systems to be able to perform experimental simulations in GPUs using familiar language as found in P-Lingua.

Next, we briefly discuss the Compute Unified Device Architecture (in short, CUDA) for GPU programming. CUDA is a parallel programming computing platform and application programming interface model developed by NVIDIA [27] [28]. CUDA allows software developers to use a CUDA enabled GPUs for general purpose GPU (in short, GPGPU) computing. Functions that execute in the GPU, known as *kernel functions*, are executed by one or more threads arranged in thread blocks. The thread blocks are then arranged in a grid of thread blocks. The arrangement of this thread hierarchy is left to the developer with a given constraint. In the CUDA programming model, the GPU is often referred to as the *device*, while the CPU is referred to as the *host*. The host performs the kernel function calls to be executed on the device.

CUDA uses a single program, multiple data (in short, SPMD) paradigm. In this paradigm, threads execute similar code, while allowing such threads to access multiple (possibly different values of) data. Aside from the thread hierarchy, CUDA also implements a memory hierarchy similar to how there exist memory hierarchies in the CPU. We do not go into details of the thread and memory hierarchy of CUDA and instead refer the reader to [27] [28]. In CuSNP and as began in [26], we follow a good memory access pattern as follows: the host generates the input (i.e. the numbers to be sorted) and then copies the input to the GPU; the device performs the entire simulation until the simulated SN P system halts; finally, the output of the device is then copied back to the host for printing and analysis. This access pattern is followed in order to prevent slow transfers (i.e. high latency) between the device and the host. A good practice for designing the hierarchy of threads used in the device is that the threads in a block must be a multiple of 32 to maximize GPU warps.

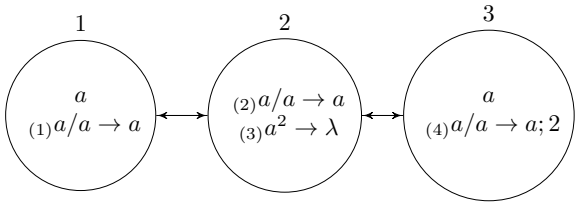

Figure 2: A 3-neuron SN P system

4. Algorithms for CuSNP

In this section we first introduce the modifications of the matrix representation given in [23] in order to simulate SN P systems with delay. After the modifications are presented, the algorithms for the simulation in CuSNP are provided. Let Π be an SN P system with delay having m neurons and n rules. We use the following definitions to represent Π.

Definition 1: (Configuration Vector). The configuration vector $C^{(k)} = \langle c_1, ..., c_m \rangle$, where c_i is the amount of spikes in σ_i at time k.

Definition 2: (Spiking Vector). A **spiking vector** $S^{(k)} = \langle s_1, ..., s_n \rangle$ where

$$s_i^{(k)} = \begin{cases} 1, \text{if } E_i \text{ is satisfied and } r_i \text{ is applied} \\ 0, \text{otherwise.} \end{cases}$$

Definition 3: (Status Vector). The kth status vector is denoted by $St^{(k)} = \langle st_1, ..., st_m \rangle$ where for each $i \in \{1, 2, ..., m\}$,

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

Definition 4: (Rule Representation). $R^{(k)} = \langle r_1, ..., r_n \rangle$ where for each $i = 1, ..., n$, $r_i = (E, j, d', c)$ where E is the regular expression for rule i, j is the neuron that contains the rule r_i,

$$d' = \begin{cases} -1, \text{if the rule is not fired,} \\ 0, \text{if the rule is fired,} \\ \geq 1, \text{if the rule is currently on delay (i.e. neuron is closed).} \end{cases}$$

and c is the number of spikes in neuron σ_j consumed if r_i is applied.

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

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

Definition 7: (Gain Vector). The kth gain vector is denoted by $GV^{(k)} = \langle gv_1, gv_2, ..., gv_m \rangle$ where for each $i \in \{1, 2, ..., m\}$, gv_i is the number of spikes sent by neighboring neurons to neuron σ_i at the step k.

Definition 8: (Transition Matrix). The transition matrix[1] of Π, is an ordered

[1]The spiking transition matrix in [23] contains both spikes consumed and produced by each neuron at step k. Transition matrix, however, only contains spikes gained by each neuron.

set of vectors TV defined as $TV = \{tv_1, ..., tv_n\}$ where for each $i \in \{1, 2, ...n\}$, $tv_i = \langle p_1, ..., p_m \rangle$ such that if $r_i \in \sigma_s$:

$$p_j = \begin{cases} \text{number of spikes produced by } r_i, \text{ if } (s, j) \in syn \\ 0, \text{ otherwise.} \end{cases}$$

Definition 9: (Indicator Vector) The indicator vector $IV^k = \langle iv_1, ..., iv_m \rangle$ indicates which rule will produce spikes at time k.

Definition 10: (Removing Matrix) The removing matrix of Π is $RM = \{rm_1, rm_2, ..., rm_n\}$ where for each $i \in \{1, 2, ...n\}$, $rm_i = \langle t_1, ..., t_m \rangle$ such that if $r_i \in \sigma_s$:

$$t_j = \begin{cases} \text{number of spikes consumed by } r_i, \text{ if } s = j \\ 0, \text{ otherwise.} \end{cases}$$

Definition 11: (Net Gain Vector). The **Net Gain vector** of Π at step k is defined as $NG^{(k)} = C^{(k+1)} - C^{(k)}$

From Figure 2, we have the following for $k = 1$: $C^{(0)} = \langle 1, 0, 1 \rangle$; spiking vector is $S^{(1)} = \langle 1, 0, 0, 1 \rangle$; A status vector is $St^{(1)} = \langle 1, 1, 1 \rangle$; $R^{(0)} = \langle r_1, r_2, r_3, r_4 \rangle$ where: $r_1 = \{a, 1, 0, 1\}$, $r_2 = \{a, 2, -1, 1\}$, $r_3 = \{a^2, 2, -1, 2\}$, $r_4 = \{a, 3, 0, 1\}$; The delay vector is $D = \langle 0, 0, 0, 2 \rangle$; The loss vector is $LV^{(0)} = \langle 1, 0, 1 \rangle$; The gain vector is $GV^{(0)} = \langle 0, 1, 0 \rangle$; The indicator vector is $IV^1 = \langle 1, 0, 0 \rangle$; We have

$$TV = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \text{ and } \quad RM = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

as the transition and removing matrix, respectively.

Lemma 1 *Let Π be an SN P systems with delay $d > 0$. The net gain vector can be obtained by $NG^{(k)} = St^{(k)} \otimes GV^{(k)} - LV^{(k)}$, where $\otimes$ is the element-wise multiplication operator, $GV^{(k)} = IV^{(k)} \cdot TV$, and $LV^{(k)} = S^{(k)} \cdot RM$.*

Proof For the loss vector $LV^{(k)}$, when a neuron applies a rule, the neuron immediately consumes the required spikes. RM indicates the number of spikes σ_i will consume if rule j is applied. The spiking vector $S^{(k)}$ selects which rule will fire so we can compute the number of spikes each neuron will lose. $S^{(k)}$ is an $1 \times m$ vector and RM is a $n \times m$ matrix so their product is a $1 \times m$ vector (the loss vector). For the gain vector $GV^{(k)}$, if σ_i applies a rule at step k, σ_i will release its spike at step $k + d$ given a delay d for the rule. $IV^{(k)}$ indicates which rule will release its spike at step k. TV is an $n \times m$ matrix indicating how many spikes σ_i will gain at the current step. $IV^{(k)}$ is an $1 \times m$ vector and TV is an $n \times m$ matrix, and their product provides a $1 \times m$ matrix that represents the number of spikes each neuron will gain. The gain vector is then multiplied to the status vector $St^{(k)}$ which sets the gain of closed neuron to 0 since closed neurons cannot receive spikes. □

Using Lemma 1, we can compute the next configuration as follows:

Theorem 2 $C^{(k+1)} = C^{(k)} + St^{(k)} \otimes (IV^{(k)} \cdot TV) - S^{(k)} \cdot RM$.

Following the definitions of the vectors provided above, we have for the system Π in Figure 2 the vectors $C^{(1)} = \langle 1, 0, 1\rangle$, $GV^{(1)} = \langle 0, 1, 0\rangle$, $St^{(1)} = \langle 1, 1, 0\rangle$, $LV^{(1)} = \langle 1, 0, 1\rangle$. We can compute $NG^{(1)} = GV^{(1)} \otimes St^{(1)} - LV^{(1)} = \langle -1, 1, -1\rangle$. We also compute the next configuration vector as $C^{(2)} = C^{(1)} + NG^{(1)} = \langle 0, 1, 0\rangle$.

Next, we provide the algorithms that make use of the definitions provided above. Given an SN P System with delay, we identify three cases for rule application in our algorithms. For each $r_i = (E, j, d', c) \in R^{(k)}$, we have the following cases: **(Case 1)** If $L(E) = a^{c_j}$, then $d' = d_i$, $St_j^{(k)} = 0$, $Lv_j^{(k)} = c$. In this case, the rule is fired, we start a countdown, and consume the required spikes. The neuron is now closed and cannot receive any spikes; **(Case 2)** If the countdown from case 1 is finished i.e. r_i was fired at step $k - d_j$ so that d' is now 0, we add the spikes that the r_i produces and open the neuron that owns r_i. We perform these by setting $IV_j^{(k)} = 1$ and $St_j^{(k)} = 1$; **(Case 3)** When r_i has a delay of 0, i.e. $d_j = 0$, we apply cases 1 and 2.

The main simulation algorithm for CuSNP given in Algorithm 1 is devised to compute the kth configuration.

The function Reset$(X_1, ..., X_n)$ given a list of vectors X_i, resets the list of vectors to a 0 vector. We now prove that Algorithm 1 indeed provides the kth configuration.

Theorem 3 *Algorithm 1 provides $C^{(k+1)}$ given inputs $C^{(k)}, R, Tv$, and $St^{(k)}$.*

Proof Algorithm 1 accepts as inputs the initial configuration vector $C^{(k)}$, the rules representation R, the status vector $St^{(k)}$, and the transition matrix TV. After determining the spiking vector at line 3, we check the three cases as defined before from line 5 to line 13. If case 1 applies, we set the value c to the corresponding element of the loss vector depending on the neuron that owns the rule (i.e. $LV_j \leftarrow c$ where $j, c \in r_i = (E, j, d', c)$. The counter is started by setting the value for d'. We make sure that only one rule will modify a single element of the loss vector based on the semantics of rule selection in SN P systems. We also set the corresponding status vector element of the neuron to 0, signifying that the neuron is closed.

For case 3, we set the corresponding iv_i to 1 and open the neuron by setting the corresponding st_j element to 1. When case 2 applies, we set iv_i to 1 and open the neuron by setting st_j to 1. We obtain the gain vector at line 18 by multiplying the IV to TV. IV is used to select which rule will send out its spike, i.e. rules where case 2 and case 3 apply. The net gain vector is given by line 19.

The status vector is used to select where the neuron will receive spikes depending on the status of the neuron, while removing the spikes consumed. 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. The vectors LV, GV and TV are reset to prevent their current values from interfering with the next step of computation. □

The next algorithm is for the Compute $S^{(k)}$ function in Algorithm 1.

Another algorithm was devised in order to be able to implement the checking of regular expressions in the GPU. The algorithm for solving regular expression is specifically designed for the singleton alphabet of SNP Systems through the use of deterministic finite automata (in short, DFA). The supported regular expressions

Algorithm 1 Main simulation algorithm for CuSNP.

procedure SIMULATE SNP $(C^{(K)}, R, Tv, St^{(k)})$
  Reset($LV^{(k)}$, $GV^{(k)}$, $NG^{(k)}$, $IV^{(k)}$)
  Compute $S^{(k)}$
  for $r_i = \{E, j, d', c\} \in R$ **do** ▷ Check for the cases
    if $S_i^{(k)} = 1$ **then** ▷ Case 1
      $LV_j^{(k)} \leftarrow c$
      $d' \leftarrow d_i$
      $IV_i^{(k)} \leftarrow 0$
      if $d' = 0$ **then** ▷ Case 3
        $IV_i^{(k)} \leftarrow 1$
        $St_j^{(k)} \leftarrow 1$
      end if
    else if $d' = 0$ **then** ▷ Case 2
      $IV_i^{(k)} \leftarrow 1$ ▷ Set indicator bit to 1
      $St_j^{(k)} \leftarrow 1$
    end if
  end for
  $GV^{(k)} \leftarrow TV * IV^{(k)}$
  $NG^{(k)} \leftarrow GV^{(k)} \otimes St^{(k)} - LV^{(k)}$
  $C^{(k+1)} \leftarrow C^{(k)} + NG^{(k)}$
  for $r_i = \{E, j, d', c\} \in R$ **do** ▷ Countdown
    if $d' \neq -1$ **then**
      $d' \leftarrow d' - 1$
    end if
  end for
  return $C^{(k+1)}$
end procedure

Algorithm 2 Compute Spiking Vector

```
1: procedure COMPUTE S^(k) (C^(k), R^(k))
2:     for r_i ∈ R do
3:         if St_j^(k) = 0 then
4:             S_i^(k) ← 0                          ▷ Neuron that owns the rule is closed
5:         else
6:             if L(E_i) matches C_j^(k) then
7:                 S_i^(k) ← 1                      ▷ Rule E is satisfied in C^(k)
8:             else
9:                 S_i^(k) ← 0                      ▷ Rule E did not match with C^(k)
10:            end if
11:        end if
12:    end for
13: end procedure
```

have the following five forms: a^*, a^+, a^k where $k \geq 1$, $a^k(a^j)^*$ and $a^k(a^j)^+$ where $k, j \geq 1$. We can represent all five forms of regular expressions as a DFA with at most 3 states if we allow transitions with a^k labels, denoting a series of states and transition requiring k copies of a's. If we do not have enough copies of a to go through the a^k edge, we do not accept such input. For example, we represent $a^2(a^3)^*$ as having $\langle 2, 3, 3\rangle$, $\langle 2, 3, 3\rangle$, and $\langle 0, 1, 1\rangle$ as the first, second, and third rows, respectively, of the matrix. To simulate the regular expression checking, we begin at the first column (i.e. the representation of the first state). When we have enough spikes to move to the next state, we subtract the number of spikes needed from the current number of spikes and move to the next one. The regular expression matches the number of spikes if no spikes remain after the algorithm halts and the DFA is in an accepting state.

The parallel implementation of Algorithm 1 is as follows: Assign a thread to each rule or neuron depending on the kernel function executed in the device. Given m neurons and n rules, we first allocate n threads for each $r_i \in R^{(k)}$ to compute the spiking vector using Algorithm 2. We can then assign n threads again to set the proper values for $R^{(k)}, St^{(k)}, LV^{(k)}, IV^{(k)}$ based on the definition given. We then compute $GV^{(k)} = IV^{(k)} \cdot TV$, $NG^{(k)} = GV^{(k)} \otimes St^{(k)} - LV^{(k)}$, and $C^{(k+1)} = C(k) + NG^{(k)}$ by using m threads. All data is stored in the global memory of the device. Initialization and pre-processing of input data is performed in the host before running the alogorithm. Pre-processing includes converting the input to the correct data structure for use in the device and copying of the data from the host memory to the device memory. After the simulation ends, data is again copied back to the host for post-processing (e.g. printing).

5. Experimental Results with CuSNP

Algorithm 1 is implemented in C++ and CUDA C++. CuSNP is able to simulate only deterministic SN P systems with delays. For both generalized and bitonic sorting networks, we compare the outputs of the sequential (CPU) and parallel (GPU) simulators. The latest version of the CuSNP code is available at [29]. The machine for the simulation runs on Intel Core i7-4790 at 3.60GHz, 16 GB Ram, with NVDIA GeForce 750 graphics card running on an Ubuntu Linux 15.04 system and CUDA v7.5.17. The graphs below show the output of the built-in *time* function in Linux when the simulators are run. Generalized sorting networks with input size of $2^i, i = 1, \ldots, 9$, were generated for testing. The simulators were run for 100 steps and the input number used for sorting was randomly generated from $0 - 99$ inclusive allowing repetitions. The 2-input generalized sorting network (smallest size) has 6 neurons and 6 rules while the largest (512-input) has 1,536 neurons and 262,656 rules. All simulations ran 100 steps to ensure the systems were tested in similar conditions.

The system was run using the sequential version which we label as C++SNP. The version of C++SNP in [30] and [26] only supports rule of the form a^k, a^*, and a^+ while in this work we now include the forms $a^k(a^j)^*$ and $a^k(a^j)^+$. The support for additional forms of regular expression comes at a linear cost, i.e. previous versions check regular expressions in constant time while the version in this work (using the DFA algorithm in the previous section) performs checking in linear time.

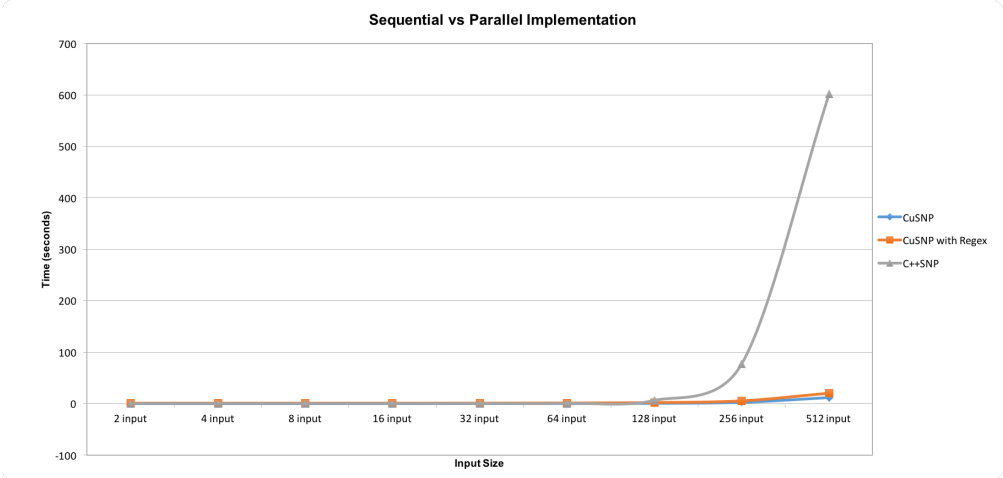

Figure 3: Runtime Comparison of C++SNP (CPU) vs CuSNP (GPU) simulators.

As shown in Figure 3, we notice the exponential growth of runtime with C++SNP due to the exponential size of the simulated networks, while the GPU simulator follows a more linear growth on the runtime. A time difference can be noticed between the version in [30] and the current version: a slight increase in the running time of the latter due to the DFA algorithm. The GPU simulator shows a slower performance with smaller input sizes but we see a speed up of greater than 1 at the 128-input generalized sorting network, in particular around a $9\times$ speed up. The GPU simulator

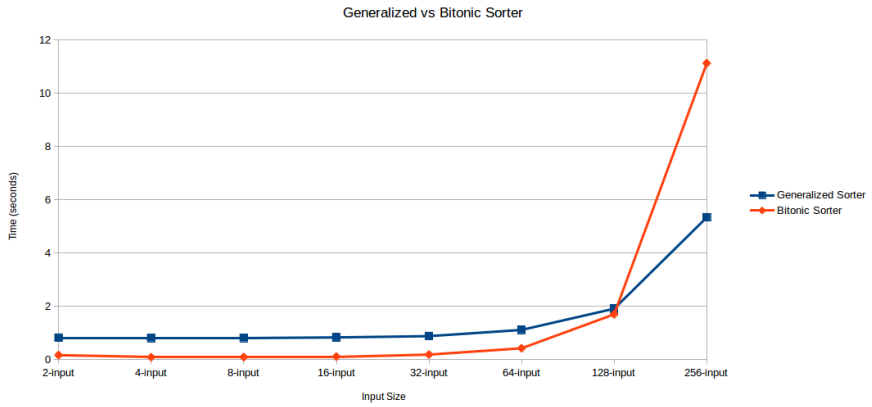

Figure 4: Generalized Sorting vs Bitonic Sorting Networks using CuSNP with regular expression.

obtains up to 50× speed up with a 512-input generalized sorting network.

The generalized sorting network simulation was also profiled using the *nvprof* utility that comes with CUDA. The runtime of each kernel function was profiled and compared with each other. The kernel functions are as follows: *SNPSolveRegex* solves regular expression of each $r_i \in R^{(k)}$ and sets the corresponding element of $S^{(k)}$. *SNPFixSpikingVector* ensures that only 1 rule per neuron will activate to follow SNP semantics. *SNPSetStates* sets the value of $IV^{(k)}$, $St^{(k)}$ and $d' \in r_i$ according to the four cases described in Section 4. *Matrix_Multiply* computes for $GV^{(k)} = IV^{(k)} \cdot TV$. *SNPCompuateNG* computes for $NG^{(k)} = GV^{(k)} \otimes St^{(k)} - LV^{(k)}$. *Vector_Addition* computes for $C^{(k+1)} = NG^{(k)} + C^{(k)}$. *SNPPostComputes* Subtracts 1 from the timer d' of all $r_i \in R^{(k)}$ if possible and sets $IV^{(k)}$ to a zero vector. And finally, *SNPReset* Converts the vectors $LV^{(k)}$, $GV(k)$ and $NG(k)$ to a zero vector. Based on our analysis, the kernel functions *Matrix_Multiply* and *SNPFixSpikingVector* consume the most resources at increasing input size, taking up 52.47% and 47.33% of the time, respectively. This shows that optimizing this functions would lead to a significant increase in simulation time at larger input sizes.

The CPU and GPU simulators were also run using the bitonic sorting networks or bitonic sorters, as given in [25]. Compared to the generalized sorter, the bitonic sorter uses several smaller modules composed of generalized sorters of size 2. We define the rule density of an SN P System as the ratio of total number of rules n over the total number of neurons n in the system. This way, the density of the generalized sorter doubles for every input size (powers of 2) considered in this work, while the density for bitonic sorters remains almost constant for any input size. The bitonic sorters are "simpler" in the sense that on average, each neuron has around 2 rules compared to the generalized sorter having more.

Figure 4 compares the runtime of generalized and bitonic sorters. At present, our simulations cannot handle the bitonic sorters of input size grater than 512 due to

thread limitations. Figure 4 shows that while bitonic sorters runs slightly faster at smaller input (i.e.) size 2 to 64, the generalized sorting network did better at larger input. This is due to the bitonic sorter, using more rules and neurons to perform a similar task of sorting numbers.

6. Final Remarks

In this work we presented several improvements on CuSNP, in particular: we proved the algorithms used by CuSNP; we allowed a workflow beginning with a `.pli` file as input up to the CuSNP output, allowing for easier access to GPU accelerated experiments; we provided an algorithm based on DFA to support more regular expression forms; we simulated and compared generalized and bitonic sorters; finally, we profiled our simulators to give us insights on how to further improve CuSNP. The systems we reported use generalized sorters of input size 512 having 1,536 neurons and 262,656 rules. Using generalized sorters of input size beyond 512 is not possible at the present version of CuSNP, due to imposed thread limitations for each kernel function. Some of the limitations of the `.pli` parser in this work include lack of error checking for the input syntax, and some Java regular expressions (e.g. "?") are not yet supported.We expect to address more of these limitations in our continuing work. We intend to further improve CuSNP by using sparse representations and operations on vectors and matrices. Lastly, we also intend to make use of multiple devices or GPUs, to further accelerate the runtime of our simulations.

Acknowledgements.
Cabarle is grateful for the support of the HRIDD HRDP grant I-15-0626-06 of the DOST PCIEERD, a FRIA 2016–2017 grant from the College of Engineering, UP Diliman, the PhDIA Project No. 161606 from the UP Diliman, OVCRD, the Soy and Tess Medina professorial chair 2016–2017, and an RLC grant 2016–2017 also from OVCRD. H. Adorna is supported by the following, all from UP Diliman: Semirara Mining Corp. professorial chair, the Gawad Tsanselor Award grant 2015-2016 and an OVCRD RLC grant 2014-2015.

References

[1] Ionescu, M., Păun, G., Yokomori, T.: Spiking Neural P Systems. Fundamenta Informaticae **71**(2,3) (February 2006) 279–308

[2] Cabarle, F.G.C., Adorna, H.N., Martínez-del Amor, M.Á., Pérez-Jiménez, M.J.: Improving GPU Simulations of Spiking Neural P Systems. Romanian Journal of Information Science and Technology **15**(1) (2012) 5–20

[3] Macías-Ramos, L.F., Pérez-Jiménez, M.J., Song, T., Pan, L.: Extending Simulation of Asynchronous Spiking Neural P Systems in P–Lingua. Fundamenta Informaticae **136** (2015) 253–267

[4] Martínez-del Amor, M., García-Quismondo, M., Macías-Ramos, L., Valencia-Cabrera, L., Riscos-Núñez, A., Pérez-Jiménez, M.: Simulating P Systems on GPU Devices: A Survey. Fundamenta Informaticae **136** (2015) 269–284

[5] Macías-Ramos, L.F., Martínez-del Amor, M.Á., Pérez-Jiménez, M.J.: Simulating FRSN P Systems with Real Numbers in P-Lingua on Sequential and CUDA Platforms. LNCS **9504** (2015) 227–241

[6] Macías-Ramos, L.F., Pérez-Hurtado, I., García-Quismondo, M., Valencia-Cabrera, L., Pérez-Jiménez, M.J., Riscos-Núñez, A.: A P–Lingua Based Simulator for Spiking Neural P Systems. In Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S., eds.: Membrane Computing: 12th International Conference, CMC 2011, Fontainebleau, France, August 23-26, 2011, Revised Selected Papers. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 257–281

[7] Chen, H., Ionescu, M., Ishdorj, T.O., Păun, A., Păun, G., Pérez-Jiménez, M.J.: Spiking Neural P Systems with Extended Rules: Universality and Languages. Natural Computing **7**(2) (2008) 147–166

[8] Ibarra, O., Woodworth, S.: Characterizing Regular Languages by Spiking Neural P Systems. International Journal of Foundations of Computer Science **18**(6) (2007) 1247–1256

[9] Neary, T.: Three Small Universal Spiking Neural P Systems. Theoretical Computer Science **567**(C) (2015) 2–20

[10] Păun, A., Păun, G.: Small Universal Spiking Neural P Systems. BioSystems **90**(1) (2007) 48–60

[11] Pan, L., Păun, G.: Spiking Neural P Systems: An Improved Normal Form. Theoretical Computer Science **411**(6) (2010) 906–918

[12] Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J.: Notes on Spiking Neural P Systems and Finite Automata. Natural Computing **15**(4) (2016) 533–539

[13] Leporati, A., Zandron, C., Ferretti, C., Mauri, G.: Solving Numerical NP-Complete Problems with Spiking Neural P Systems. Lecture Notes in Computer Science **4860 LNCS** (2007) 336–352

[14] Ishdorj, T.O., Leporati, A., Pan, L., Zeng, X., Zhang, X.: Deterministic Solutions to QSAT and Q3SAT by Spiking Neural P Systems with Pre-Computed Resources. Theoretical Computer Science **411**(25) (2010) 2345–2358

[15] Leporati, A., Mauri, G., Zandron, C., Păun, G., Pérez-Jiménez, M.J.: Uniform Solutions to SAT and Subset Sum by Spiking Neural P Systems. Natural Computing **8**(4) (2009) 681–702

[16] Wang, J., Hoogeboom, H., Pan, L., Păun, G., Pérez-Jiménez, M.: Spiking Neural P Systems with Weights. Neural Computation **22**(10) (2010) 2615–2646

[17] Pan, L., Păun, G.: Spiking Neural P Systems with Anti-Spikes. International Journal of Computers, Communications and Control **4**(3) (2009) 273–282

[18] Pan, L., Wang, J., Hoogeboom, H.: Spiking Neural P Systems with Astrocytes. Neural Computation **24**(3) (2012) 805–825

[19] Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J., Song, T.: Spiking Neural P Systems with Structural Plasticity. Neural Computing and Applications **26**(8) (2015) 1905–1917

[20] Song, T., Pan, L., Păun, G.: Spiking Neural P Systems with Rules on Synapses. Theoretical Computer Science **529** (2014) 82 – 95

[21] Zhang, G., Rong, H., Neri, F., Pérez-Jiménez, M.J.: An Optimization Spiking Neural P System for Approximately Solving Combinatorial Optimization Problems. International Journal of Neural Systems **24** (2014) 1–16

[22] Wang, T., Zhang, G., Zhao, J., He, Z., Wang, J., Prez-Jimnez, M.J.: Fault Diagnosis of Electric Power Systems Based on Fuzzy Reasoning Spiking Neural P Systems. IEEE Transactions on Power Systems **30**(3) (May 2015) 1182–1194

[23] Zeng, X., Adorna, H., Martínez-del Amor, M.Á., Pan, L., Pérez-Jiménez, M.J. In: Matrix Representation of Spiking Neural P Systems. Springer Berlin Heidelberg, Berlin, Heidelberg (2011) 377–391

[24] Ionescu, M., Sburlan, D.: Some Applications of Spiking Neural P Systems. Computing and Informatics **27**(3) (2008) 515–258

[25] Ceterchi, R., Tomescu, A.I.: Implementing Sorting Networks with Spiking Neural P Systems. Fundam. Inf. **87**(1) (January 2008) 35–48

[26] Carandang, J., Villaflores, J., Cabarle, F.G.C., Adorna, H., Martínez-del Amor, M.Á.: Improvements on GPU Simulations of Spiking Neural P Systems in CUDA GPUs: CuSNP. 14th Brainstorming Week on Membrane Computing (2016) 135–150

[27] NVIDIA corporation: CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide` (2015) Accessed: 2015-11-19.

[28] Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. 2 edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2013)

[29] Carandang, J., Villaflores, J., Cabarle, F.G.C., Adorna, H., Martínez-del Amor, M.Á.: CuSNP Version 06.06.16. `http://aclab.dcs.upd.edu.ph/productions/software/cusnp_v060616` (2016)

[30] Carandang, J., Villaflores, J., Cabarle, F.G.C., Adorna, H.: CuSNP: Improvements on GPU Simulations of Spiking Neural P Systems in CUDA. 16th Philippine Computing Science Congress 77–84