A Spiking Neural P System Simulator Based on CUDA

Francis George C. Cabarle[1], Henry Adorna[1], and Miguel A. Martínez–del–Amor[2]

[1] Algorithms & Complexity Lab,
Department of Computer Science,
University of the Philippines Diliman,
Diliman 1101 Quezon City, Philippines
fccabarle@up.edu.ph, hnadorna@dcs.upd.edu.ph
[2] Research Group on Natural Computing,
Department of Computer Science and Artificial Intelligence,
University of Seville,
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
mdelamor@us.es

Abstract. In this paper we present a Spiking Neural P system (SNP system) simulator based on graphics processing units (GPUs). In particular we implement the simulator using NVIDIA CUDA enabled GPUs. The massively parallel architecture of current GPUs is very suitable for the maximally parallel computations of SNP systems. We simulate a wider variety of SNP systems, after presenting a previous work on SNP system matrix representation which led to their simulation in GPUs, and the simulation algorithm included here. Finally, we compare and present the performance speedups of the CPU-GPU based simulator over the CPU only simulator.

Keywords: Membrane Computing, Spiking Neural P systems, Parallel Computing, GPU Computing, CUDA.

1 Introduction

Inspiration taken from *nature* can help us define new *computing models*, with the aim of providing efficient solutions to the limitations of conventional models of computation. In this respect *Membrane Computing*, a research area initiated by Gheorghe Păun in 1998 [14], provides distributed, parallel, and nondeterministic computing models known as *P systems*. These models are basically *abstractions* of the compartmentalized structure and parallel processing of biochemical *information* in biological cells.

Many P system *variants* have been defined in literature, and many of them have been proven to be *computationally complete*. Moreover, several general classifications of P systems are considered depending on the level of abstraction: *cell-like* (a rooted tree where the *skin* or outermost cell membrane is the root,

and its inner membranes are the children or leaf nodes in the tree), *tissue-like* (a graph connecting the cell membranes) and *neural-like* (a directed graph, inspired by *neurons* interconnected by their axons and synapses). The last type refers to *Spiking Neural P systems* (in short, SNP systems), where the *time difference* (when neurons fire and/or spike) plays an essential role in the computations [11].

One key reason of interest for P systems is that they are able to solve computationally *hard problems* (e.g. NP-complete problems) usually in *polynomial* to *linear time* only, but requiring *exponential space* as trade off. These solutions are inspired by the capability of cells to produce an *exponential* number of new membranes via methods like *mitosis* (membrane division) or *autopoiesis* (membrane creation). However, because of this massively parallel, distributed and nondeterministic nature of P systems, they are yet to be fully implemented *in vivo*, *in vitro*, or even *in silico*. Thus, practical computations of P systems are driven by silicon-based simulators.

Since P systems were introduced, many simulators have been produced by using different software and hardware technologies [7]. In practice, P system simulations are limited by the physical laws of silicon architectures, which are often inefficient or not suitable when dealing with P system features, such as the exponential workspace creation and massive parallelism. However, in order to improve the efficiency of the simulators, it is necessary to exploit current technologies, leading to solutions in the area of *High Performance Computing*. In this way, many simulators have been developed over highly parallel platforms, including reconfigurable hardware as in FPGAs [15], CPU-based clusters [6], as well as the *Graphical Processing Units (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 (*transition* and *active membrane* P systems). Efficiently simulating SNP systems would thus require new attempts in parallel computing.

GPUs are the leading exemplars of modern high throughput-oriented architectures [9]. Nowadays, these kinds of processors are used for tackling problems where parallelism is prevalent. Indeed, GPUs have been successfully used to speedup many parallel applications. Modern GPUs are not limited only to graphics processing, as the first graphic cards were, since current GPUs can now be used for general purpose computations [10]; they are now multi-core and data-parallel processors [12]. By using GPU computing, also known as GPGPU (General Purpose computing on the GPU), a programmer can achieve with a single GPU, a throughput similar to that of a CPU based cluster [10,17]. Thus, the main *advantages* of using GPUs are their low-cost, low-maintenance and low power consumption relative to conventional parallel clusters and setups, while providing *comparable* or *improved* computational power. Moreover, parallel computing concepts such as hardware abstraction, scaling, and so on are handled efficiently by current GPUs.

Given that SNP systems have already been represented as matrices due to their graph-like properties [16], simulating them in parallel devices such as GPUs

is the next natural step. Matrix algorithms are well known in parallel computing literature, including GPUs [8], due to the highly parallelizable nature of linear algebra computations mapping directly to the data-parallel GPU architecture.

Previously, SNP systems have been faithfully simulated in GPUs using their matrix representation [1]. This simulator combined both the *object oriented programming language* (OOPL) Python (CPU part) and CUDA/C (GPU part) codes, and so it has been improved [2], in performance, by using the PyCUDA [13] library. In this paper we present an extension of [2] in order to simulate SNP systems with more general *regular expressions* associated to their firing rules. This extension allows us to simulate larger and wider varieties of SNP systems in order to test the speedup achieved by the GPU and CPU based simulators.

This paper is organized as follows: Section 2 introduces SNP systems formally, as well as their matrix representation. Section 3 provides background for GPU computing with CUDA. The design of the simulator and simulation results are given in Section 4 and Section 5, respectively. Finally, conclusions, future work, acknowledgements, and references end this paper.

2 Spiking Neural P Systems

In this section the SNP system model is introduced, together with a matrix representation of the model. This representation is the *basis* for the simulation algorithm in this paper. Additionally, the two examples used to test and analyse the simulator are also described.

2.1 The SNP System Model

Many variants of SNP systems have been introduced in recent works, such as those with delays, weights, extended firing rules, deterministic systems, division, budding, and so on. Each one has specific features in complexity, but the majority of them have been shown to be *computationally complete* [11,5]. This paper is focused on a restricted variant of SNP systems, with no delays associated to the rules (i.e. neurons fire immediately once they are able to do so), which are of the following form:

Definition 1. *An SNP system without delay, of degree $m \geq 1$, is a construct of the form*

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

where:

1. $O = \{a\}$ *is the alphabet made up of only one object a, called spike;*
2. $\sigma_1, \ldots, \sigma_m$ *are m neurons of the form*

$$\sigma_i = (n_i, R_i), 1 \leq i \leq m,$$

where:

(a) $n_i \geq 0$ *gives the initial number of spikes (a) contained in neuron* σ_i*;*
(b) R_i *is a finite set of rules of the following forms:*
(b-1) $E/a^c \rightarrow a^p$*, are known as Spiking rules, where* E *is a regular expression over* a*, and* $c \geq 1$*, for* $p \geq 1$ *number of spikes are produced (with the restriction* $c \geq p$*), transmitted to each adjacent neuron with* σ_i *as the originating neuron, and* $a^c \in L(E)$*;*
(b-2) $a^s \rightarrow \lambda$*, are known as Forgetting rules, for* $s \geq 1$*, such that for each rule* $E/a^c \rightarrow a^p$ *of type (b-1) from* R_i*,* $a^s \notin L(E)$*;*

3. $syn = \{(i,j) \,|\, 1 \leq i,j \leq m,\, i \neq j\}$ *are the synapses i.e. connections between neurons;*
4. $in, out \in \{1, 2, \ldots, m\}$ *are the input and output neurons, respectively.*

A *spiking rule* (type (b-1)) $r_s \in R_i$, where $1 \leq i \leq m$, is applied if the corresponding σ_i contains k spikes, $a^k \in L(E)$ and $k \geq c$. This means that *consuming* (removing) c spikes (thus only $k - c$ spikes remain in σ_i), the neuron is fired, producing p spikes that reach all σ_j neurons immediately (no delays) such that $(i,j) \in syn$. An SNP system whose spiking rules have $p = 1$ (they produce only one spike) is said to be of the standard type (*non-extended*). Forgetting rules (type (b-2)) are selected if σ_i contains s spikes. Thus, s spikes are *'forgotten'* or removed from the neuron once the rule is applied. Finally, a special case of (b-1) are rules of type (b-3) where $a^c \rightarrow a$, $L(E) = \{a^c\}$, $k = c, p = 1$.

It is noteworthy that the neurons in an SNP system operate in parallel and in unison, under a *global clock* [11]. Similar to the usual way of using rules in other P system variants, there is maximal parallelism at the level of the system, in the sense that in each step all neurons which can spike (i.e. fire a rule) have to do it. However, only one rule can be applied at a given time in each neuron [11,16]. The *nondeterminism* of SNP systems comes with this fact: $L(E) \cap L(E') \neq \emptyset$ for two different spiking rules with regular expressions E and E', i.e. exactly one rule among several other applicable rules is chosen nondeterministically.

2.2 Examples of SNP Systems

Next, two specific SNP systems are described to show their formal description based on Definition 1. The examples are from [16] and [11], and they are called Π_1 and Π_2 respectively. These examples are used in Section 5 to analyse the performance of the simulator.

The SNP system Π_1 shown in Figure 1 generates all numbers in the set $\mathbb{N}$ - $\{1\}$ (so it doesn't *halt*) and the *outputs* of the computation are derived from the time difference between the *first spike* of the output neuron (to the environment) and its *succeeding spikes*. It can be seen that a total system ordering is given to neurons (from σ_1 to σ_3) and rules (from R_1 to R_5). This SNP system is of the form $\Pi_1 = (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, out)$ where the first neuron $\sigma_1 = (2, \{R_1, R_2\})$, $n_1 = 2$, $R_1 = \{a^2/a \rightarrow a\}$, $R_2 = \{a^2 \rightarrow a\}$, ($\sigma_2$ to σ_3 and their n_is and R_is can be similarly shown), $syn = \{(1,2), (1,3), (2,1), (2,3)\}$ are the synapses for Π_1, and the output neuron $out = \sigma_3$, as can be seen by the arrow not pointing to any neuron. Π_1 has no input neuron.

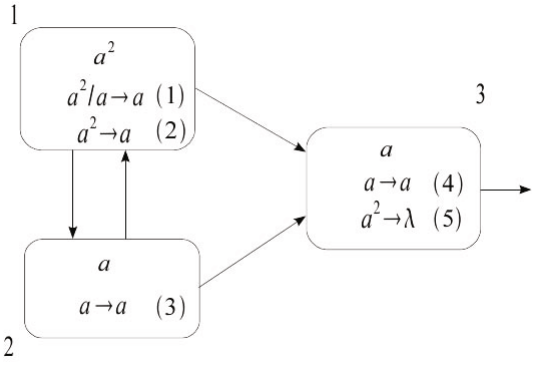

Fig. 1. Π_1, an SNP system generating all numbers in the set $\mathbb{N}$ - {1}, from [16].

A second SNP system Π_2, adapted from Figure 8 of [11], is shown in Figure 2. Π_2 is larger (in terms of the number of neurons and rules) than Π_1 (from Figure 1) and is formally defined as follows:

$\Pi_2 = (\{a\}, \sigma_1, \sigma_{f_1}, \sigma_{f_2}, \sigma_{f_3}, \sigma_{f_4}, \sigma_{f_5}, \sigma_{f_6}, \sigma_{l_h}, \sigma_{out}, syn, out)$

where the σ (neuron) labelling are taken from [11]. Including the total ordering of the rules in Π_2, we have the following:

$\sigma_1 = (0, \{R_1, R_2\})$, $n_1 = 0$ (σ_1 in this case initially has no spikes), $R_1 = a^3(aa)^+/a^2 \to a$, $R_2 = a^3 \to a$,
$\sigma_{f_1} = (0, \{R_3\})$, $n_{f_1} = 0$, $R_3 = \{a \to a\}$,
$\sigma_{f_2} = (0, \{R_4\})$, $n_{f_2} = 0$, $R_4 = \{a \to a\}$,
$\sigma_{f_3} = (0, \{R_5, R_6\})$, $n_{f_3} = 0$, $R_5 = \{a^2 \to a\}, R_6 = \{a \to \lambda\}$,
$\sigma_{f_4} = (0, \{R_7, R_8\})$, $n_{f_4} = 0$, $R_7 = \{a^2 \to a\}, R_8 = \{a \to \lambda\}$,
$\sigma_{f_5} = (0, \{R_9\})$, $n_{f_5} = 0$, $R_9 = \{a \to a\}$,
$\sigma_{f_6} = (0, \{R_{10}, R_{11}\})$, $n_{f_6} = 0$, $R_{10} = \{a^3 \to \lambda\}, R_{11} = \{a^2 \to 2\}$,
$\sigma_{l_h} = (0, \{R_{12}, R_{13}\})$, $n_{l_h} = 0$, $R_{12} = \{a^2 \to a\}, R_{13} = \{a \to \lambda\}$,
$\sigma_{out} = (0, \{R_{14}\})$, $n_{out} = 0$, $R_{14} = \{a \to a\}$

As with Π_1, it is evident that for Π_2 we have 17 synapses connecting the neurons (σs). Π_2 doesn't have an input σ but it has $out = \sigma_{out}$.

2.3 Matrix Representation of SNP Systems

In [16], a *matrix representation* of SNP systems without delays was introduced. By using this algebraic representation, it is easy to describe a simulation algorithm for computing configurations in SNP systems without delays by only using vector-matrix operations. The matrix representation inherently and conveniently involves another SNP system variant, the variant with *extended firing rules* where neurons can fire more than one spike at a given time [5]. Another convenient use for this algebraic representation is the possibility of

performing *backwards computation*, from the current configuration to an earlier configuration. This representation makes use of the following vectors and matrix definitions:

Configuration vector C_k is the vector containing all spikes in every neuron on the kth computation step/time, where C_0 is the initial vector containing all spikes in the system at the beginning of the computation. For Π_1 (the example in Figure 1) $C_0 =< 2, 1, 1 >$. For Π_2 we don't have a default C_0 so we will assign several later, for the purpose of simulation. For this work, since we are not simulating SNP systems with delays, the steps focused on are purely computational and not time steps.

Spiking vector S_k shows, at a given configuration C_k, if a rule is applicable (having value *1*) or not (having value *0* instead). For Π_1 we have a spiking vector $S_k =< 1, 0, 1, 1, 0 >$ given C_0. Note that a second spiking vector, $S'_k =< 0, 1, 1, 1, 0 >$, is possible and *valid* if we use rule (2) over rule (1) instead (but not both at the same time). *Validity* in this case means that only one among several applicable rules is used and thus represented in the spiking vector. We recall the applicability of rules from the definitions of (b-1) and (b-3). It is worth mentioning that there are several S_ks for every possible rule selection in the system. We cannot have an $S''_k =< 1, 1, 1, 1, 0 >$ because we cannot use rule (1) and rule (2) at the same time, hence this S_k is invalid.

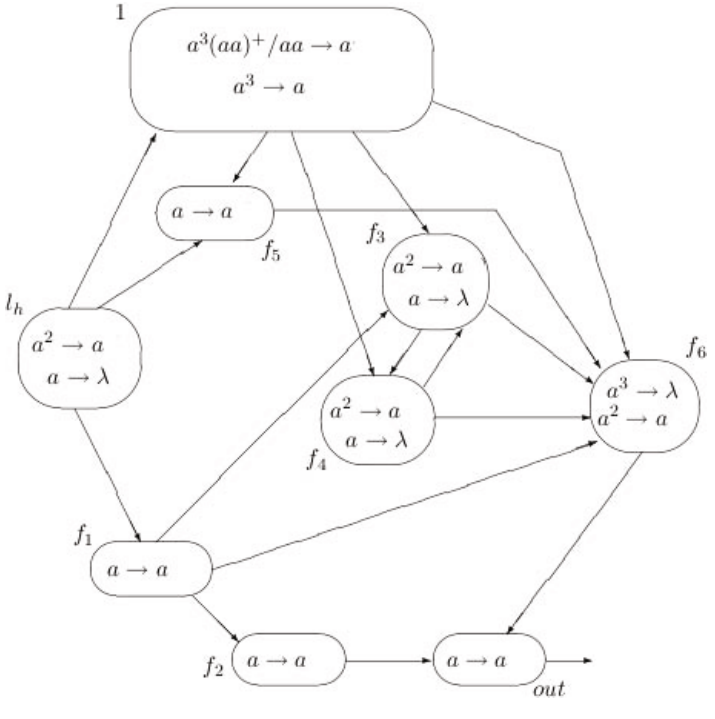

Fig. 2. Π_2, a larger SNP system with 9 neurons and 14 rules, adapted from [11]

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

Definition 2

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

In such a scheme, rows represent rules and columns represent neurons. As mentioned earlier, SNP systems with extended firing rules can be also represented by the spiking transition matrix.

A negative entry in the spiking transition matrix corresponds to the consumption of spikes. Thus, it is easy to observe that each row has exactly one negative entry, and each column has at least one negative entry [16]. For Π_1 and Π_2, the spiking transition matrices M_{Π_1} and M_{Π_2} are shown in the equations (1) and (2), respectively.

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

$$M_{\Pi_2} = \begin{pmatrix} -2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ -3 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -2 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \tag{2}$$

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

$$C_{k+1} = C_k \ + \ S_k \cdot M_{\Pi} \tag{3}$$

3 GPU Computing

High Performance Computing provides solutions for improving the performance of software applications by using *accelerators* or many-core processors. In this respect, *Graphics Processing Units (GPUs)* have been consolidated as accelerators thanks to their throughput-oriented highly-parallel architecture [9], as well as their low-power consumption and low-cost compared to other parallel clusters and setups. At the moment, for around $500, the latest GPUs of NVIDIA with 512 cores and with a performance comparable to a *cluster* of multi-core CPUs, is readily available at consumer electronics stores. The programming and architectural aspects of GPUs are described in this section.

3.1 Compute Unified Device Architecture (CUDA)

As many-core based platforms, GPUs are massively parallel processors which have high *chip scalability* in terms of processing units (core), and high bandwidth with internal GPU memories. The common CPU architectures are composed of transistors associated with different computing tasks: 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 devoted for ALU [12]. 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.

The programmability of GPUs has been focused on graphics, but using a type of parallel computing technique called *GPGPU* (General Purpose computation on GPUs), the large amount of internal cores can be used in parallel for accelerating the execution of data-parallel algorithms. In order to provide a straightforward, easy-to-learn, and scalable programming framework for GPUs, NVIDIA corporation (a well known manufacturer of graphics processors) introduced the *Compute Unified Device Architecture (CUDA)* in 2007 [12]. CUDA is a programming model and hardware architecture for general purpose computations in NVIDIA's GPUs (G80 and newer family of GPUs)[12]. By extending popular languages such as *C*, CUDA allows programmers to easily create software that will be executed in parallel, avoiding low-level graphics and hardware primitives[17]. Among the other benefits of CUDA are *abstracted* and *automated scaling*: GPUs with more cores will make the parallelized code run faster than GPUs with fewer cores [17].

As seen in Figure 3, CUDA implements a *heterogeneous computing* architecture, where two different parts are often considered: the *host* (CPU side) and the *device* (GPU side). The host/CPU part of the code is responsible for controlling the program execution flow, allocating memory in the host or device/GPU, and obtaining results from the device by executing specific codes. The device (or devices if there are several GPUs in the setup) acts as a parallel *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.

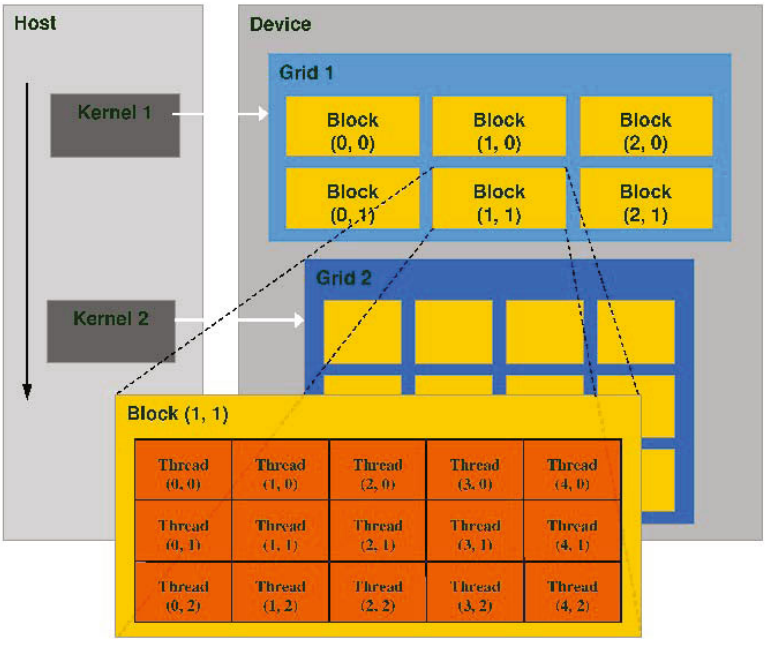

Fig. 3. Structure of the CUDA programming model, from [4]

The code to be executed in a GPU is written in CUDA C (CUDA *extended* ANSI C programming language). The parallel distribution of the execution units (threads) in CUDA can be split up into multiple threads within multiple thread blocks, each contained within a grid of (thread) blocks (see Figure 3). These grids belong to a single device/single GPU. Each device has multiple cores, each capable of running its own *block of threads* [12,17]. A function known as a *kernel function* is one that is called from the host but executed in the device. Using *kernel functions*, the programmer can specify the GPU resources: the layout of the threads (from one to three dimensions) and the blocks (from one to two dimensions). GPUs with the same architecture as the one used in this work has a maximum number of threads per block equal to 512. The maximum size of each dimension of a thread block is $(512 \times 512 \times 64)$, pertaining to the *x,y,* and *z* dimensions of a block respectively. Lastly, the maximum size of each dimension of a grid of thread block is $(65535 \times 65535 \times 1)$ for the grid's *x,y,* and *z* dimensions.

3.2 SNP System GPU Simulation Considerations

To successfully simulate SNP systems, the input files are file versions of C_k, S_k, M_{SNP}, and a file r containing the list of rules R_i. Since SNP systems involve

regular expressions (similar to other P system variants), *string* manipulation of the number of spikes in a neuron satisfying the regular expression E is involved. An OOPL such as Python is very well suited for string manipulation. For the computations involving linear algebra, since C_k, S_k and M_{SNP} only have integral values, the C programming language (which NVIDIA extended for their purposes as CUDA C) is well suited. Another reason for using C is because of this C language extension of CUDA. In actuality, only the kernel functions are written in C, and those functions are *embedded* within the Python code. A *row-major* ordering, a well known linear array input format where rows are placed one after another from top to bottom, was used to input the elements of M_{SNP}. As an example, row-major ordering of M_{Π_1} is: 1, 1, 1, 2, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 2.

The current *dichotomy* of the CUDA programming model is that the data or inputs are loaded and 'prepared' in the host part, then they are moved to the device part and the parallel part of the process is executed on the data. Finally, the results of the parallel processing are moved back to the host for further processing. As seen from Figure 3 kernel functions are called from the host sequentially but once executed in the device, their blocks and the threads within these blocks are executed in parallel.

PyCUDA was chosen in order to fully utilize the speedup of CUDA as well as minimize development time, and is a Python programming interface to CUDA [13]. PyCUDA was developed by mathematician Andreas Klöckner for his dissertation for a more efficient parallel computing on CUDA using Python: safer in terms of memory handling, object cleanup (among others), and faster in terms of development time via abstractions [13].

4 Simulating SNP Systems in CUDA

In this paper we designate as *snpgpu-sim3* our improved simulator, as compared to *snpgpu-sim2* from [2]. The simulation algorithm for *snpgpu-sim3*, as with *snpgpu-sim2*, is shown in Algorithm 1. Algorithm 1 shows which part of the algorithm (and hence the simulation) is run: whether in the host part or the device part, or both.

Firstly, the simulation inputs are loaded as mentioned in Subsection 3.2, as white space separated files (for C_k, S_k, and M) and using delimiters '@' and '&' in r to delineate one rule from another in the same neuron, and from one neuron to another, respectively. For example, the file r pertaining to the rules of Π_1 contains: *aa 1 1@aa 2 1&a 1 1&a 1 1@aa 1 0*. R_1 of Π_1 pertains to *aa 1 1*. In this encoding, the first part is the regular expression E (in this case, a^2 or aa), the middle part is the number of spikes consumed (one spike) and lastly, the number of spikes produced (again, one spike).

In part II, the number of spikes in a neuron are checked if they satisfy the regular expression E. In *snpgpu-sim2* only rules of the form (b-3), and not (b-1), were simulated, thus *snpgpu-sim3* can simulate more general SNP systems. A function created in Python, *chkRegExp(regexp, spikNum)* returns a boolean value of *True* if and only if the number of spikes given by *spikNum* (and hence

the number of spikes in a σ) are in the language generated by the regular expression *regexp*. Otherwise, function *chkRegExp* returns a boolean *False*. Part II is responsible for generating all possible and *valid* S_ks out of the current C_ks and the rule file r. Note that not all possible S_ks (strings of 1s and 0s) are valid, since exactly one rule only, chosen nondeterministically, is applied per neuron per time step. The simulation 'implements' the nondeterminism (as nondeterminism is yet to be fully realized in hardware) by producing all the possible and valid S_ks for the given C_ks, and proceeds to compute each of the C_{k+1} from these.

The process by which all possible and valid S_ks are produced is as follows: Once all the rules in the system are identified, given the current n_is (number of spikes present in each of the σ_is), the {*1,0*} strings (at the moment they are treated as strings, and then as integral values later on) are produced on a per neuron level. As an example, given that $n_1 = 2$ for Π_1, and its 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 '1' (R_3 of σ_2 has the needed single spike, 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, from left to right (since there is a need for ordering), until finally all the valid and possible S_ks from the current C_ks are produced. For Π_1, given $C_0 =< 2, 1, 1 >$ we have S_ks *(1,0,1,1,0)* and *(0,1,1,1,0)*.

Part III performs Equation 3, which is done in parallel in the device. The previously loaded values of C_k, S_k, r, and M which were treated as *strings* (for the purposes of concatenation, regular expression checking, among others) are now treated as integral values. Each thread in the device contains either a matrix element from M or a vector element from S_k or C_k, and Equation 3 is performed in parallel. The newly produced C_{k+1} are then moved back from the device to the host. Part IV then checks whether to proceed or to stop based on 2 *stopping criteria* for the simulation: (I) if a *zero vector* (vector of zeros) is encountered, (II) if the succeeding C_ks have all been produced in previous computations. Both (I) and (II) make sure that the simulation *halts* and does not enter an *infinite loop*.

Algorithm 1. Overview of SNP system simulation algorithm

Require: Input files: Ck, M, r (file counterparts of C_k, M, R_i).

I. **(HOST)** Load input files. M, r are loaded once only. $C0$ is also loaded once, then Cks, $1 \leq k \leq m$, afterwards.

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

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

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

In Figure 4 we see a graphical illustration of the simulation process, emphasizing the parts executed in the host/CPU and in the device/GPU. In the figure, it is assumed that there are n number of C_{k+1}s produced from the current C_k. The CPU executes the simulation from top to bottom, calling the kernel function, and hence the GPU, in the third box. The smaller, multiple boxes in the device/GPU part illustrate the parallel computations of all the C_{k+1}s using Equation 3. Afterwards, the computed C_{k+1}s are sent back to the CPU which then decides what to do next, based on Algorithm 1, part IV.

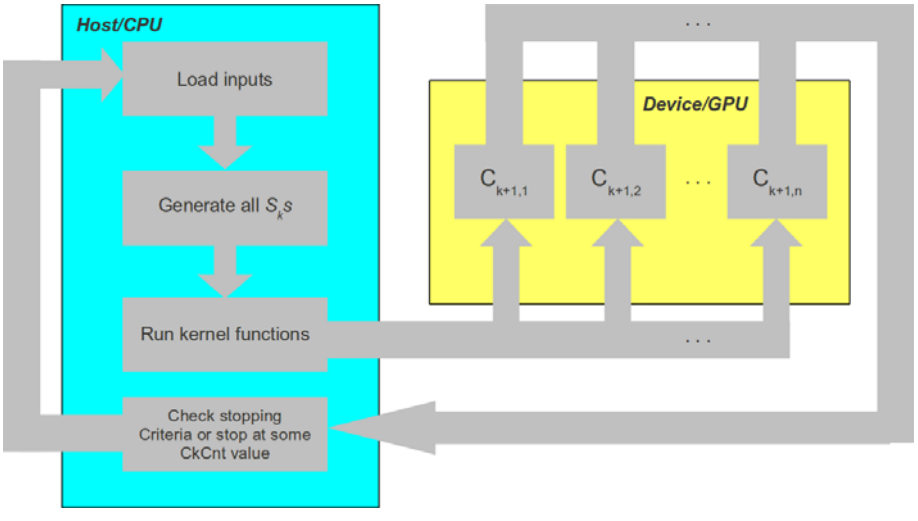

Fig. 4. Diagram showing the simulation flow, with the host and device emphasized

5 Simulation Results and Observations

Similar to the setup of *snpgpu-sim2*, *snpgpu-sim3* simulated Π_1 and Π_2 using an Apple iMac running Mac OS X 10.5.8, with an Intel Core2Duo CPU with 2 cores at 2.66GHz (maximum clock speed) per core, and with a 6MB L2 cache having 4GB of RAM. The GPU of the iMac is an NVIDIA GeForce 9400 graphics card at 1.15 GHz, with 256 MB Video RAM (or around 266×10^6 bytes), 16 cores, running CUDA version 3.1.

In order to compare the CPU-only SNP system simulator (we designate this as *snpcpu-sim*) the parallel parts of *snpgpu-sim3* were executed in a sequential manner. Hence, *snpcpu-sim* runs entirely on the CPU only, while *snpgpu-sim3* uses both the CPU and the GPU of the iMac. Both simulators use Python and C, however only *snpgpu-sim3* use the CUDA enabled GPU. The simulations were done such that both *snpcpu-sim* and *snpgpu-sim3* have the same C_0 or starting configuration as inputs. The simulations are run three times and the average of the three trial runs is taken. The simulation comparison for Π_1 run in both simulators is shown in Figure 5.

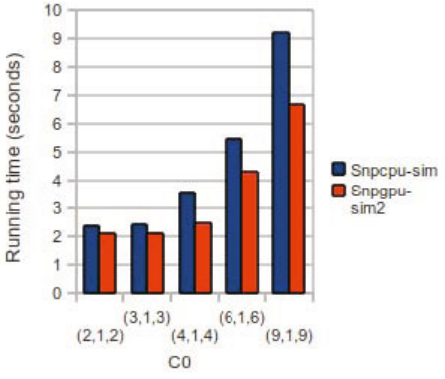

Fig. 5. Runtime graph of *snpcpu-sim* versus *snpgpu-sim3* for Π_1

From Figure 5, the five C_0 values used in the simulation comparison are: (2,1,2), (3,1,3), (4,1,4), (6,1,6), and (9,1,9). The horizontal axis in Figure 5 are the C_0 values, while the vertical axis is the running time (in seconds) of the simulation on a given C_0. We see in Figure 5 that *snpgpu-sim3* performs faster than *snpcpu-sim* given increasing values of C_0. A speedup of up to *1.4 times* is achieved if Π_1 is run in *snpgpu-sim3* instead of *snpcpu-sim*.

For Π_2, we introduce a variable $CkCnt$ in both simulators to limit the number of C_ks produced to a certain integer value so the simulation will not run indefinitely. Figure 6 shows the simulation of Π_2 for different values of n (the number of spikes in σ_1 of Π_2 is $2n$) and $CkCnt$. In particular, the *(n, CkCnt)* pairs used are (5, 20), (10, 25), (15, 30), (20, 35), (50, 45), (70, 50), (100, 55), and (200, 60).

From Figure 6 we see that, for increasing values of n and $CkCnt$, the simulation time of Π_2 in *snpcpu-sim* increases dramatically, unlike in *snpgpu-sim3*. The lack of a dramatic increase in simulation time in *snpgpu-sim3* is because of the fact that Π_2 has more rules and neurons, and hence, exploits the parallel nature of the CUDA GPU all the more. A speedup of up to *6.8 times* is achieved from the simulation on *snpgpu-sim3* over *snpcpu-sim*.

We can calculate the maximum number of neurons *snpgpu-sim3* can simulate using the 266×10^6 bytes of the NVIDIA GeForce 9400 GPU. Global memory used is $GbMem = 4 \times sizeof(Ck) + sizeof(M)$. Here we speak in terms of the $sizeof$ operator in the standard C language since we use the data type *int* which is of size *4 bytes or 32 bits*. The length of Ck is the number of neurons, and M is of size R (rows are the number of rules) $\times$ Ck (columns are number of neurons). We multiply Ck to 4 because we also alocate memory for Sk, $Ck+1$, and MSk which temporarily holds the product of M and Sk, all of which are of the same size as Ck. We therefore have the following, substituting the real value of $GbMem$: $266 \times 10^6(bytes) = 16(bytes) \times Ck + 4(bytes) \times R \times Ck$. Simplifying this equation and isolating Ck we have $Ck = 266 \times 10^6(bytes)/(16(bytes) + 4(bytes) \times R)$ i.e. the number of neurons that can be simulated is a function of the number of rules in the system.

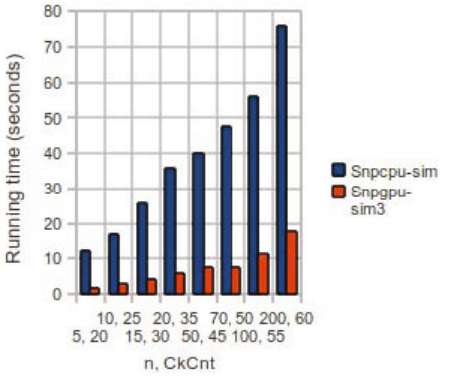

Fig. 6. Runtime graph of *snpcpu-sim* versus *snpgpu-sim3* for Π_2

Additionally, aside from the iMac setup mentioned above (setup 1), a second setup was used to provide further insights of the current simulator implementation. Setup 2 consisted of two Intel Xeon E5504 processors, each having 4 cores (2 GHz max per core), an L2 cache of 1MB and an additional L3 cache of 4MB. The GPU is an NVIDIA Tesla C1060 running at 1.3 GHz with 4GB of Video RAM and having 240 cores. Setup 2 has 64bit Ubuntu 10.04.1 Linux OS, and effectively has 8 cores: 2×4 core Intel Xeon processor. The first 4 cores of processor 1 are labeled core 0 to 3, and the next 4 cores of processor 2 are labeled core 4 to 7.

Using the Linux command *taskset*, *snpgpu-sim3* and *snpcpu-sim* were run for both Π_1 and Π_2 in setup 2. The *taskset* command allows the setting of the CPU affinity of a process, thus "bonding" the process to a specified number of processor cores alone. For both *snpcpu-sim* and *snpgpu-sim3*, tests were done to set the simulation affinity to 2, 4, 8 cores, and a natural affinity setting (this is the default in the Linux OS). For simulations with 2 and 4-core affinities, the cores chosen are from the same processor i.e. either between cores 0 to 3, or 4 to 7, but not cores 3 and 4 together (for 2-core affinity) or 2,3,4, and 5 together (for 4-core affinity), since these core selections come from different processors and simulations will incur performance penalties. From the setup 2 tests (and for both *snpcpu-sim* and *snpgpu-sim3*), simulations done with a 2-core affinity executed in less time than with 4-core affinity simulations. Similarly, and again for both *snpcpu-sim* and *snpgpu-sim3*, 4-core affinity simulations executed in less time than 8-core affinity simulations. The 8-core affinity simulations were practically indistinguishable from the default (natural) affinity by the Linux OS.

6 Conclusions and Future Work

In this paper we have simulated a wider variety of SNP systems using *snpgpu-sim3*, while maintaining the efficiency of the previous simulator *snpgpu-sim2*.

The extension of the previous simulator *snpgpu-sim2*, presented here as *snpgpu-sim3*, can now simulate larger and wider varities of SNP systems by way of more general regular expressions (those of the form (b-1)). Two SNP systems were simulated: a basic one, Π_1 (primarily for illustration purposes of the simulator) and a larger one (in terms of rules and neurons), Π_2, to exemplify the speedup when using GPUs over CPU-only simulators. The speedup of *snpgpu-sim3* over *snpcpu-sim* for Π_1 went up to 1.4 times, while it was 6.8 for Π_2. These results show that SNP system simulation on GPUs can greatly benefit from the parallel architecture of GPUs, and that increasing the parameters (in this case neurons and rules) offer even larger speedups. This benefit in speedup is coupled with the fact that the CUDA enabled graphics cards are readily available in consumer electronic stores. These cards offer boosts in general purpose computations as co-processors of commonly used CPUs, at a fraction of the power consumption of CPU clusters.

From setup 2 which is a "larger" setup compared to setup 1, we see that for "small" systems e.g. Π_1, the CPU can outperform the GPU for *snpgpu-sim3*. Setup 2 helped validate and support the performance results from setup 1 using *snpgpu-sim3*. The current simulation in Algorithm 1 is thus recommended for up to 4 cores on a single processor only, as the CPU part is not multi-threaded (only the GPU part is). Using more than 4 cores on a single processor, or more than one core from more than one processor in a multiprocessor setup will not yield better performance. Alternatively, simulating "larger" SNP systems can capitalize on the massively parallel architecture of GPUs compared to multi-core CPUs, and thus increase the speedup even further.

For future work, a generic P system parser based using *P-lingua* would be able to provide a more standardized input parser and formatting for the GPU based SNP system simulator. P-lingua is programming environment for Membrane computing which takes in as input a certain P system variant, and outputs a standardized XML file format for the simulator to use. This standardized simulator input format would then be carried on to other simualtors when simulating other P system variants.

Further understanding of the CUDA architecture as well as adapting the simulator to GPUs with more cores and their newer parallel technologies, as well as their subtleties, are also for future work. Increasing the level of simulation parallelism is also part of this work, to gain even higher performance speedups over CPU only simulations.

More SNP system variants can be simulated by extending the current GPU simulator (with neuron budding, with delays et al). Lastly, the use of the simulator to empirically test the models of problems solved with SNP systems is also a future work.

Acknowledgments. Francis George Cabarle is supported by the DOST-ERDT program. Henry Adorna is funded by the DOST-ERDT research grant and the Alexan professorial chair of the UP Diliman Department of Computer Science.

Miguel A. Martínez–del–Amor is supported by "Proyecto de Excelencia con Investigador de Reconocida Valía" of the "Junta de Andalucía" under grant P08-TIC04200, and by the project TIN2009–13192 of the "Ministerio de Ciencia e Innovación" of Spain, both co-financed by FEDER funds.

References

1. Cabarle, F., Adorna, H., Martínez-del-Amor, M.A.: Simulating Spiking Neural P systems without delays using GPUs. In: Proceedings of the 9th Brainstorming Week on Membrane Computing, Sevilla, Spain (February 2011)
2. Cabarle, F., Adorna, H., Martínez-del-Amor, M.A.: An Improved GPU Simulator For Spiking Neural P Systems. In: IEEE Sixth International Conference on Bio-Inspired Computing: Theories and Applications, Penang, Malaysia (September 2011), doi:10.1109/BIC-TA.2011.37
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), 313–322 (2010)
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), 317–325 (2010)
5. Chen, H., Ionescu, M., Ishdorj, T.-O., Păun, A., Păun, G., Pérez-Jiménez, M.: Spiking neural P systems with extended rules: universality and languages. Natural Computing: an International Journal 7(2), 147–166 (2008)
6. Ciobanu, G., Wenyuan, G.: P Systems Running on a Cluster of Computers. In: Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2003. LNCS, vol. 2933, pp. 123–139. Springer, Heidelberg (2004)
7. Díaz, D., Graciani, C., Gutiérrez, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Software for P systems. In: Păun, G., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing, ch. 17, pp. 437–454. Oxford University Press, Oxford (2009)
8. 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 2004), pp. 133–137. ACM, NY (2004)
9. Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. Communications of the ACM 53(11), 58–66 (2010)
10. Harris, M.: Mapping computational concepts to GPUs. In: ACM SIGGRAPH 2005 Courses, NY, USA (2005)
11. Ionescu, M., Păun, G., Yokomori, T.: Spiking Neural P Systems. Journal Fundamenta Informaticae 71(2,3), 279–308 (2006)
12. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands on Approach, 1st edn. Morgan Kaufmann, MA, USA (2010)
13. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA: GPU Run-Time Code Generation for High-Performance Computing, no. 2009-40. Scientific Computing Group, Brown University, RI, USA (2009)
14. Păun, G., Ciobanu, G., Pérez-Jiménez, M. (eds.): Applications of Membrane Computing. Natural Computing Series. Springer, Heidelberg (2006)

15. Nguyen, V., Kearney, D., Gioiosa, G.: A Region-Oriented Hardware Implementation for Membrane Computing Applications. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) WMC 2009. LNCS, vol. 5957, pp. 385–409. Springer, Heidelberg (2010)
16. Zeng, X., Adorna, H., Martínez-del-Amor, M.Á., Pan, L., Pérez-Jiménez, M.J.: Matrix Representation of Spiking Neural P Systems. In: Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., Salomaa, A. (eds.) CMC 2010. LNCS, vol. 6501, pp. 377–391. Springer, Heidelberg (2010)
17. NVIDIA corporation, NVIDIA CUDA C programming guide, version 3.0. NVIDIA, CA, USA (2010)