Spiking Neural P system without delay simulator implementation using GPGPUs

Francis Cabarle
Algorithms and Complexity Laboratory
Department of Computer Science
University of the Philippines Diliman
fccabarle@up.edu.ph

Henry Adorna
Algorithms and Complexity Laboratory
Department of Computer Science
University of the Philippines Diliman
hnadorna@dcs.upd.edu.ph

ABSTRACT

This paper presents a parallel simulator for a type of P system known as spiking neural P system (SNP system) using general purpose graphics processing units (GPGPUs). GPGPUs, unlike the more conventional and general purpose, multi-core CPUs, are used for parallelizable problems due to their architectural optimization for parallel computations.

Membrane computing or P systems on the other hand, are cell-inspired computational models which compute in a maximally parallel and non-deterministic manner. SNP systems, w/c compute via time separated spikes and whose inspiration was taken from the way neurons operate in living organisms, have been represented as matrices.

The matrix representation of SNP systems provides a crucial step into their simulation on parallel devices such as GPGPUs. Simulating the highly parallel nature of SNP systems necessitates the use of hardware intended for parallel computations. The simulator algorithms, design considerations, and implementation are presented. Finally, simulation results, observations, and analyses using an SNP system that generates all numbers in $\{\mathbb{N} - 1\}$ are discussed.

Keywords

Membrane computing, Parallel computing, GPU computing

1. INTRODUCTION

1.1 Parallel computing: Via graphics processing units (GPUs)

The trend for massively parallel computation is moving from the more common multi-core CPUs towards GPGPUs for several significant reasons [7][8]. One important reason for such a trend in recent years include the low consumption in terms of power of GPGPUs compared to setting up machines and infrastructure which will utilize multiple CPUs in order to obtain the same level of parallelization and performance[9]. Another more important reason is that GPGPUs are architectured for massively parallel computations since unlike the architectures of most general purpose CPUs, a large part of GPGPUs are devoted for arithmetic operations and not on control and caching [7][8]. Arithmetic operations are at the heart of many basic operations as well as scientific computations, and these are performed with larger speedups when done in parallel, by GPGPUs over CPUs.

1.2 Paralel computing: Via Membranes

Membrane computing or its more specific counterpart, a P system, are Turing complete computing models that perform computations nondeterministically, exhausting all possible computations at any given time. This type of unconventional model of computation was introduced by Gheorghe Păun in 1998 and takes inspiration, similar to other members of natural computing, from nature[4][5]. Specifically, P systems try to mimic the constitution and dynamics of the living cell: the multitude of elements inside it, and their interactions within themselves and their environment, or outside the cell's skin membrane.

SN P systems differ from other types of P systems precisely because they are mono-membranar and only use one type of object in its computation. These characteristics, among others, are meant to capture the workings of a special type of cell known as the neuron. Neurons, such as those in the human brain, communicate or 'compute' by sending indistinct electro-chemical signals more commonly known as spikes. Information is then communicated and encoded not by the spikes themselves, since the spikes are unrecognizable from one another, by means of time duration as well as the number of spikes sent/received from one neuron to another, oftentimes under a certain time interval[1]. The time duration between two spikes, or several successive spikes, transmit information from one cell to another.

It has been shown that SN P systems, given their nature, are representable by matrices[2][3]. This representation allows design and implementation of an SN P system simulator using parallel computing machines such as GPGPUs.

1.3 Simulating SNP systems in GPGPUs

Matrix operations and their algorithms have been studied and efficiently implemented in GPGPUs [10][11]. Thus the matrix representation of SNP systems bridges the gap between the highly theoretical yet still computationally pow-

erful SNP systems and the applicative and more tangible GPGPUs, via an SNP system simulator. The design of the simulator, including the algorithms deviced, architectural considerations, are then implemented using a particular type of GPGPU, namely NVIDIA CUDA (compute unified device architecture). NVIDIA CUDA extends the widely known ANSI C programming language and makes parallel computations, via GPGPUs manufactured by NVIDIA.

This paper starts out by introducing and defining the type of SNP system that will be simulated. Afterwards the NVIDIA CUDA model and architecture are discussed, baring the scalability and parallelization CUDA offers. Next, the design of the simulator, constraints and considerations, as well as the details of the algorithms used to realize the SNP system are discussed. The simulation results are presented next, as well as observations and analysis of these results. The paper ends by providing the conclusions and future work.

2. SPIKING NEURAL P SYSTEMS

2.1 Computing with SNP systems

The type of SNP systems focused on by this paper are those without delays i.e. those that spike or transmit signals the moment they are able to do so [2][3]. A variant, which allows for delays before a neuron produces a spike, are also available [1]. An SNP system without delay is 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, the system spike a.
2. $\sigma_1, \ldots, \sigma_m$ are m number of neurons of the form
$$\sigma_i = (n_i, R_i), 1 \leq i \leq m,$$
where:
   a) $n_i \geq 0$ gives the initial number of as i.e. spikes contained in neuron σ_i
   b) R_i is a finite set of rules of with two forms:
      (b-1) $E/a^c \rightarrow a$, are known as *Spiking rules*, where E is a regular expression over a, and $c \geq 1$, such that $c \geq 1$.
      (b-2) $a^s \rightarrow \lambda$, are known as *Forgetting rules*, for $s \geq 1$, such that for each rule $E/a^c \rightarrow a$ 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. connection between neurons.
4. $in, out \in \{1, 2, \ldots, m\}$ are the input and output neurons, respectively.

Furthermore, rules of type (b-1) are applied if σ_i contains k spikes, $a^k \in L(E)$ and $k \geq c$. Using this type of rule uses up or consumes k spikes from the neuron, producing a spike to each of the neurons connected to it via a forward pointing arrow i.e. away from the neuron. In this manner, for rules of type (b-2) if σ_i contains s spikes, then s spikes are forgotten or removed once the rule is used. Rules of type (b-1) can be simplified with the notation

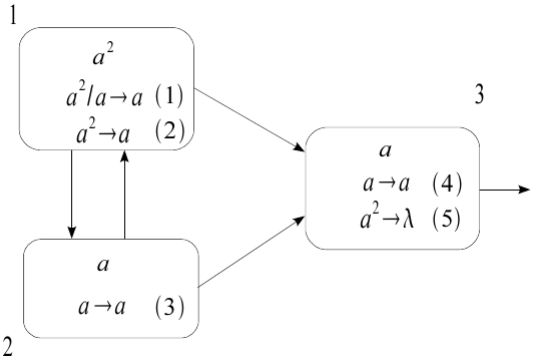

Figure 1: An SNP P system Π, generating all numbers in {ℕ - 1}, from [3].

(b-3) $a^k \rightarrow a$

where the regular expression $E = a^k$, again consuming k spikes and producing a spike.

The non-determinism of SNP systems comes with the fact that more than one rule of the several types are applicable at a given time, given enough spikes. The rule to be used is chosen non-deterministically in the neuron. However, only one rule can be applied or used at a given time [1][2][3]. The neurons in an SN P system operate in parallel and in unison, under a global clock [1]. For Figure 1 no input neuron is present, but neuron 3 is the output neuron, hence the arrow pointing towards the environment, outside the SNP system. The SNP system in Figure 1 is a 3 neuron system whose neurons are labeled (neuron/σ_1 to 3) and whose rules have a total system ordering from (1) to (5)

2.2 Matrix representation of SNP systems

A matrix representation of an SN P system makes use of the following vectors and matrix definitions [2][3] . It is important to note that, just as in Figure 1, a total ordering of rules is in order.

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 (in Figure 1) the initial configuration vector is $C_0 =< 2, 1, 1 >$.

Spiking vector which shows, at a given configuration C_k, if a rule is applicable (has value 1) or not (has value 0 instead). For Π we have the spiking vector $S_k =< 1, 0, 1, 1, 0 >$ given C_0. Note that a 2nd spiking vector, $S_k =< 1, 0, 1, 1, 0 >$, is possible if we use rule (2) over rule (1) instead (but not both at the same time, hence we cannot have a vector equal to < 1, 1, 1, 1, 0>).

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

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

For Π, the M_Π is as follows:

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

In such a scheme, rows represent rules and columns represent neurons.

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

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

3. THE NVIDIA CUDA ARCHITECTURE

NVIDIA, a well known manufacturer of GPUs, released in 2006 the CUDA programming model and architecture [9]. Using extensions of the widely known C language, a programmer can write parallel code which will then execute in multiple threads within multiple thread blocks, each contained within a grid of (thread) blocks. These grids belong to a single device i.e. a single GPGPU. Each device/GPGPU has multiple cores, each capable of running its own grids. The program run in the CUDA model scales up or down, depending on the number of cores the programmer currently has in a device. This scaling is done in a manner that is abstracted from the user, and is efficiently as well. Automatic and efficient scaling is shown in Figure 2 Parallelized code will run faster with more cores than with fewer ones [8].

Figure 3 shows another important feature of the CUDA model: the host and the device parts. As mentioned earlier, device pertains to the GPGPU/s of the system, while the host pertains to the CPU/s. A function known as a kernel function, is a function called from the host but executed in the device.

A general model for creating a CUDA enabled program is shown in Listing 1.

Listing 1: General code flow for CUDA programming

```
1 //allocate memory on GPU e.g.
2 cudaMalloc( (void**)&dev_a, N * sizeof(int)
3
4 //populate arrays
5  . . .
6
7 //copy arrays from host to device e.g.
```

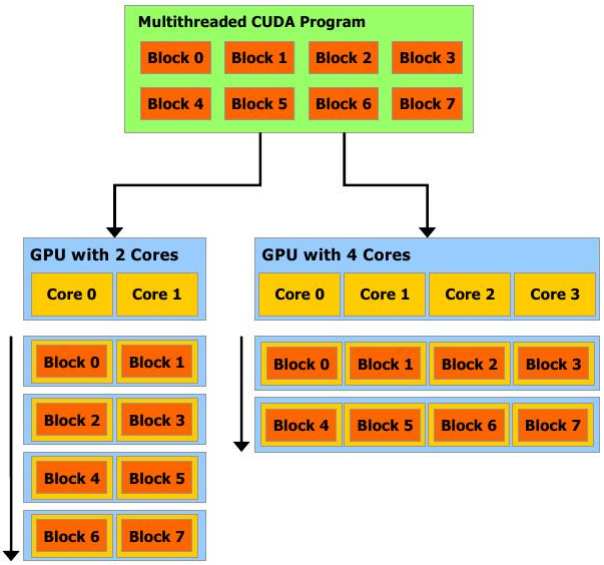

Figure 2: NVIDIA CUDA automatic scaling, hence more cores result to faster execution, from [8].

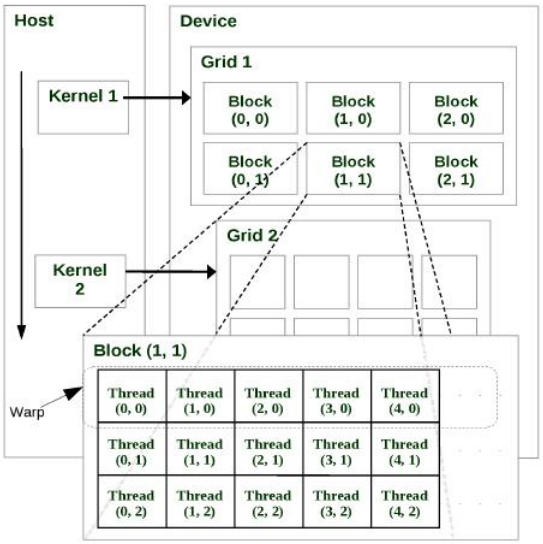

Figure 3: NVIDIA CUDA programming model showing the sequential execution of the *host* code alongside the parallel execution of the *kernel* function on the *device* side, from [6].

```
8 cudaMemcpy( dev_a, a, N * sizeof(int),
9 cudaMemcpyHostToDevice)
10
11 //call kernel (GPU) function e.g.
12 add<<<N, 1>>>( dev_a, dev_b, dev_c );
13
14 // copy arrays from device to host e.g.
15 cudaMemcpy( c, dev_c, N * sizeof( int),
16 cudaMemcpyDeviceToHost )
17
18 //display results
19
20 //free memory e.g.
21 cudaFree( dev_a );
```

Lines 2 and 19, implement CUDA versions of the standard C language functions e.g. the standard C function *malloc* has the CUDA C function counterpart being *cudaMalloc*, and the standard C function $free$ has *cudaFree* as its CUDA C counterpart.

Lines 8 and 15 show a CUDA C specific function, namely *cudaMemcpy*, which, given an input of pointers (from Listing 1 host code pointers are single letter variables such as a and c,while device code variable counterparts are prefixed by *dev_* such as *dev_a* and *dev_c*) and the size to copy (as computed by the *sizeof* function), moves data from host to device (parameter $cudaMemcpyHostToDevoce$) or device to host (parameter $cudaMemcpyDeviceToHost$).

A kernel function call uses the double < and > operator, in this case the kernel function

$add << N, 1 >> (dev\_a, dev\_b, dev\_c)$.

This function adds the values, per element (and each element is associated to 1 thread), of the variables dev_a and dev_b sent to the device, collected in variable dev_c before being sent back to the host/CPU. The variable N in this case allows the programmer to specify N number of threads which will execute the *add* kernel function in parallel, with 1 specifying only one block for all N threads.

3.1 Design considerations given hardware and software setup

The kernel function i.e. code that is executed in parallel in the device, needs to have its results initially moved from the CPU/host to the device, and then back from the device to the host after computation. This movement of data back and forth should be minimized in order to obtain more efficient, in terms of time, execution. Implementing an equation such as 2, which involves multiplication and addition between vectors and a matrix, can be done in parallel with the previous considerations in mind. In this case, C_k, S_k, and M_Π are loaded, manipulated, and pre-processed within the host code, before being sent to the kernel function which will perform computations on these function arguments in parallel. In order to represent C_k, S_k, and M_Π, text files are created in order to house each input, whereby each element of the vector or matrix is entered in the file in order, from left to right, with a blank space in between as a delimiter. The matrix however is entered in row-major (a linear array of all the elements, rows first, then columns) order format i.e. for the matrix M_Π seen in 1, the row-major order version is simply

$$-1, 1, 1, -2, 1, 1, 1, -1, 1, 0, 0, -1, 0, 0, -2 \qquad (3)$$

Row major ordering is a well-known ordering and representation of matrices for their linear as well as parallel manipulation in corresponding algorithms [7]. Once all computations are done for the $(k+1)th$ configuration, the result of equation 2 are then collected and moved back from the device back to the host, where they can once again be operated on by the host/CPU. It is also important to note that these operations in the host/CPU provide logic and control of the data/inputs, while the device/GPU provides the arithmetic or computational 'muscle', the laborious task of working on multiple data or instructions at a given time, hence the current dichotomy of the CUDA programming model [6]. This division of labor is implemented in exemplified in Listing 1 .

3.2 Matrix computations and CPU-GPGPU interactions

Once all 3 initial and necessary inputs are loaded, as is to be expected from equation 2, the device is first instructed to perform multiplication between the spiking vector S_k and the matrix M_Π. To further simplify computations at this point, the vectors are treated and automatically formatted by the host code to appear as single row matrices, since vectors can be considered as such. Multiplication is done per element (one element is in one thread of the device/GPGPU), and then the products are collected and summed to produce a single element of the resulting vector/single row matrix.

Once multiplication of the S_k and M_Π is done, the result is similarly added to the configuration vector C_k, once again element per element, with each element belonging to one thread, executed all at the same time as the others.

For this simulator, the host code consists almost entirely of the programming language Python, a well-known high- level, object oriented programming (OOP) language. The reason for using a high-level language such as Python is because the initial inputs, as well as succeeding ones resulting from exhaustively applying the rules and equation 2 require manipulation of the vector/matrix elements or values as strings to be concatenated, checked on (if they conform to the form (b-3) for example) by the host, as well as manipulated in ways which will be elaborated in the following sections along with the discussion of the algorithm for producing all possible and valid spiking vectors and configuration vectors given initial conditions. A language such as Python is well-suited for such a task, and can be byte-compiled like a C program for improved performance. The host code/Python part thus implements the logic and control as mentioned earlier, while in it, the device/GPU code which is written in C executes the parallel parts of the simulator.

4. DESIGN AND IMPLEMENTATION OF AN SNP SYSTEM SIMULATOR USING CUDA GPGPUS

The current SNP simulator, which is based on the type of SNP systems currently without time delays, is capable of implementing rules of the form (b-3) i.e. whenever the regular expression E is the same as the number of spikes consumed in that rule. Rules are entered in the same manner as the earlier mentioned vectors and matrix, as blank space delimited values (from one rule to the other, belonging to the same neuron) and $ delimited (from one neuron to the other). Thus for the SNP system Π shown earlier, the file r containing the blank space and $ delimited values is as follows:

$$2\ 2\ \$\ 1\ \$\ 1\ 2 \tag{4}$$

That is, rule (1) from Figure 1 has the value 2 in the file r (though rule (1) isn't of the form (b-3) it nevertheless consumes a spike since its regular expression is of the same regular expression type as the rest of the rules of Π). Another implementation consideration was the use of *lists* in Python, since unlike dictionaries or tuples, lists in Python are *mutable*, which is a direct requirement of the vector/matrix element manipulation to be performed later on (concatenation mostly). Hence a configuration vector $C_k =< 2, 1, 1 >$ is represented as $[2, 1, 1]$ in Python. That is, at the kth configuration of the system, the number of spikes of neuron 1 are given by accessing the index (starting at zero) of the configuration vector Python $list$ variable $confVec$, in this case if

$$confVec = [2, 1, 1] \tag{5}$$

then $confVec[0] = 2$ are the number of spikes available at that time for neuron 1, $confVec[1] = 1$ for neuron 2, and so on. The file r, which contains the ordered list of neurons and the rules that comprise each of them, is represented as a $listofsub - lists$ in the Python/host code. For SNP Π we have the following:

$$r = [[2, 2], [1], [1, 2]] \tag{6}$$

Neuron 1's rules are given by accessing the sub-lists of r (again, starting at index zero) i.e. rule (1) is given by $r[0][0] = 2$ and rule (4) is given by $r[2][1] = 1$.

4.1 Algorithms in the implementation of the SNP simulator

The general algorithm is as shown in Listing 2

Listing 2: Overview of the algorithm for the SNP simulator

```
Require: creation of files confVec, M, and r

I. (HOST) Load inputs: configuration vector
file (confVec ), spiking transition matrix
file (M), and rule criteria file ( r ).
Note that M and r need only be loaded
once since they are unchanging for a
given SN P system.

II. (HOST) Determine if a rule/element in r
is applicable based on the the spike value
in configVec, and then generate all valid +
possible spiking vectors in a list of lists
spikVec given r and confVec.

III. (DEVICE) From part II, run the kernel
function on spikVec, which contains all
the valid + possible spiking vectors for
the current  confVec and r. This will
generate further Ck and their corresponding Sk.

IV. (HOST+DEVICE) Repeat steps I to IV until
a zero configuration vector (vector with
only zeros as elements) or further Ck
produced are repetitions of a Ck
produced at an earlier time.
```

Step IV of Listing 2 makes the algorithm stop with 2 *stoppingcriteria* to do this: One is when there are no more available spikes in the system (hence a zero value for a configuration vector), and the second one being the fact that all previously generated configuration vectors have been produced in an earlier time or computation, hence using them again in part I of the simulator algorithm overview would be pointless, since a redundant, unnecessary infinite loop will only be formed. Another important point to notice is that either of the stopping criterion could allow for a deeply nested computation tree, one that can continues on executing for a certain length of time even with a multi-core CPU and even more parallel GPGPU.

Each line in Listing 2 mentions which parts of the simulator run in which part of the CUDA programming model, either in the device (DEVICE) or in the host (HOST) part.

4.2 Further inspection and detailing of the SNP system simulator

The more detailed algorithm for part II of Listing 2 is as follows.

A few definitions are in order at this point:

$$\Sigma = |r|, total number of neurons. \tag{7}$$

$$\Psi = |\sigma_{V1}||\sigma_{V2}|...|\sigma_{Vn}|, n \in \mathbb{N}, n \leq \infty \tag{8}$$

where

$|\sigma_{Vn}|$

means the total count of the number of rules in the nth neuron which satisfy the regular expresion E in (b-3)

During the exposition of the algorithm, the previous Python lists (from their vector/matrix counterparts in earlier sections) (5) and (6) will be utilized. For part II Listing 2 we have the following sub-algorithm for generating all valid and possible spiking vectors given input files M (the $list$ version of (3)) , $configVec$, and r:

tmp in increasing order of N, as long as the element/s satisfy the rule's regular expression E of a rule (given by list r). Elements that don't satisfy E are marked with 0.

In this case

tmp = r

then tmp = [[2, 2], [1], [1, 2]] and that given (a) we thus have the following passes as we traverse tmp, with underlined elements being the elements that are currently being checked at that stage/pass: 1st pass: tmp = [[1, 2], [1], [1, 2]] Remark/s: previously, tmp [0][0] was equal to 2, but now has been changed to 1, since it satisfies E (configVec[0] = 2 w/c is less than or equal to 2, the number of spikes consumed by that rule). 2nd pass: tmp = [[1, 2], [1], [1, 2]] Remark/s: previously tmp[0][1] = 2, which has now been changed (incidentally) to 2 as well, since it's the 2nd element of neuron 1 which satisfies E. 3rd pass: tmp = [[1, 2], [1], [1, 2]] Remark/s: 1st (and only) element of neuron 2 which satisfies E. 4th pass: tmp = [[1, 2], [1], [1, 2]] Remark/s: Same as the 1st pass 5th pass: tmp = [[1, 2], [1], [1, 0]] Remark/s: element tmp[2][1], or the 2nd element/rule of neuron 3 doesn't satisfy E. Final result: tmp = [[1, 2], [1], [1, 0]] At this point we can observe the following, based on the earlier definitions: Îč = 3 (3 neurons in total, one per element/value of confVec) ÎÍ = |ÏČV1| |ÏČV2| |ÏČV3| = 2 * 1 * 1 = 2 ÎÍ tells us the number of valid strings of 1s and 0s i.e. spiking vectors, which needs to be produced later, for a given configuration, in this case we have (a). There are only 2 valid spiking vectors from configVec = [2, 1, 1] and the rules given in (b) encoded in r. These spiking vectors are 01 1 10 10 1 10

Listing 3: Detailed algorithm for part *II* in Listing 2

```
II-1 Create a list tmp, a copy of r, marking
each element of tmp in increasing order of N,
as long as the element/s satisfy the rule's
regular expression E of a rule (given by list
r ). Elements that don't satisfy E are marked
 with 0.
In this case
tmp = r
then
tmp = [ [ 2, 2 ], [ 1 ], [ 1, 2 ] ]
and that given (a) we thus have the following passes as we
traverse tmp, with underlined elements being the elements
that are currently being checked at that stage/pass:
1st pass:
tmp = [ [ 1, 2 ], [ 1 ], [ 1, 2 ] ]
Remark/s: previously, tmp [ 0 ][ 0 ] was equal to 2,
but now has been changed to 1, since it satisfies E
( configVec[ 0 ] = 2 w/c is less than or equal to 2, the
number of spikes consumed by that rule).
2nd pass:
tmp = [ [ 1, 2 ], [ 1 ], [ 1, 2 ] ]
Remark/s: previously tmp[ 0 ][ 1 ] = 2, which has
now been changed (incidentally) to 2 as well, since
it's the 2nd element of neuron 1 which satisfies E.
3rd pass:
tmp = [ [ 1, 2 ], [ 1 ], [ 1, 2 ] ]
Remark/s: 1st (and only) element of neuron 2 which
satisfies E.
4th pass:
tmp = [ [ 1, 2 ], [ 1 ], [ 1, 2 ] ]
Remark/s: Same as the 1st pass
5th pass:
tmp = [ [ 1, 2 ], [ 1 ], [ 1, 0 ] ]
Remark/s: element tmp[ 2 ][ 1 ], or the 2nd
element/rule of neuron 3 doesn't satisfy E.
Final result:
tmp = [ [ 1, 2 ], [ 1 ], [ 1, 0 ] ]
At this point we can observe the following, based o
earlier definitions:
Îč = 3 ( 3 neurons in total, one per element/value
confVec )
ÎÍ = |ÏČV1| |ÏČV2| |ÏČV3| = 2 * 1 * 1 = 2
ÎÍ tells us the number of valid strings of 1s and 0
spiking vectors, which needs to be produced later,
given configuration, in this case we have (a). The
2 valid spiking vectors from configVec = [ 2, 1, 1
rules given in (b) encoded in r. These spiking vect
01 1 10
10 1 10
```

5. CONCLUSIONS AND FUTURE WORK

Using a highly parallel computing device such as a GPGPU, particularly NVIDIA CUDA, an SNP system simulator was successfully designed and implemented. The use of a high level programming language such as Python for host tasks, mainly for logic and string representation and manipulation of values (vector/matrix elements) provided the necessary expressivity to implement the algorithms created to produce and exhaust all possible and valid configuration and spiking vectors. For the device tasks, CUDA C allowed the manipulation of the NVIDIA CUDA enabled GPGPU which took care of repetitive and highly parallel computations (addition and multiplication essentially).

Future versions of the SNP system simulator will focus on several improvements. These improvements include the use of an algorithm for matrix computations without requiring the input matrix to be turned into a square matrix (this is currently handled by the simulator by padding zeros to an otherwise non-square matrix input). Another improvement would be the simulation of systems not of the form b-3). Byte-compiling the Python/host part of the code to improve performance as well as metrics to further enhance and measure execution time are desirable as well. Finally, deeper understanding of the CUDA architecture, such as inter- thread/block communication, for extremely large systems with equally large matrices, is required. These improvements as well as the current version of the simulator should also be run in a machine with higher versions of GPGPUs running NVIDIA CUDA.

6. ACKNOWLEDGMENTS

The authors are supported by the *ERDT Project*. They also wish to acknowledge the *Algorithms and Complexity laboratory* of *UP Diliman Department of Computer Science* for the use of Apple iMacs with NVIDIA CUDA enabled GPUs, which provided the proper environment for the simulations, their development, design and tests.

7. REFERENCES

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

[2] X. Zeng, H. Adorna, M. A. Martinez-del-Amor, L. Pan, "When Matrices Meet Brains", *Proceedings of the Eighth Brainstorming Week on Membrane Computing* , Sevilla, Spain, Feb. 2010.

[3] X. Zeng, H. Adorna, M. A. Martinez-del-Amor, L. Pan, M. Pérez-Jiménez, "Matrix Representation of Spiking Neural P Systems", *11th International Conference on Membrane Computing* , Jena, Germany, Aug. 2010.

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

[5] P systems resource website. (2010, Jan) [Online]. Available: `www.ppage.psystems.eu`.

[6] J. Cecilia, J. Garcia, G. Guerrero, M. Martinez-del-Amor, I. Perez-Jurtado, M.J. Pérez-Jiménez, "Simulating a P system based efficient solution to SAT by using GPUs", *Journal of Logic and Algebraic Programming* , Vol 79, issue 6, pp. 317-325, Apr. 2010.

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

[8] NVIDIA corporation, *"NVIDIA CUDA C programming guide"* , version 3.0, CA, USA: NVIDIA, 2010.

[9] NVIDIA CUDA developers resources page: tools, presentations, whitepapers. (2010, Jan) [Online]. Available: `http://developer.nvidia.com/page/home.html` .

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

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