PARALLEL IMPLEMENTATION OF SPIKING NEURAL P SYSTEMS SIMULATOR USING GPGPUS [DRAFT]

Francis George CABARLE[1], Henry ADORNA[1]

[1]University of the Philippines, Department of Computer Science (Algorithms and Complexity lab), Diliman, Quezon City, Philippines

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 multi-core CPUs, are used for parallelizable problems due to their architectural optimization for parallel computing. Membrane computing or P systems, are 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 algorithms, design considerations, implementation of the simulator, as well as simulation results are discussed.

Key words : Parallel computing, Membrane computing, GPGPUs

1 INTRODUCTION

The trend for massively parallel computation is moving from the more common multi-core CPUs towards GPGPUs for several significant reasons [CITE]. 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 [CITE] in order to obtain the same level of parallelization and performance. 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 [CITE]. 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.

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 George Paun in 1998 [CITE] and takes inspiration, similar to other members of natural computing, from nature. 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. [CITE]. 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 [CITE]. This representation allows design and implementation of an SN P system simulator using parallel computing machines such as GPGPUs.

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). NVDIA CUDA extends the widely known ANSI C programming language and makes parallel computation via GPGPUs manufactured by NVIDIA more accessible.

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. An SNP system of this type is of the form .

$$\Pi=(O, \sigma_1, \ldots, \sigma_m, syn, \sigma_{inp}, \sigma_{outp}) \qquad (2a)$$

(2a) represents an SNP system without delay of degree $m \geq 1$, that is, with m neurons, where

1. $O = \{ a \}$ represents the singleton set made up of a, the spike

2. $\sigma_1, \ldots, \sigma_m$ *neurons of the form*

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

where

a) $n_i \geq 0$ are the initial number of spikes in the neuron σ_i

b) R_i are finite set of rules of the form

*b-1) $E/a^c \rightarrow a^p$, **spiking rules**, where E is the regular expression over a, and $c \geq 1$, $p \geq 1$ such that $c \geq p$*

*b-2) $a^s \rightarrow \lambda$ are **forgetting rules** such that a s spikes are removed from the neuron, and that for each rule of type b-1) from R_i, $a^s \notin L(E)$*

3. *syn are the synapses or connections between neurons such that no synapse connects a neuron to itself*

4. σ_{inp},σ_{outp} are 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 p spikes to neurons adjacent to it. 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

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

where E = a^k , again consuming k spikes and producing a single spike.

The non-determinism of SNP systems come 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 [CITE]. The neurons in an SN P system operate in parallel and in unison, under a global clock. For Fig. 1 no input neuron is present, but neuron 3 is the output neuron, hence the arrow pointing towards the environment, outside the SNP system [CITE]. The SNP system in Fig. 1 is a 3 neuron system.

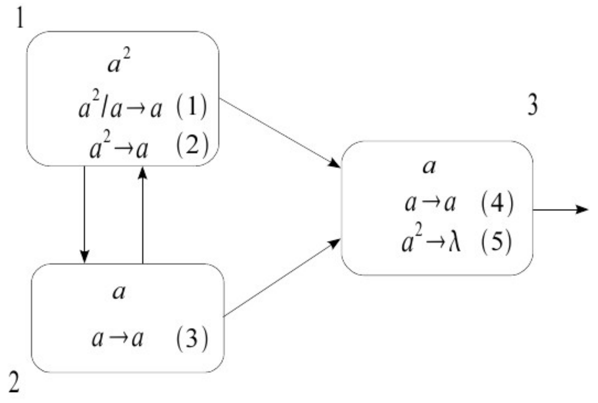

Fig. 1: An SNP system, Π, generating all natural numbers except the number 1

2.2 Matrix representation of SNP systems

A matrix representation of an SN P system consists of the following. It is important to note that, just as in Fig. 1, a total ordering of rules (in this case rule (1) to (5)) is in order.

Configuration vector of C_k, where C_0 is the initial vector containing all spikes in the system, per neuron. For Π (SNP system in Fig. 1) the initial configuration vector is *(2, 1, 1)* per neuron, respectively.

Spiking vector which shows, at a given configuration C_k, if a rule is applicable (has value 1 in the total ordering of the vector) or not (has value 0 instead). For Π we have the spiking vector at the *kth* configuration $S_k = (1, 0, 1, 1, 0)$. 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 such that a_{ij} is equal to

-*c*, if rule r_i is in σ_j and it's applied consuming c spikes

p, if rule r_i is in σ_s, s not being equal to j and that (s,j) is a synapse in Π and is applied using p spikes

0, if rule r_i is in σ_s, s not being equal to j and that (s,j) is *not* a synapse in Π

For Π, the M_Π is as follows:

$$\begin{matrix} -1 & 1 & 1 \\ -2 & 1 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & -2 \end{matrix}$$

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

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

$$C_k = C_{k-1} + S_{k-1} * M_\Pi \qquad (2b)$$

3 PARALLEL COMPUTING AND THE NVIDIA CUDA ARCHITECTURE

NVIDIA, a well known manufacturer of GPUs, released in 2006 the CUDA programming model and architecture. 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 Fig. 2. Parallelized code will run faster with more cores than with fewer ones [CITE].

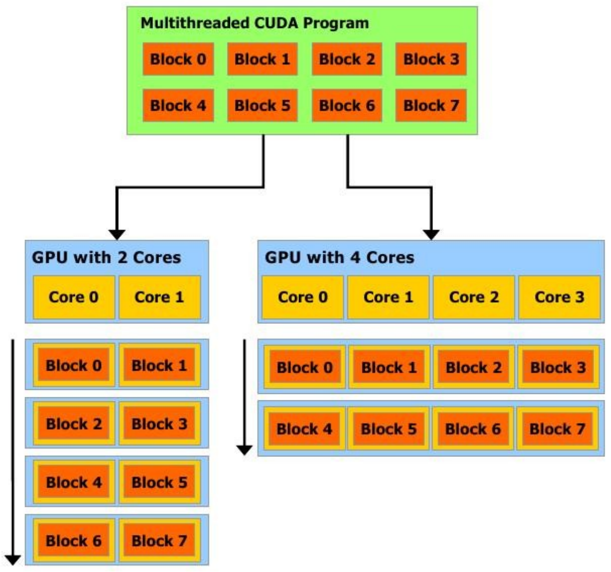

Fig. 2: NVIDIA CUDA automatic scaling, hence more cores result to faster execution

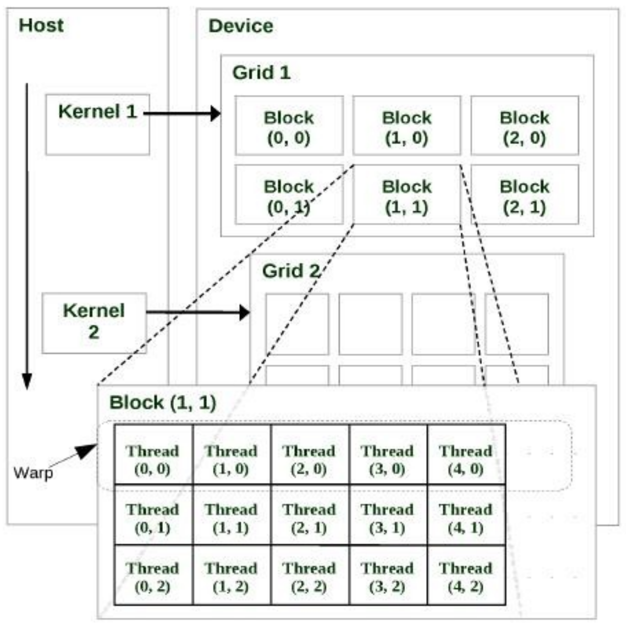

Fig. 3: NVIDIA CUDA programming model

Fig. 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.

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

Listing 1. General code flow for CUDA programming

Lines 2 and 19, in this case, implement CUDA versions of the standard C language functions e.g. the standard C function *malloc* has the CUDA C function counterpart being *cudaMalloc*.

Lines 8 and 14 show a CUDA C specific function, namely *cudaMemcpy*, which, given an input of pointers (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* this case allows the programmer to specify *N* number of threads which will execute the *add* kernel function, with *1* specifying only 1 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 (2b), 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 within the host, 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 were 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}, the row-major order version is simply

-1, 1, 1, -2, 1, 1, 1, -1, 1, 0, 0, -1, 0, 0, -2

Row major ordering is a well-known ordering and representation of matrices for their linear as well as parallel manipulation in corresponding algorithms [CITE]. Once all computations are done for the *kth* configuration, the result of equation (2b) are then collected or 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 model [CITE]. This division of labor is implemented in Listing 1.

3.2 Matrix computations and CPU-CUDA/GPU interactions

Once all 3 initial and necessary inputs are loaded, as is to be expected from equation (2b), the device is first instructed to perform multiplication between the spiking vector and the matrix. 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), and then the products are collected and summed to produce a single element of the resulting vector/single row matrix.
Once multiplication of the spiking vector and M_{Π} is done, the result is similarly added to the configuration vector, 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 (2b) 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 th e 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

That is, rule (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 with value (*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 is given by accessing the index (starting at zero) of the configuration vector list variable *confVec*, in this case if

confVec = [2, 1, 1]

then *confVec[0] = 2* is 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 them, is represented as a list of sub-lists in the Python/host code. For SNP Π we have the following:

r = [[2, 2], [1], [1, 2]]

Neuron 1's rules are given by accessing the sub-lists of *r* (again, starting at index zero) i.e. rule (1) is given *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.

```
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 C_k and their corresponding S_k .*

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

Listing 2. Overview of the algorithm for the SNP simulator

Step *IV* of the overview of the simulator algorithm makes the algorithm stop with 2 criteria 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 an redundant, unnecessary 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 the simulator algorithm overview mentions which parts of the simulator run in which part of the CUDA programming model, either in the device or in the host.

4.2 Further inspection and detailing of the SNP system simulator

The more detailed algorithm for part *II* of the SNP system simulator algorithm is as follows.
A few definitions are in order at this point:

$\Sigma = |r|$, *total number of neurons.*

$\Psi = |\sigma_{V1}|\ |\sigma_{V2}|\ ...|\sigma_{Vn}|$, $n \in \mathbb{N}$ *and* $n < \infty$, *where* $|\sigma_{Vn}|$ *means the total count of the number of rules in each neuron which satisfy the regular expression of the form b-3)*

During the exposition of the algorithm, the following Python lists (from their vector/matrix counterparts in earlier sections) will be used for demonstration purposes:

(a) configVec = [2, 1, 1]

(b) r = [[2, 2], [1], [1, 2]]

For part *II* Listing 2. (SNP simulator algorithm overview) we have the following sub-algorithm for generating all valid and possible spiking vectors given *M, configVec,* and *r* as inputs:

II-1 Create a list __tmp__, a copy of __r__, marking each element of __tmp__ in increasing order of $\mathbb{N}$, 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:

$\Sigma = 3$ *(3 neurons in total, one per element/value of __confVec__)*

$\Psi = |\sigma_{V1}|\ |\sigma_{V2}|\ |\sigma_{V3}| = 2 * 1 * 1 = 2$

Ψ 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

In order to produce these spiking vectors algorithmically, it's important to notice that first, all possible and valid spiking vector strings (made up of 1s and 0s) per neuron have to be produced first, which is facilitated by *II-1* and its output (the current value of the list *tmp*). The above 10 strings are segmented/separated to indicate they belong to the 3 different neurons.

II-2 To generate all possible and valid spiking vectors from ***tmp****, we go through each neuron i.e. all elements of* ***tmp****, since we know a priori* $\boldsymbol{\Sigma}$ *as well as the number of elements per neuron which satisfy* $\boldsymbol{E}$*. We only need to iterate through each neuron/element of tmp ω times, where ω is the both the largest and last number in the sub-list/neuron, which tells us how many elements of that neuron satisfy* $\boldsymbol{E}$ *(from II-1.). We then produce a new list,* ***tmp2****, which is made up of a sub-list of strings from all possible and valid 10 strings a.k.a. spiking vectors per neuron.*

given that now from *II-1*,

tmp = [[1, 2], [1], [1, 0]]

we iterate over neuron 1 twice, since $\omega = 2$, i.e. neuron 1 has only 2 elements which satisfy $\boldsymbol{E}$, and consequently, it is its 2nd element,

tmp[0][1] = 2.

For neuron 1, our first pass along its elements/list is as follows. Its 1st element,

tmp[0][0] = 1

is the first element to $\boldsymbol{E}$, hence it requires a 1 in its place, and 0 in the others. We therefore produce the string

10

for it. Next, the 2nd element satisfies $\boldsymbol{E}$ and it too, deserves a 1, while the rest get 0s. We produce the string

01

for it.
The new list, *tmp2*, collecting the strings produced for neuron 1 therefore becomes

tmp2 = [[10, 01]]

Following these procedures, for neuron 2 we get *tmp2* to be as follows:

tmp2 = [[10, 01], [1]]

Since neuron 2 which has only one element only has 1 possible and valid string, the string 1.
Finally, for neuron 3, we get *tmp2* to be

tmp2 = [[10, 01], [1], [10]]

In neuron 3, we iterated over it only once because ω, the number of elements it has which satisfy $\boldsymbol{E}$, is equal to 1 only. Observe that the sublist

tmp2[0] = [10, 01]

is equal to all possible and valid 10 strings for neuron 1, given rules in (b) and the number of spikes in *configVec*.

II-3. A naïve method of obtaining all possible and valid 10 strings (spiking vectors), given that there are multiple strings to be concatenated (as in *tmp2's* case), pairing up the neurons first, in order, and then exhaustively distributing every element of the first neuron to the elements of the 2nd one in the pair. That is, given the following valid and possible 10 strings for neurons 1, 2, and 3 (separated by spaces) from (a) and (b)

```
10    1    10
01
```

first, pair the strings of neurons 1 and 2, and then distribute them exhaustively to the other neuron's possible and valid strings.

```
10 →    1 => 101
01
```

and

```
10
01 →    1 => 011
```

now we have to create a new list from *tmp2*, which will house the concatenations we'll be doing. In this case,

tmp3 = [101, 011]

next, we pair up *tmp3* and the possible and valid strings of neuron 3

```
101 → 10 => 10110
011
and
101
011 → 10 => 01110
```

eventually turning *tmp3* into

tmp3 = [10110, 01110]

The final output of the sub-algorithm for the generation of all valid and possible spiking vectors is a list,

tmp3 = [10110, 01110]

Thus *tmp3* is the list of all possible and valid 10 strings given (a), and (b) in this instance. Furthermore, *tmp3* includes all possible and valid spiking vectors for a given configuration of an SN P system with all its rules and synapses (interconnections). Part *II-3* is done *Σ - 1* times, albeit exhaustively still so, between the two lists/neurons in the pair.

5 SIMULATION COMPUTATION RESULTS, OBSERVATIONS, AND ANALYSIS

Running the SNP system simulator (combination of Python and CUDA C) implements the algorithms in section 4 earlier. A sample simulation run with the SNP system *Π* is shown in Listing 3 (output truncated for space constrains)

```
****SN P system simulation run STARTS here****
Spiking transition Matrix:
...
Rules of the form a^n/a^m -> a^i or a^n ->a^i loaded:
['2', '2', '$', '1', '$', '1', '2']
Initial configuration vector:
211
Number of neurons for the SN P system is 3
Neuron 1  rules criterion/criteria and total order
...
 tmpList = [['10', '01'], ['1'], ['10']]
   All valid spiking vectors: allValidSpikVec = [['10110', '01110']]
  All Generated Ck with current Ck-1 ['211', '212', '112']
 End of C0
**
**
**
 initial total Ck list is  ['211', '212', '112']
Current confVec: 212
    All Generated Ck with current Ck-1= ['211', '212', '112', '213', '113']
**
**
**
Current confVec: 112
All Generated Ck with current Ck-1= ['211', '212', '112', '213', '113', '202', '201']
**
**
...
```

Listing 3. Sample simulator run (truncated output) of the SNP simulator for SNP system Π

The simulation for *Π* ends with the total generated configuration vector list equal to

allGenCk = ['211', '212', '112', '213', '113', '202', '201', '214', '114', '203', '111', '012', '011', '215', '115', '204', '013', '102', '101', '216', '116', '205', '014', '103', '100', '217', '117', '206', '015', '104', '218', '118', '207', '016', '105', '219', '119', '208', '017', '106', '2110', '1110', '209', '018', '107', '019', '108', '109']

That is, the computation tree for *Π* went down as deep as *confVec = 109*. At that point, all configuration vectors for all possible and valid spiking vectors have been produced. It's also noteworthy that simulation for *Π* didn't stop at the 1st stopping criteria (arriving at a zero vector) since *Π* generates all natural counting numbers greater than 1, hence a loop is to be expected. Thus the simulation was able to exhaust all possible configuration vectors and their spiking vectors, stopping only since a repetition of an earlier generated *confVec* would introduce a loop (triggering the 2nd stopping criteria).
Graphically (though not shown exhaustively) the computation tree for is as *Π* follows

```
211
├── 112
│   ├── 201
│   │   ├── 011
│   │   └── 111
│   │       └── 201
│   └── 202
│       ├── 011
│       ├── 012
│       │   ├── 101
│       │   │   └── 100
│       │   │       └── 000
│       │   └── 102
│       │       ├── 100
│       │       └── 101
│       ├── 111
│       └── 112
└── 212
    ├── 112
    ├── 113
    │   ├── 202
    │   └── 203
    │       ├── 012
    │       ├── 013
    │       │   ├── 102
    │       │   └── 103
    │       │       ├── 101
    │       │       └── 102
    │       ├── 112
    │       └── 113
    ├── 212
    └── 213
        ├── 113
        ├── 114
        │   ├── 203
        │   └── 204 (...)
        ├── 213
        └── 214
            ├── 114
├── 115 (...)
├── 214
└── 215 (...)
```

Listing 4. Computation tree for SNP system Π

The *confVecs* followed by (…) are the *confVec*s that went deeper than the Listing 4 has shown.

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.

ACKNOWLEDGMENT

The author wishes to acknowledge the Algorithms and Complexity laboratory of the Department of Computer Science. The use of an Apple iMac with its NVIDIA CUDA enabled GPUs provided the proper training grounds for the simulations, their development, design, and tests.

REFERENCES

[1] Ionescu, M., Paun, G., and Yokomori, T. (February 2006). Spiking Neural P Systems. Journal Fundamenta Informaticae, Vol. 71, issue 2,3 pp. 279-308

[2] Zeng, X., Adorna, H., Martinez-del-Amor, M., Pan, L., and Perez-Jimenez, M.J. (August 2010). Matrix Representation of Spiking Neural P Systems. 11th International Conference on Membrane Computing, Jena, Germany

[3] Cecilia, J., Garcia, J., Guerrero, G., Martinez-del-Amor, M., Perez-Jurtado, I., and Perez-Jimenez, M.J. (April 2010). 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

[4] Kirk, D., Hwu, W.. (February 2010). Programming Massively Parallel Processors: A Hands On Approach. Morgan Kaufmann 1st ed. Burlington, MA, USA

[5] NVIDIA corporation. (February 2010). Programming Massively NVIDIA CUDA C programming guide version 3.0. Santa Clara, CA, USA