Sparse-matrix Representation of Spiking Neural P Systems for GPUs

Miguel Á. Martínez-del-Amor[1], David Orellana-Martín[1], Francis G.C. Cabarle[2], Mario J. Pérez-Jiménez[1], Henry N. Adorna[2]

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

Summary. Current parallel simulation algorithms for Spiking Neural P (SNP) systems are based on a matrix representation. This helps to harness the inherent parallelism in algebraic operations, such as vector-matrix multiplication. Although it has been convenient for the first parallel simulators running on Graphics Processing Units (GPUs), such as CuSNP, there are some bottlenecks to cope with. For example, matrix representation of SNP systems with a low-connectivity-degree graph lead to sparse matrices, i.e. containing more zeros than *actual* values. Having to deal with sparse matrices downgrades the performance of the simulators because of wasting memory and time.

However, sparse matrices is a known problem on parallel computing with GPUs, and several solutions and algorithms are available in the literature. In this paper, we briefly analyse some of these ideas and apply them to represent some variants of SNP systems. We also conclude which variant better suit a sparse-matrix representation.

Keywords: Spiking Neural P systems, Simulation Algorithm, Sparse Matrix Representation, GPU computing, CUDA

1 Introduction

Spiking Neural P (SNP) systems [9] are a type of P systems [16] composed of a directed graph inspired by how neurons are interconnected by axons and synapses

in the brain. Neurons communicate through spikes, and the time difference between them plays an important role in the computation.

The simulation of SNP systems have been carried out through sequential simulators such as pLinguaCore [11]. For parallel simulation, a matrix representation was introduced [17], so that the simulation algorithm is based on applying matrix operations. For instance, efficient algebra libraries have been defined for GPUs, given that they fit well to the highly parallel architecture of these devices. This have been harnessed already to introduce the first parallel SNP system simulators on GPUs, cuSNP [4, 5].

However, this matrix representation can be sparse, having a majority of zero values, because the directed graph of SNP systems are not normally fully connected. In many disciplines, sparse vector-matrix operations are natural, so many solutions have been proposed in the literature [6]. For this reason, we transfer some of these ideas to the simulation of SNP systems with matrix operations. First, we give a first approach, which is further optimized by splitting the main matrix into several structures. Second, ideas to deal with dynamic networks are given.

The paper is structured as follows: Section 2 gives a short formal definition of SNP systems; Section 3 summarizes the matrix-based simulation algorithm; Section 4 briefly introduces GPU computing and sparse vector-matrix representations; Section 5 discussed the ideas on introducing sparse vector-matrix representation for SNP system simulation; and the paper finishes with conclusions and future work.

2 Spiking Neural P Systems

Definition 1. *A* spiking neural P system *of degree* $q \geq 1$ *is a tuple*

$$\Pi = (O, syn, \sigma_1, \ldots, \sigma_q, i_{out})$$

where:

- $O = \{a\}$ *is the singleton alphabet (a is called spike);*
- $syn = (V, E)$ *is a directed graph such that* $V = \{\sigma_1, \ldots, \sigma_q\}$ *and* $(\sigma_i, \sigma_i) \notin E$ *for* $1 \leq i \leq q$*;*
- $\sigma_1, \ldots, \sigma_m$ *are neurons of the form*

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

  where:
  - $n_i \geq 0$ *is the initial number of spikes within neuron labelled by* i*; and*
  - R_i *is a finite set of rules associated to neuron labelled by* i*, of the following forms:*
    (1) $E/a^c \rightarrow a^p$*, being* E *a regular expression over* $\{a\}$*,* $c \geq p \geq 1$ *(firing rules);*

(2) $a^s \to \lambda$ *for some* $s \geq 1$, *with the restriction that for each rule* $E/a^c \to a^p$ *of type of type* (1) *from* R_i, *we have* $a^s \notin L(E)$ *(forgetting rules).*

- $i_{out} \in \{1, 2, \ldots, q\}$ *such that* $outdegree(i_{out}) = 0$

A spiking neural P system of degree $q \geq 1$ can be viewed as a set of q neurons $\{\sigma_1, \ldots, \sigma_q\}$ interconnected by the arcs of a directed graph *syn*, called *synapse graph* There is a distinguished neuron i_{out}, called output neuron, which communicates with the environment.

If a neuron σ_i contains k spikes at an instant t, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \to a^p$ can be applied. By the application of that rule, c spikes are removed from neuron σ_i and the neuron fires producing p spikes immediately. The spikes produced by a neuron σ_i are received for all neuron σ_j such that $(\sigma_i, \sigma_j) \in E$. If σ_i is the output neuron then the spikes are sent to the environment.

The rules of type (2) are *forgetting* rules, and they are applied as follows: If neuron σ_i contains exactly s spikes, then the rule $a^s \to \lambda$ from R_i can be applied. By the application of this rule all s spikes are removed from σ_i.

In spiking neural P systems, a global clock is assumed, marking the time for the whole system. Only one rule can be executed in each cell at step t. As models of computation, spiking neural P systems are *Turing complete*, i.e. as powerful as Turing machines. On one hand, common way to introduce input to the system is to encode into some or all of the n_i's the input(s) of the problem to be solved. On the other hand, a common way to obtain the output is by observing i_{out}: either by getting the interval $t_2 - t_1 = n$, where i_{out} sent its first two spikes at times t_1 and t_2 (we say n is computed or generated by the system), or by counting all the spikes sent by i_{out} to the environment until the system halts.

Aside from computing numbers, spiking neural P systems can also compute strings, and hence, languages. More general ways to provide the input or receive the output include the use of *spike trains*, i.e. a stream or sequence of spikes entering or leaving the system. Further results and details on computability, complexity, and applications of spiking neural P systems are detailed in [15], a dedicated chapter in the Handbook in [8], and an extensive bibliography until February 2016 in [14]. There are some interesting ingredients we are going to explain here. A broader explanation of them and more variants is provided at [1, 3, 13].

2.1 Spiking Neural P Systems with Budding

Based on the idea of neuronal budding, where a cell is divided in two new cells, we can abstract it to *budding rules*. In this process, the new cells can differ in some aspects: their connections, contents and shape. A budding rule has the following form:

$$[E]_i \to [\ ]_i / [\ ]_j,$$

where E is a regular expression and $i, j \in \{1, \ldots, q\}$.

If a neuron σ_i contains s spikes, $a^s \in L(E)$, and there is no neuron σ_j such that there exists a synapse (i, j) in the system, then the rule $[E]_i \to [\ ]_i / [\ ]_j$ is enabled

and it can be executed. A new neuron σ_j is created, and both neurons σ_i and σ_j are empty after the execution of the rule. This neuron σ_i keeps all the synapses that were going in, and this σ_j inherits all the synapses that were going out of σ_i in the previous configuration. There is also a synapse (i, j) between neurons σ_i and σ_j, and the rest of synapses of σ_j are given to the neuron depending on the synapses of *syn*.

2.2 Spiking Neural P Systems with Division

Inspired by the process of *mitosis*, division rules have been widely used within the field of *Membrane Computing*. In SN P systems, a division rule can be defined as follows:

$$[E]_i \rightarrow [\ ]_j | | [\ ]_k,$$

where E is a regular expression and $i, j, k \in \{1, \ldots, q\}$.

If a neuron σ_i contains s spikes, $a^s \in L(E)$, and there is no neuron σ_g such that the synapse (g, i) or (i, g) exists in the system, $g \in \{j, k\}$, then the rule $[E]_i \rightarrow [\ ]_j | | [\ ]_k$ is enabled and it can be executed. Neuron σ_i is then divided into two new cells, σ_j and σ_k. The new cells are empty at the time of their creation. The new neurons keep the synapses previously associated to the original neuron σ_i, that is, if there was a synapse from σ_g to σ_i, then a new synapse from σ_g to σ_j and a new one to σ_k are created, and if there was a synapse from σ_i to σ_g, then a new synapse from σ_j to σ_g and a new one from σ_k to σ_g are created. The rest of synapses of σ_j and σ_k are given by the ones defined in *syn*.

2.3 Spiking Neural P Systems with Plasticity

It is known that new synapses can be created in the brain thanks to the process of *synaptogenesis*. We can recreate this process in the framework of spiking neural P systems defining *plasticity rules* in the following form:

$$E/a^c \rightarrow \alpha k(i, N_j),$$

where E is a regular expression, $c \geq 1$, $\alpha \in \{+, -, \pm, \mp\}$, $k \geq 1$ and $N_j \subseteq \{1, \ldots, q\}$. For a neuron σ_i, let us define the *presynapses* of this neuron as $pres(i) = \{j | (i, j) \in syn\}$.

If a neuron σ_i contains s spikes, $a^s \in L(E)$, then the rule $E/a^c \rightarrow \alpha k(i, N_j)$ is enabled and can be executed. The rule consumes c spikes and, depending on the value of α, it performs one of the following:

- If $\alpha = +$ and $N_j - pres(i) = \emptyset$, or if $\alpha = -$ and $pres(i) = \emptyset$, then there is nothing more to do.
- If $\alpha = +$ and $|N_j - pres(i)| \leq k$, deterministically create a synapse to every σ_g, $g \in N_j - pres(i)$. Otherwise, if $|N_j - pres(i)| > k$, then non-deterministically select k neurons in $N_j - pres(i)$ and create one synapse to each selected neuron.

- If $\alpha = -$ and $|pres(i)| \leq k$, deterministically delete all synapses in $pres(i)$. Otherwise, if $|pres(i)| > k$, then non-deterministically select k neurons in $pres(i)$ and delete each synapse to the selected neurons.
- If $\alpha = \{\pm, \mp\}$, create (respectively, delete) synapses at time t and then delete (resp., create) synapses at time $t + 1$. Even when this rule is applied, neurons are still open, that is, they can continue receiving spikes.

Let us notice that if, for some σ_i, we apply a *plasticity rule* with $\alpha \in \{+, \pm, \mp\}$, when a synapse is created, a spike is sent from σ_i to the neuron that has been connected. That is, when σ_i attaches to σ_j through this method, we have immediately transferring one spike to σ_j.

3 Simulation of SNP Systems

So far, P system parallel simulators make use of *ad-hoc* representations, specifically defined for a certain variant [12]. In order to ease the simulation of SNP system and its deployment to parallel environments, a matrix representation was introduced [17]. By using a set of algebraic operations, it is possible to reproduce the transitions of a computation. Although the baseline representation only involves SNP systems without delays and static structure, many extensions have followed such as for enabling delays or supporting non-determinism [4, 5].

This representation includes the following vectors and matrices, for a SNP system π of degree (n, m) (n rules and m neurons):

Configuration vector: C_k is the vector containing all spikes in every neuron on the k^{th} computation step/time, where C_0 denotes the initial configuration. It contains m elements.

Spiking vector: S_k shows if a rule is going to fire at the transition step k (having value *1*) or not (having value *0*). Given the non-determinism nature of SNP systems, it would be possible to have a set of valid spiking vectors. However, for the computation of the next configuration vector, only a spiking vector is used. It contains n elements.

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

Definition 2.

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

Thus, rows represent rules and columns represent neurons in the spiking transition matrix. Note also that a negative entry 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 [17]. It contains $n \cdot m$ elements.

Hence, to compute the transition k, it is enough to select a spiking vector S_k and calculate: $C_k = S_k \cdot M_\pi + C_{k-1}$.

4 GPU Computing and SpMV Operations

The Graphics Processing Unit (GPU) has been employed for P system simulations since the introduction of CUDA . This technology allows to run scientific computations in parallel on the GPU, given that a device typically contains thousands of cores and high memory bandwidth [10]. However, parallel computing on a GPU has more constraints than on a CPU: threads have to run in a SIMD fashion, accessing data in a coalesced way; that is, best performance is achieved when the execution of threads is synchronized and accessing contiguous data from memory.

Some algorithms fit perfectly to the GPU parallel model, such as algebraic operations. Indeed, matrix computation is a "hello world" when getting started with CUDA [18], and there are many efficient libraries for algebra computations like cuBLAS. It is usual that when working with large matrices, these are almost "empty", or with a majority of zero values. This is known as *sparse matrix*, and this downgrades the runtime in two ways: lot of memory is wasted, and lot of operations are redundant.

Given the importance of linear algebra in many computational disciplines, sparse vector-matrix operations (*SpMV*, in short) have been subject of study in parallel computing (and so, on GPUs). Today there exists many approaches to tackle this problem [2]. In this paper, we will focus on two formats to represent sparse matrices, assuming that threads will access rows in parallel:

- *CSR* format. Only non-null values are represented by using 3 arrays: row pointers, non-zero values and columns (see Figure 1 for an example). First, the row-pointers array is accessed, which contains a position per row of the original matrix. Each position says the index where the row start in the non-zero values and columns arrays. The non-zero values and the columns arrays can be seen as a single array of pairs, since every entry has to be accessed at the same time. Once a row is indexed, then a loop over the values in that row has to be performed, so that the corresponding column is found, and therefore, the value. If the column is not present, then the value is assumed to be zero, since this data structures contains all non-zero values. The main advantage is that it is a full-compressed format if $NumNonZeroValues \cdot 2 > NumZeroValues$, where $NumNonZeroValues$ and $NumZeroValues$ are the number of non-zero and zero values in the original matrix, respectively. However, the drawbacks is that the search of elements in the non-zero values and columns arrays is not coalesced when using parallelism per row. Moreover, since it is a full-compressed format, there is no room for modifying the values, such as introducing new non-zero values.

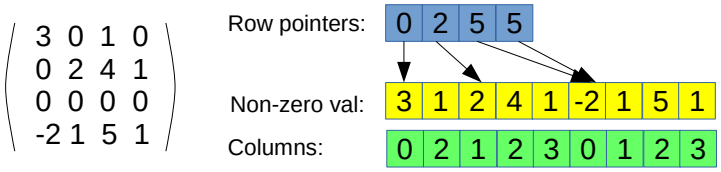

Fig. 1: CSR format example

- *ELL* format. This representation aims at increase the memory coalescing access of threads in CUDA. This is achieved by using a matrix of pairs, containing a trimmed, transposed version of the original matrix (see Figure 2 for an example). Each column of the ELL matrix is devoted for each row of the matrix, even though the row is empty (all elements are zero). Every element is a pair, where the first position denotes the column and the second is the value, of only the non-zero elements in the corresponding row. However, the size of the matrix is fixed, so the number of columns equals the number of rows of the original matrix, but the number of rows is the maximum length of a row in terms of non-zero values; in other words, the maximum amount of non-zero elements in a row of the original matrix. Rows containing fewer elements will pad the difference with null elements. The main advantage of this format is that threads will always access the elements of all rows in coalesced way, and the null elements padded by short rows can be utilize to incorporate new data. However, there is a waste of memory, which is worst when the rows are unbalance in terms of number of zeros.

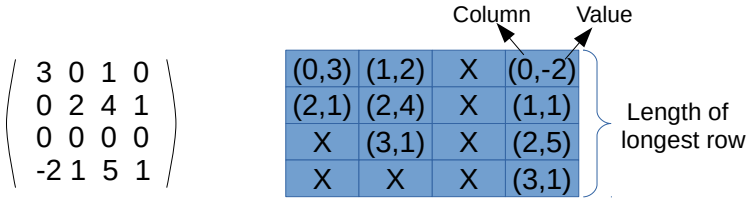

Fig. 2: ELL format example

5 Sparse Matrix Representation of SNP Systems

SNP systems in the literature typically are not fully connected graphs. In such situations, the transition matrix gets sparse, and therefore, further optimizations based on SpMV can be conveyed. In the following subsections, we discuss some

approaches. Of course, if the graph inherent to a SNP system leads to a dense transition matrix, then a normal format can be employed, because using sparse formats will increase the memory footprint.

5.1 Approach with ELL Format

The first approach to compress the representation of a sparse transition matrix, M_π, is to use the ELL format (see Figure 3 for an example), leading to the compressed matrix M_π^s. The following aspects have to be taken into consideration:

- The number of rows of M_π^s equals the maximum amount of non-zero values in a row of M_π, denoted by Z. It can be shown that $Z = MaxOutDegree + 1$, where $MaxOutDegree$ is the maximum output degree found in the neurons of the SNP system. Z can be derived from the composition of the transition matrix, where a row devoted for a rule $E/a^c \to a^p$ contains the values $+p$ for every neuron (columns) to which the neuron it belongs has a synapse, and a value $-c$ for consuming the spikes in the neuron it belongs.
- The values inside columns can be sorted, so that the consumption of spikes ($-c$ values) are placed at the first row. In this way, all threads can start with the same task, consuming spikes.
- Every position of M_π^s is a pair (although not represented in Figure 3), where the first element is a neuron label, and second is $+p$.

The idea is to assign a thread to each rule, and so, one per column of the spiking vector S_k and one per column of M_π^s (real rows of the transition matrix). For the vector-matrix multiplication, it is enough to iterate Z times (number of rows in M_π^s). In each iteration, the computed value is added to the corresponding neuron position in the configuration vector C_k. Since some threads will possibly write to the same positions in the configuration vector, a solution would be to use atomic operations, which are available on GPUs to calculate additions, among others.

5.2 Optimized Approach

If, in general, there are more than one rule in the neurons, lot of threads in the first approach will be inactive (having a 0 in the spiking vector), causing branch divergence and non-coalesced memory access. Moreover, note in Figure 3 that columns corresponding to rules belonging to the same neuron will contain redundant information: the generation of spikes is replicated for all synapses.

Therefore, a more efficient sparse matrix representation can be obtained when maintaining the synapses separated from the rule information. This can be done as follows:

- *Rule information.* By using a CSR-like format (see Figure 4 for an example), rules of the form $E/a^c \to a^p$ (also forgetting rules are included, assuming

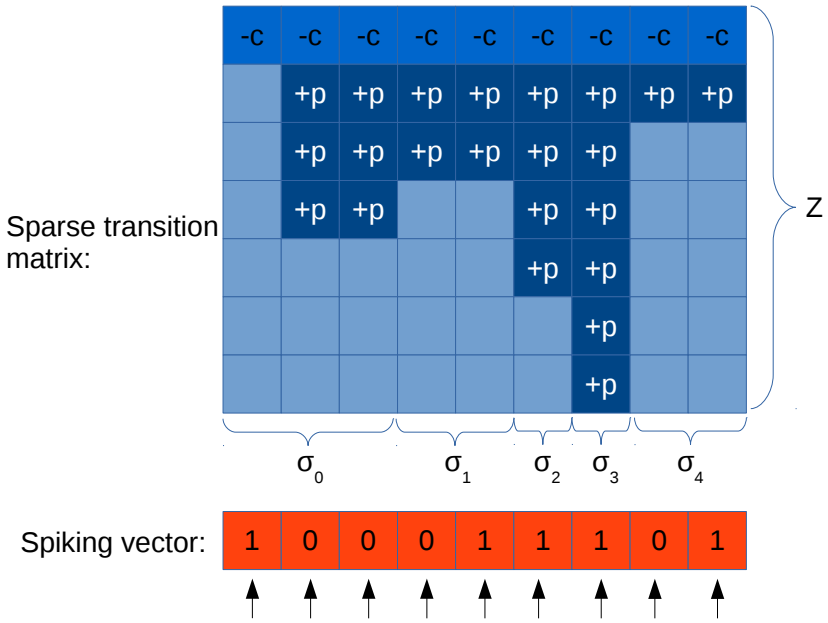

Fig. 3: Sparse matrix representation of a SNP system based on ELL format

$p = 0$) can be represented by a double array storing the values c and p (also the regular expression, but this is required only to select a spiking vector, and hence is out of scope of this work). A pointer array is employed to relate, for each neuron, the subset of rules that has associated.

- *Synapse matrix*, Sy_π. It has a column per neuron i, and a row for every neuron j such that $(i, j) \in Syn$ (there is a synapse). That is, every element of the matrix corresponds to a synapse or null, given that the number of rows equals to the maximum output degree in the neurons of the SNP system π, and padding is required.
- *Spiking vector* is modified, containing only m positions, one per neuron, and stating which rule $0 \leq r \leq n$ is selected.

The way to operate with this approach is to assign a thread to each column of the synapse matrix (requiring m threads, one per neuron). Each thread will access to the corresponding rule stated in the spiking vector, delete c in the configuration vector C_k, and add p to each neuron defined in synapse matrix in C_k.

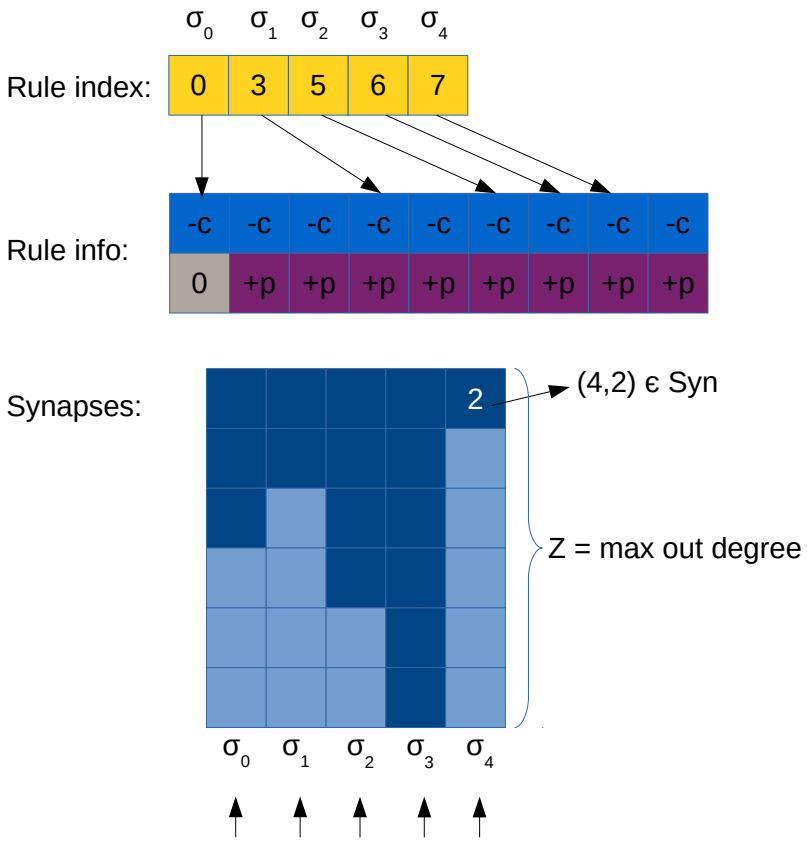

Fig. 4: Optimized sparse matrix representation

5.3 Ideas for Dynamic Networks

The optimized sparse matrix representation discussed in Section 5.2 can be further extended to support rules that modify the network, such as budding, division or plasticity.

Figure 5 shows an example on how a budding rule can be supported. First, the synapse matrix has to be flexible enough to host new neurons. This can be accomplished by allocating a matrix large enough to populate new neurons (probably up to fill the whole memory available on the GPU). Thus, for a budding rule $[E]_i \rightarrow []_j/[]_k$, the required operations to modify the synapse matrix are:

1. Allocate a column for the new neuron k.
2. Copy column i to k.

3. Delete content of column i and add only one element for k.
4. Change label i to j.

This will require to use a map of the column labels of the synapse matrix, saying to which neuron it corresponds. The same map can be used to index the rule information structure. A further optimization is to swap the labels i for k instead of copying the column content. One major drawback of this approach is that the creation of new neurons cannot run fully in parallel; that is, assigning new columns to created neurons in a transition step is a serialized process. Some techniques such as prefix sum can be applied to cope with this issue and convert a serial process into a logarithmic-step operation.

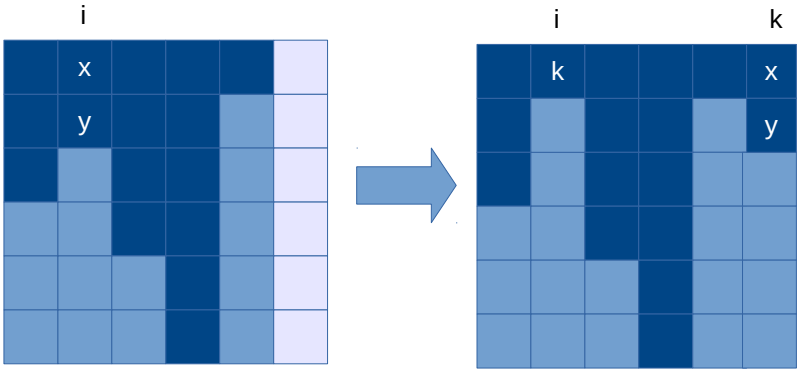

Fig. 5: Extension for budding

An example for division rules can be seen in Figure 6. Again, the synapse matrix has to be extended to contain empty columns to host populated new neurons during the simulation, and a map of neuron labels have to be managed.

For a division rule $[E]_i \rightarrow []_j || []_k$, the following operations have to be performed:

1. Allocate a new column for neuron k.
2. Copy column i to k.
3. Change label of i to j.
4. Find all occurrences of i in the synapse matrix, change it for j and add k in the column.

The last operation can be shown to be very expensive, since it requires to loop all over the synapse matrix. Moreover, when adding k in all the neurons containing i in the synapses, it would be possible to exceed the predetermined size Z. For this situation, a special array of overflows will be needed, like ELL+COO format for SpMV [2].

Finally, Figure 7 shows an example for plasticity rules. In this case, the synapse matrix can be allocated in advance to an exact size, since no new neurons

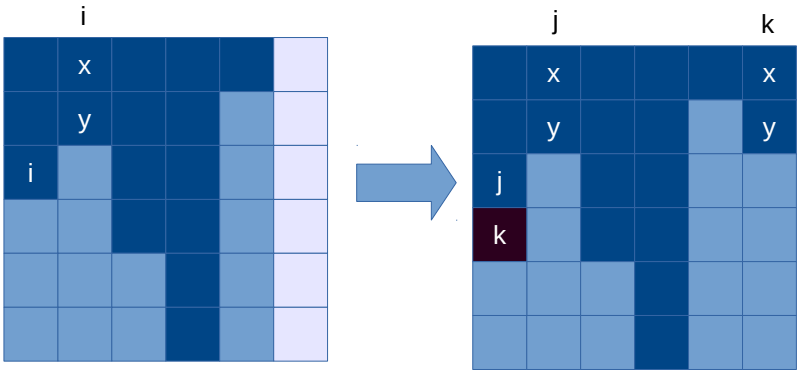

Fig. 6: Extension for division

are created. However, enough rows Z have to be pre-established to support the maximum amount of synapses, what can be precomputed by looking to the initial out degrees of the neurons and the size of the N sets in the plasticity rules for adding synapses: $E/a^c \rightarrow \alpha k(i, N_j)$, with $\alpha = +/\pm/\mp$.

The following operations have to be performed to reproduce the behaviour:

1. When deleting synapses, loop Z times to find the corresponding neurons in the matrix, and set them to null. Holes might appear in the columns.
2. When adding synapses, loop Z times to find holes in the column and add the corresponding neurons.

Since holes might appear in the columns when deleting synapses, we will need to loop over Z times every column to compute the next transition, or to add new synapses. Sorting algorithms can be run in parallel, but most probably it will not worth the effort.

6 Conclusions and Future Work

In this paper, we have analysed the problem of having sparse matrices in the matrix representation of SNP systems. Downgrades in the simulator performance would appear if no solutions are found. However, this is a known issue in other disciplines, and efficient sparse matrix representations have been introduced in the literature.

We proposed a two efficient sparse representations for SNP systems, one based on the classic format ELL, and an optimized one based on CSR and ELL. We also analysed their behaviour when supporting rules for dynamic networks: division, budding and plasticity. The representation for plasticity poses more advantages than the one for division and budding, since the synapse matrix size can be pre-computed. Thus, no label mapping nor empty columns for new neurons are

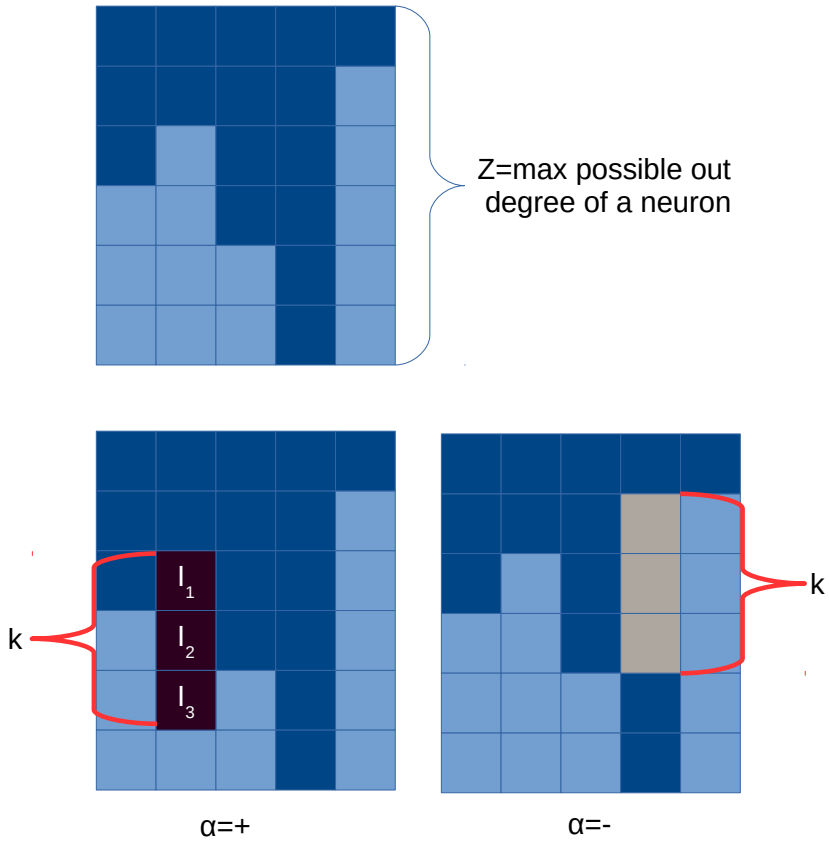

Fig. 7: Extension for plasticity

required. Moreover, simulating the creation of new neurons in parallel can damage the performance of the simulator significantly, because this operation can be sequential. Plasticity rules do not create new neurons, so this is avoided.

As future work, we plan to provide implementations of this ideas within cuSNP framework, and deep analyse the different results with real examples from the literature. We believe that this ideas will help to bring efficient tools to simulate SNP systems on GPUs, enabling the simulation of large networks in parallel.

References

1. H. Adorna, F. Cabarle, L.F. Macías-Ramos, L. Pan, M.J. Pérez-Jiménez, B. Song, T. Song, L. Valencia-Cabrera. Taking the pulse of SN P systems: A quick survey,

in: M. Gheorghe, I. Petre, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa (eds.), *Multidisciplinary creativity*, Spandugino, 2015, pp. 3–16.
2. N. Bell, M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, 2008.
3. F. Cabarle, H. Adorna, M.J. Pérez-Jiménez, T. Song. Spiking neural P systems with structural plasticity, *Neural Computing and Applications*, **26**, 8 (2015), pp. 1905–1917
4. J.P. Carandang, J.M.B. Villaflores, F.G.C. Cabarle, H.N. Adorna, M.A. Martínez-del-Amor. CuSNP: Spiking Neural P Systems Simulators in CUDA. Romanian Journal of Information Science and Technology, **20**, 1 (2017), 57–70.
5. J.P. Carandang, F.G.C. Cabarle, H.N. Adorna, N.H.S. Hernandez, M.A. Martínez-del-Amor. Nondeterminism in Spiking Neural P Systems: Algorithms and Simulations. *Asian Conference on Membrane Computing* 2017. Submitted.
6. 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, 2004, pp. 133-137.
7. M. Harris. Mapping computational concepts to GPUs, *ACM SIGGRAPH 2005 Courses*, NY, USA, 2005.
8. O. Ibarra, A. Leporati, A. Păun, S. Woodworth. Spiking Neural P Systems, in: Gh. Păun and G. Rozenberg and A. Salomaa (eds.), *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010, pp. 337–362.
9. M. Ionescu, Gh. Păun, T. Yokomori. Spiking Neural P Systems, *Journal Fundamenta Informaticae*, **71**, 2-3 (2006), 279-308.
10. D. Kirk, W. Hwu, *Programming Massively Parallel Processors: A Hands On Approach*, 1st ed. MA, USA: Morgan Kaufmann, 2010.
11. L.F. Macías, I. Pérez-Hurtado, M. García-Quismondo, L. Valencia-Cabrera, M.J. Pérez-Jiménez, A. Riscos-Núñez. A P-Lingua based simulator for Spiking Neural P systems, *Lecture Notes in Computer Science*, **7184** (2012), 257–281.
12. M.A. Martínez-del-Amor, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez. Simulating P systems on GPU devices: a survey. Fundamenta Informaticae, **136**, 3 (2015), 269-284.
13. L. Pan, Gh. Păun, M.J. Pérez-Jiménez. Spiking neural P systems with neuron division and budding, *Science China Information Sciences*, **54**, 8 (2011), 1596–1607.
14. L. Pan, T. Wu, Z. Zhang. A Bibliography of Spiking Neural P Systems. *Bulletin of the International Membrane Computing Society*, June 2016, 63–78.
15. Gh. Păun, M.J. Pérez-Jiménez. Algorithmic Bioprocesses, *Spiking neural P systems. Recent results, research topics*, Springer, 2009, pp. 273–291.
16. G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and TUCS Report No 208.
17. X. Zeng, H. Adorna, M. A. Martínez-del-Amor, L. Pan, M. J. Pérez-Jiménez. Matrix representation of spiking neural p systems. *Lecture Notes in Computer Science*, **6501** (2011), 377–391.
18. NVIDIA corporation,*NVIDIA CUDA C programming guide*, version 3.0, CA, USA: NVIDIA, 2010.