Optimizations in CuSNP Simulator for Spiking Neural P Systems on CUDA GPUs

Blaine Corwyn D. Aboy
dept. of Computer Science
University of the Philippines Diliman
Quezon city, Philippines

Edward James A. Bariring
dept. of Computer Science
University of the Philippines Diliman
Quezon city, Philippines

Jym Paul Carandang
dept. of Computer Science
University of the Philippines Diliman
Quezon city, Philippines

Francis George C. Cabarle
dept. of Computer Science
University of the Philippines Diliman
Quezon city, Philippines
fccabarle@up.edu.ph

Ren Tristan De La Cruz
dept. of Computer Science
University of the Philippines Diliman
Quezon city, Philippines

Henry N. Adorna
dept. of Computer Science
University of the Philippines Diliman
Quezon city, Philippines

Miguel Ángel Martínez-del-Amor
dept. of Computer Science and A.I.
Universidad de Sevilla
Seville, Spain

***Abstract*—Spiking Neural P systems (in short, SNP systems) are computing models based on living neurons. SNP systems are non-deterministic and parallel, hence making use of a parallel processor such as a graphics processing unit (in short, GPU) is a natural candidate for simulations. Matrix representations and algorithms were previously developed for simulating SNP systems. In this work, our two results extend previous works in simulating SNP systems in the GPU: (a) the number of neurons the simulator can handle is now arbitrary; (b) SNP systems are now represented in a dense instead of sparse way. The impact in terms of time and space of these extensions to the GPU simulator are analysed. As expected, SNP systems with more neurons need more simulation time, although the simulator performance can scale (i.e. perform better) with larger GPUs. The dense representation helps in the simulation of larger systems.**

***Index Terms*—Membrane computing, Spiking neural P systems, GPU computing, CUDA, Sparse Matrix-Vector**

I. Introduction

A branch in natural computing called membrane computing bases its computational model from that of biological cells [9]. P systems are parallel, nondeterministic, and computationally universal models in membrane computing. These systems currently cannot entirely be implemented on modern technology although some of their components or restricted versions have been simulated. The focus of this work is on spiking neural P systems (SNP systems for short) [5]. In SNP systems the neurons are vertices in a digraph, and edges between neurons are called synapses. Neurons process spikes and send these to other neurons using synapses.

There are already efforts in creating simulators for P systems such as [1]–[4] for SNP systems, including a survey in [7]. In this work we extend such simulators, specifically the current implementation of the CUDA SNP (CuSNP) simulator from [4]. We use the Compute Unified Device Architecture (CUDA for short) which is a general purpose parallel computing platform and programming model using the GPUs by NVIDIA. GPUs are massively parallel processors which can launch hundreds to thousands of threads.

A. Spiking Neural P Systems

The reader is assumed to be familiar with the basics of membrane computing and formal language theory found in many monographs such as [9]. An SNP system Π is a construct $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, out)$, where:

1. $O = \{a\}$ is the alphabet containing a single symbol (the spike).
2. $\sigma_1, \ldots, \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$ where:
   (a) $n_i \geq 0$ is the initial number of spikes contained in σ_i
   (b) R_i is a finite set of rules of the following two forms:
      (i) (Spiking Rule) $E/a^c \rightarrow a^p; d$ where E is a regular expression over O and $c \geq p \geq 1, d \geq 0$.
      (ii) (Forgetting Rule) $a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each rule of type (i) in R_i we have $a^s \notin L(E)$;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $i \neq j$ for all $(i, j) \in syn, 1 \leq i, j \leq m$ (synapses between neurons);

The authors are grateful for the computing facility by the CoARE project of DOST-ASTI, Philippines. F.G.C. Cabarle thanks Project No. 191904 ORG of OVCRD in UP Diliman, Philippines. M.A. Martínez-del-Amor also acknowledges the support by the research project MABICAP (TIN2017-89842-P), co-financed by *Ministerio de Economía, Industria y Competitividad (MINECO)* of Spain, through the *Agencia Estatal de Investigación (AEI)*, and by *FEDER* of the European Union.

4. $in, out \in \{1, 2, \ldots, m\}$ indicate the input and output neurons, respectively.

A spiking rule is applied as follows: if a neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \rightarrow a^p; d \in R_i$ can be applied. This means we remove c spikes so that $k - c$ spikes remain in σ_i, the neuron then fires and produces p spikes after d time steps. Spikes are fired after step $t + d$ where t is the current time step of the computation. If $d = 0$, the spikes are fired immediately. Between step t and $t+d$, we say σ_i has not fired the spike yet and is *closed*, i.e. σ_i cannot receive spikes from other neurons connected to it. If neurons with a synapse to σ_i fire, the spikes are lost. At step $t + d$, the spikes are fired, and σ_i is now *open* to receive spikes. At $t + d + 1$, σ_i can apply a rule.

A forgetting rule is applied as follows: If σ_i contains exactly s spikes, then the rule $a^s \rightarrow \lambda$ from R_i can be applied, meaning all of the s spikes are removed from σ_i. Rule of type (i) where $E = a^c$ can be written in the shortened form of $a^c \rightarrow a^p; d$. In the case two or more rules of σ_i are applicable at the same step, only one rule is applied and is non-deterministically chosen. A *configuration* of the system at step t is denoted as $C_t = \langle r_1/k_1, \ldots, r_m/k_m \rangle$, where for $1 \leq i \leq m$ and σ_i contains r_i spikes and remains closed for k_i more steps.

A computation is any (finite or infinite) sequence of configurations such that: (a) the first term is the initial configuration C_0; (b) for each $n \geq 1$, the n^{th} configuration of the sequence is obtained from the previous configuration in one transition step; and (c) if the sequence is finite (called *halting computation*) then the last term is a *halting configuration*, i.e. a configuration where all neurons are open and no rule can be applied. Two common ways to obtain the output of an SNP system are as follows: obtaining the time interval between exactly the first two steps when the output neuron σ_{out} spikes or between 0 and until the system halts. In this work, we consider systems that produce their output as given in [6]. To demonstrate our CuSNP simulator later in this work, we make use of SNP systems that solve the NP-complete Subset Sum problem. The Subset Sum problem asks if, given a finite set $V = \{v_1, \ldots, v_n\} \subset \mathbb{N}$ and $S \in \mathbb{N}$, is there a $V' \subseteq V$ whose elements sum to S? More information about SNP systems solving Subset Sum in a uniform and nonuniform way are given in detail in [6].

B. Matrix Representation

In [11] a matrix representation for SNP systems without delay was first introduced. In [4] the representation was modified to include delays. We use the example shown at Figure 1 to illustrate the representation from [4].

Let Π be an SNP system with m neurons and n rules. The following definitions are from [4].

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

For the example in Figure 1, we have the Configuration Vector as $C^0 = \langle 2, 1 \rangle$.

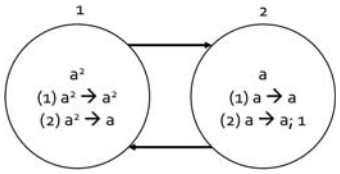

Figure 1. An example of SNP system

Definition 2: The k^{th} status vector is denoted by $St^{(k)} = \langle st_1, ..., st_m \rangle$ such that,

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

For the example in Figure 1, we have the Status Vector as $St^0 = \langle 1, 1 \rangle$. Several Status Vectors can be gathered together by rows in a **Status Matrix**.

Definition 3: The countdown vector at time step k is represented by $D'^{(k)} = \langle d'_1, ..., d'_m \rangle$ such that

$$d'_i = \begin{cases} -1 & \text{if rule } i \text{ is not fired} \\ 0 & \text{if rule } i \text{ is activated} \\ \geq 1 & \text{if rule } i \text{ is fired but has a delay} \end{cases}$$

Note that d' tells us the time remaining before a rule releases its spikes. Several Countdown Vectors can be disposed by rows in a **Countdown Matrix**.

For the example in Figure 1, we have the Countdown Vector as $D'^0 = \langle -1, -1, -1, -1 \rangle$

Definition 4: We then define the **System Configuration** of Π as $Conf^{(k)}$ at time step k with m neurons and n rules by the following:

$$Conf^{(k)} = (C^{(k)}, St^{(k)}, D'^{(k)})$$

For the example in Figure 1, we have the System Configuration as $Conf^0 = (C^0, St^0, D'^0)$.

Definition 5: A **Spiking Matrix** of SNP system Π is defined as follows

$$S^{(k)} = \left[s_{ij}^{(k)} \right]_{q \times n}$$

Where:

$$s_{ij}^{(k)} = \begin{cases} 1 & \text{if } E_j \text{ is satisfied and } r_j \text{ is applied in instance } i \\ 0 & \text{otherwise} \end{cases}$$

and q is the number of valid spiking vectors from a single configuration $Conf^{(k)}$.

For the example in Figure 1, we have the Spiking Matrix as follows:

$$S^0 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Definition 6: An **Indicator Matrix** of system Π is defined as follows

$$IM^{(k)} = \left[im_{ij}^{(k)} \right]_{q \times n}$$

where

$$im_{ij}^{(k)} = \begin{cases} 1 & \text{if rule } r_j \text{ with delay } d \text{ fired at time step} \\ & s \text{ where } s = k - d \text{ (i.e. the rule fired} \\ & \text{and is done with its delay)} \\ 0 & \text{otherwise} \end{cases}$$

For the example in Figure 1, we have the Indicator Matrix as follows:

$$IM^0 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Definition 7: A **Status Matrix** of system Π is defined as follows

$$SM^{(k)} = \left[St_{ij}^{(k)} \right]_{q \times m}$$

where

$$St_{ij}^{(k)} = \begin{cases} 1 & \text{if neuron } \sigma_i \text{ is open at instance } j \\ 0 & \text{otherwise} \end{cases}$$

For the example in Figure 1, we have the Status Matrix as follows:

$$S^0 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Definition 8: The **Transition Matrix** of system Π can be defined as follows

$$TM = \left[tm_{ij} \right]_{n \times m}$$

where

$$tm_{ij} = \begin{cases} p & \text{if rule } r_i \text{ is in neuron } \sigma_s \ (s \neq j \text{ and} \\ & (s, j) \in syn) \text{ and is applied producing } p \text{ spikes} \\ -c & \text{if rule } r_i \text{ is in neuron } \sigma_j \\ & \text{and is applied consuming } c \text{ spikes} \\ 0 & \text{otherwise} \end{cases}$$

From the definition of TM, we define two more matrices based from TM (Production and Consumption) as follows

$$TM^+ = \left[tm_{ij}^+ \right]_{n \times m}, \ TM^- = \left[tm_{ij}^- \right]_{n \times m}$$

where

$$tm_{ij}^+ = \begin{cases} tm_{ij} & \text{if } tm_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}, \ tm_{ij}^- = \begin{cases} tm_{ij} & \text{if } tm_{ij} < 0 \\ 0 & \text{otherwise} \end{cases}$$

For the example in Figure 1, we have the Production and Consumption Transition Matrices as follows:

$$TM^+ = \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \ TM^- = \begin{bmatrix} -2 & 0 \\ -2 & 0 \\ 0 & -1 \\ 0 & -1 \end{bmatrix}$$

Definition 9: The **Net Gain Matrix** is defined as:

$$NG^{(k)} = \left[C^{(k+1)} - C^{(k)} \right]_q$$

i.e. each row for $w, 1 \leq w \leq q$ is a single Net Gain Vector. We can compute for $NG^{(k)}$ by:

$$NG_{q \times m}^{(k)} = St_{q \times m}^{(k)} \otimes (IV_{q \times n}^{(k)} * TM_{n \times m}^+) + S_{q \times n}^{(k)} * TM_{n \times m}^-$$

where $\otimes$ is the elementwise multiplication for matrix.

For the example in Figure 1, we have the Net Gain Vector:

$$NG^0 = \begin{bmatrix} -1 & 1 \\ -2 & -1 \\ -1 & 0 \\ -2 & -1 \end{bmatrix}$$

After solving NG^k, we can now compute for C^{k+1} using the equation $C^{k+1} = C^k + NG^k$. In the example from Figure 1, we can arrive to the next Configuration Vector C^1.

$$C^1 = \begin{bmatrix} 2 & 1 \\ 2 & 1 \\ 2 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} -1 & 1 \\ -2 & -1 \\ -1 & 0 \\ -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 0 & 0 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}$$

II. CuSNP

CuSNP is a simulator for nondeterministic SNP Systems with delays that can run in the CPU or GPU. The GPU simulator is implemented using CUDA C. Algorithm 1 from [4] is used in CuSNP. Algorithm 1 uses a queue, implemented with a 1-dimensional array, to store system configurations so the computation tree is processed in a breadth first manner. The steps are repeated until the computation halts or an arbitrary number of iterations from the user is achieved. The algorithm generates the Indicator, Countdown, and Status matrices based on their definitions above and using the Spiking Vector. For generating the spiking matrix $S_{q \times n}^{(k)}$, we define other vectors and matrices from [4].

Definition 10: The **Applicability Vector** AP for a system Π with m neurons and n rules at timestep k is defined as follows

$$AP^{(k)} = \langle ap_1^{(k)}, ap_2^{(k)}, ..., ap_n^{(k)} \rangle$$

where

$$ap_i^{(k)} = \begin{cases} 1 & \text{if regular expression } E_i \text{ of rule } r_i \text{ is satisfied} \\ & \text{and } r_i \text{ can be fired} \\ 0 & \text{otherwise} \end{cases}$$

Definition 11: The **Count Vector** $CV^{(k)} = \langle cv_1^{(k)}, cv_2^{(k)}, \ldots, cv_m^{(k)} \rangle$ where $cv_i^{(k)}, 1 \leq i \leq m$ contains the number of applicable rules of neuron i based on C^k. We compute the number of possible nondeterministic outcomes q of Π, as follows: $q = \prod_{i=1}^{m} cv_i^{(k)}$.

In Figure 1, we have $AP^0 = \langle 1, 1, 1, 1 \rangle$, $CV^0 = \langle 2, 2 \rangle$ and $q = 4$.

Definition 12: The **Change Frequency Vector** $FV^{(k)}$ where for each $fv_i^{(k)} \in FV^{(k)}, 1 \leq i \leq m$:

$$fv_i^{(k)} = \begin{cases} 1 & \text{if } i = 1 \\ fv_{i-1}^{(k)} * cv_i^{(k)} & \text{otherwise} \end{cases}$$

Definition 13:

Algorithm 1 Main simulation algorithm for a non-deterministic SNP system Π

1: Given a queue Q which initially contains the system configuration $Conf^{(0)}$
Require: Q
2: Get a $Conf^{(k)} = (C^{(k)}_{1\times m}, St^{(k)}_{1\times m}, D'^{(k)}_{1\times n})$ from Q.
3: Generate the Spiking Matrix $S^{(k)}_{q\times n}$, where q is the number of possible nondeterministic outcomes Π has based on $Conf^{(k)}$.
4: Generate the Indicator Matrix $IM^{(k)}_{q\times n}$, $D'^{(k)}_{q\times n}$, and Status Matrix $SM^{(k)}_{q\times m}$,
5: $NG^{(k+1)}_{q\times m} = SM^{(k)} \otimes (IM^{(k)} * TM^{+}) + S^{(k)} * TM^{-}$.
6: Collect all $D'^{(k)}_{1\times n}$ vectors to form matrix $D'^{(k+1)}_{q\times n}$.
7: **for** $d' \in D'^{(k)}$ **do**
8: $d' \leftarrow MAX(d'-1, -1)$
9: **end for**
10: $w \leftarrow 1$
11: **while** $w \leq q$ **or** (C_k **and** $C_{k-1} \neq$ zero vector) **do**
12: Generate a new system configuration $Conf^{(k+1)}_w = \{C^{(k)} + NG_w, SM^{(k+1)}_w, D'^{(k+1)}_w\}$.
13: Add the new configuration to Q
14: **end while**
15: **return** Q

By assigning a total order for each rule r_i that can fire with respect to its neuron i, we have a closed formula for generating SV, i.e. the **Order Vector** $O^{(k)} = \langle o^{(k)}_1, o^{(k)}_2, ..., o^{(k)}_n \rangle$. Based from the definition above, we can generate the Spiking Matrix $S^{(k)}_{(q\times n)}$ by using Algorithm 2.

In Figure 1, we have $FV^0 = \langle 1, 2\rangle$ and $O^0 = \langle 1, 2, 1, 2\rangle$.

Algorithm 2 Compute Spiking Vector

Require: $Conf^{(k)}$
1: Generate $AP^{(k)}, CV^{(k)}, q, FV^{(k)}, O^{(k)}$ from the definitions above
2: $i \leftarrow 1$
3: **while** $i \leq q$ **do**
4: $j \leftarrow 1$
5: **while** $j \leq n$ **do**
6: **if** $ap^{(k)}_i = 1$ and $(o_i - 1) \equiv \lfloor (j-1)/fv^{(k)}_{\sigma_s} \rfloor \bmod cv^{(k)}_{\sigma_s}$ **then**
7: $s^{(k)}_{ij} \leftarrow 1$
8: **else**
9: $s^{(k)}_{ij} \leftarrow 0$
10: **end if**
11: $j \rightarrow j + 1$
12: **end while**
13: $i \rightarrow i + 1$
14: **end while**

III. Current Limitations and Solutions

In this section we focus on extending the CuSNP simulator in [4]. The first limitation in the CuSNP of [4] is simulating an SNP system Π with a maximum of 1024 neurons. This neuron limitation comes from the hardware limitation of CUDA devices only having 1024 threads per block and is also caused by how q is computed in Π from Definition 11. The second limitation from [4] is how the transition matrices (TM^+ and TM^-) tend to be sparse, causing unnecessary space usage. Next we discuss how to improve the CuSNP simulator and fix both limitations.

From the definition of q, it is the product of all the elements in the Count Vector so it requires $O(m)$ time to compute it sequentially. To improve the computation time for q we use a multiblock parallel prefix scan. A scan is a generalized term of the prefix sum wherein the operation is applicable not only to addition but to other binary operations. In this case we apply it to the multiplication operation for computing q. We use a work-efficient, and step-efficient scan algorithms from [10]. For SNP systems with at most 1024 neurons we use the algorithm from [4]. For SNP systems with more than 1024 neurons we use the multiblock parallel prefix scan.

Next, we use an example to demonstrate the prefix scan. Consider an arbitrary input

$$CV = \langle 2, 1, 3, 1, 3, 4, 1, 2, 2, 3, 1, 2, 5, 3, 1, 2, \rangle$$

and consider block size of 4. We put this CV in an array to compute for the prefix scan of each block in CP_0 then retrieve the last element of each block CP_1 as follows: CP_0 = $[2, 2, 6, 6, 3, 12, 12, 24, 2, 6, 6, 12, 5, 15, 15, 30]$, CP_1 = $[6, 24, 12, 30]$

After getting the prefix scan of each block we compute for the prefix scan of the individual block's respective prefix scan, i.e. $CP_2 = [6, 144, 1728, 51840]$. To compute the final prefix scan we retain the first block of the prefix scanned CP_0. To compute the following blocks we multiply CP_2 to the elements one block after the first in CP_0. Now the cumulative product of the array is completed in parallel with the following values.

$$CP = [2, 2, 6, 6, 18, 72, 72, 144, 288, 864, 864, 1728, 8640, 25920, 25920, 51840].$$

SNP systems are typically not fully connected, making their transition matrices sparse. Next we discuss a variation of the Compressed Sparse Row (CSR) format approach based on [8]. In [8] they discuss the use of the CSR format in the transformation of the Transition Matrices. However we modify the CSR representation to only use two resultant arrays instead of three from the original CSR format by transforming the Transition Matrix into a 1-dimensional array and storing only the index of non-zero elements instead of storing the row and column on different arrays in the original CSR format.

Due to how the Transition Matrices TM^+ and TM^- are defined, we need to modify how the reduction of the matrices will be handled to further reduce the number of resultant arrays from four (reductions of both TM^+ and TM^-) to just three.

Algorithm 3 shows how both TM^+ and TM^- are checked for sparsity at the same time. We note that algorithm 3 produces only a partial reduction of the sparse arrays due to how zero elements are checked in line 3. From this, there are cases where the arrays are dense, or TM^+ and TM^- have a

Algorithm 3 Sparse to dense matrix representation of Transition Matrix

INPUT: TM^+ and TM^-
OUTPUT: TMr^+, TMr^-, $TMIndex$

Initialize empty arrays TMr^+, TMr^-, $TMIndex$
for i in $0, ..., N * M$ **do**
  if TM_i^+ or $TM_i^- \neq 0$ **then**
    Append TM_i^+ to TMr^+
    Append TM_i^- to TMr^-
    Append i to $TMIndex$
  end if
end for

zero element in one and non-zero element in the other resulting to at most 50% increase in the total memory footprint of the transition matrices.

$$TM^+ = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 23 & 1 \end{bmatrix} \qquad TM^- = \begin{bmatrix} 0 & 0 & -4 & -2 & -1 \\ 0 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & -3 & 0 \\ 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & -1 & 0 \end{bmatrix}$$

As an example, we use a set of transition matrices given above. Following Algorithm 3 we have the following vectors: $TMr^+ = \langle 1, 1, 0, 0, 0, 1, 1, 0, 2, 23, 1 \rangle$, $TMr^- = \langle 0, 0, -4, -2, -1, -1, -1, -3, -2, -1, 0 \rangle$, $TMIndex = \langle 0, 1, 2, 3, 4, 8, 10, 13, 19, 23, 24 \rangle$. From these values we can compute the compression ratio, i.e. there were 50 elements initially in TM^+ and TM^- that were reduced to 33 elements giving the compression ratio of 50/33 or about 1.51. Note that such transformations in the matrices lead to changes in the algorithms that use the transformed matrices. The algorithms affected in the simulator are generation of the Applicability Vector and computation of Net Gain. A worst case example is shown using the transition matrices derived from Figure 1 where the resulting vectors are $TMr^+ = \langle 0, 2, 0, 1, 1, 0, 1, 0 \rangle$, $TMr^- = \langle -2, 0, -2, 0, 0, -1, 0, -1 \rangle$, $TMIndex = \langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle$ which not only have the same contents as the original transition matrices, but also have an extra vector $TMIndex$, resulting to more memory usage.

Algorithm 4 Generate Applicability Vector

Input: TM^-, Rules, RuleOwner, M
Output: $AP^{(k)}$

Initialize empty vector $AP^{(k)}$
for i in $0, ..., N$ **do**
  $consume \leftarrow TM^-_{i*M+RuleOwner_i}$
  if Rule i can be fired and $C_i^{(k)} + consume \geq 0$ **then**
    $AP_i \leftarrow 1$
  else
    $AP_i \leftarrow 0$
  end if
end for

Algorithm 4 shows how Applicability Vectors are generated using the sparse TM^- array. Algorithm 5 on the other hand makes adjustments on how elements on TM^- are accessed.

Algorithm 5 Generate Applicability Vector with transformed matrix

Input: Transformed TM^-, TMIndex array, Rules, RuleOwner, M
Output: $AP^{(k)}$

Initialize empty vector $AP^{(k)}$
for i in $0, ..., N$ **do**
  if $z \leftarrow BinarySearch(TMIndex, i * M + RuleOwner_i)$ is successful **then**
    $consume \leftarrow TM_z^-$
  else
    $consume \leftarrow 0$
  end if
  if Rule i can be fired and $C_i^{(k)} + consume \geq 0$ **then**
    $AP_i \leftarrow 1$
  else
    $AP_i \leftarrow 0$
  end if
end for

In Algorithm 4, $consume$ gets its value directly from TM^-, so the algorithm makes use of binary search to find whether the target index with respect to its original form exists in the transformed matrix. Non-existence of the target index means that the value of the index in the sparse array is 0.

Algorithm 6 Compute Net Gain Vector

Input: $TM^+, TM^-, S^{(k)}, St^{(k)}, IM^{(k)}, q, M, N$
Output: $NG^{(k)}$

for i in $0, ..., M * q$ **do**
  $r \leftarrow i/M; c \leftarrow i \mod M$
  $pos \leftarrow 0; neg \leftarrow 0$
  for x in $0, ..., N$ **do**
    $pos+ = IM_{r*N+x} * TM^+_{x*M+c}$
    $neg+ = St_{r*N+x} * TM^-_{x*M+c}$
  end for$NG_i \leftarrow (S_i * pos) + neg$
end for

Similar to how the Applicability vector is computed, the change from Algorithm 6 to Algorithm 7 is about how the transition matrices are accessed. In Algorithm 6 the access to the transition matrices are straightforward while binary search is again used to check the existence of the element in Algorithm 7. Note that the changes in the algorithms affect their computation time. For generating the Applicability Vector, the time complexity changed from $O(N)$ to $O(N \log y)$, while on the computation of the Net Gain Vector, the time complexity changed from $O(M * q * N)$ to $O(M * q * N \log y)$ where y refers to the size of the reduced Transition Matrix.

IV. Testing and Results

We use the CuSNP simulator in [4] with $|V|$ for Subset Sum ranging from 3 to 17 due to hardware limitations. Such set instances as inputs are currently sufficient to demonstrate

Algorithm 7 Compute Net Gain Vector using Transformed Transition Matrices

Input: Transformed TMr^+, TMr^-, TMIndex array, $S^{(k)}, St^{(k)}, IM^{(k)}, q, M, N$
Output: $NG^{(k)}$

1: **for** i in $0, ..., M * q$ **do**
2: $r \leftarrow i/M; c \leftarrow i \mod M$
3: $pos \leftarrow 0; neg \leftarrow 0$
4: **for** x in $0, ..., N$ **do**
5: **if** $z \leftarrow BinarySearch(TMIndex, x * M + c)$ is successful **then**
6: $pos+ = TMr_z^+ * IM_{r*N+x}$
7: $neg+ = TMr_z^- * St_{r*N+x}$
8: **end if**
9: **end for** $NG_i \leftarrow (S_i * pos) + neg$
10: **end for**

the changes made to the algorithms. Integers between 50 and 100 inclusive were chosen randomly and a random half of the generated set was chosen to serve as the solution. The same set is used for uniform and nonuniform systems and is averaged over five runs. All simulation results were collected using the NVIDIA Profiler. The simulation was done in a system running on Ubuntu 16.04 with Intel Core i7-4750HQ CPU at 2.00GHz, an NVIDIA GeForce GTX 950M GPU with 640 CUDA cores at 1GHz and 2GB of VRAM, and 8GB of system memory. The simulator is compiled using nvcc on CUDA version 8.0. Nvprof, a profiling tool developed by NVIDIA, is used for measurements. Another simulation for a different set of inputs is made both with the previous hardware discussed and another faster machine running on Ubuntu 16.04 with Intel Core i7-6700 CPU, an NVIDIA GeForce GTX 1070 GPU with 1920 CUDA cores at 1.84GHz with 8GB of VRAM, and 32GB of system memory. Source code for this work is given in http://aclab.dcs.upd.edu.ph/productions/software/cusnp_v170218.

Recall the following distinction between the SNP systems from [6] we use in our experiments, given instance $V = \{v_1, v_2, \ldots, v_n\}$ and S of Subset Sum. On the one hand, uniform solutions have the same set of rules and neurons since such solutions depend only on the size $|V|$ and not on the values of each v_i and S. On the other hand, nonuniform solutions can have more or less neurons and rules for the same size $|V|$ since such solutions depend on the values of $|V|$, each v_i, and S.

First, we compare the average kernel time for the computation of q which involves prefix scan. In Figure 2, we can see that computations involving less than 1024 neurons runs with similar time compared to CuSNP in [4]. The original function is used and the occasional time differences is due to the GPU being used by other computer processes impacting runtime. The increase in runtime with regards to inputs greater than 1024 can be explained due to the function computing q requiring twice the number of reads and writes in the global memory upon implementation of said function, leading to a significant slowdown due to the slower speed of global memory compared to shared memory.

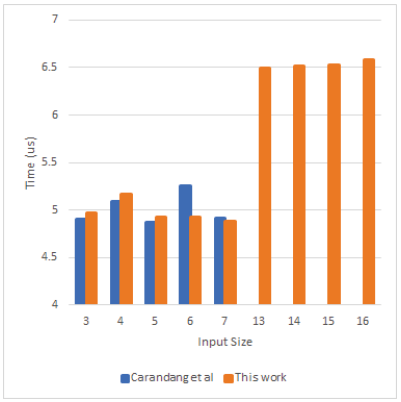

Figure 2. Computation of q Kernel Time Average

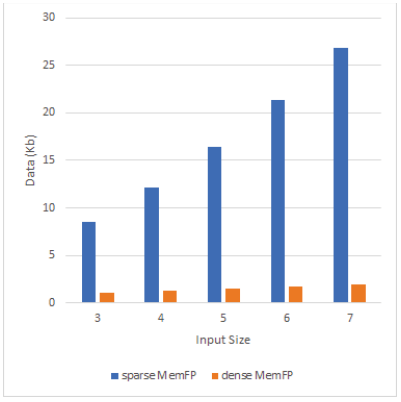

Figure 3. Uniform Memory Footprint

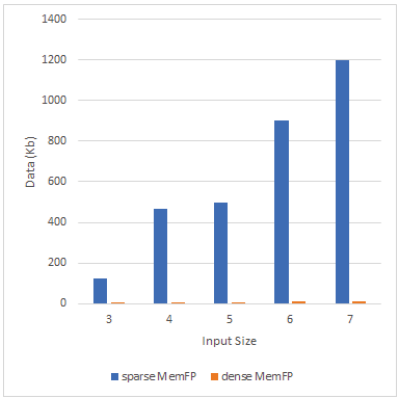

Figure 4. Nonuniform Memory Footprint

Next we show how much memory is reduced when using the dense and transformed Transition Matrices compared to the sparse and original form of the Transition Matrices used by CuSNP in [4]. In Figure 4, we can see a significant reduction of the memory footprint for each input size. We note that these

values are the cumulative memory footprint of the Transition Matrices, i.e. for the sparse version these correspond to both TM^+ and TM^- while for the dense version they are the total memory footprint of TMr^+, TMr^- and $TMIndex$. Other factors affecting the memory footprint of CuSNP not included in this work are part of our future work.

Next we discuss the effect of the dense matrices to the runtime of kernels that use such matrices. In Figures 5 and 6, we can see a significant rise in kernel runtimes on both uniform and nonuniform simulations when generating the Applicability Vector. The same increase can be seen when computing the Net Gain vector, although the runtime is slower due to how each element in the vector is computed. Quantifying the results, we can see that when generating the Applicability Vector, there is 83% to 98% slowdown compared when using sparse matrices. Computing Net Gain Vector on the other hand has a slowdown of 18% to 73%.

We also notice that runtimes of some inputs are faster even when compared to lower input sizes. This is due to the simulator stopping at some configuration C^k and C^{k-1} being zero vectors, so that there are no spikes left in the system and no more computations can be done. There are also certain inputs that lead to a configuration, particularly in the nonuniform solutions, where there is a "large number" of spikes in the output neuron (the output neuron in [6] fires only at exactly S spikes) even if the neuron cannot fire. Such configurations lead to "wasted" computations due to such large values that occupy the GPU in loops for a while.

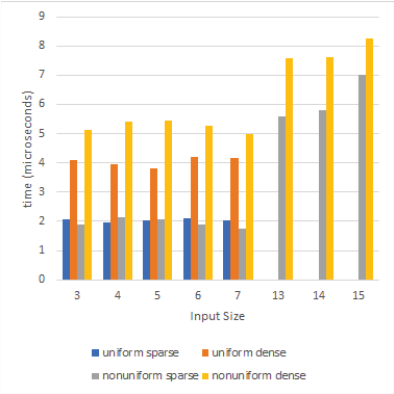

Figure 5. Generate Applicability Vector runtime.

Next we briefly discuss the results of our CuSNP simulator with our second and faster machine, both in CPU and GPU. In this faster machine we can simulate larger inputs, i.e. large values of $|V|$ and v_i, since both CPU and GPU have more memory. The input sets for the nonuniform solutions used are inputs with set size of each V ranging from 3 to 19 and each v_i ranging from 1 to 150 inclusive. The inputs are run in CuSNP with our machines with GTX 950m and GTX 1070 GPUs which we label as machine A and B, respectively. Only nonuniforms solutions are provided in what follows. Although uniform solutions also work with our CuSNP, their runtimes are not provided since their runtimes are much larger due to

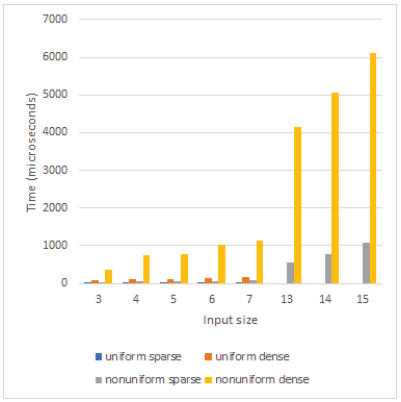

Figure 6. Compute Net Gain runtime.

larger values of v_i used here. Another limitation imposed for this comparison is that runtimes considered for our CuSNP is 20 minutes maximum, while the uniform solutions for such values of v_i can run for more than 1 hour.

In Figures 7,8,9, and 10 we see some trend can be observed in the graphs of the machine A and B with regards to the runtime of the simulator. They both follow a similar curve albeit machine A takes longer time finishing the computation. Similar trends can be seen when generating the Applicability Vector, computation of the Net Gain Vector, and computing for q. These trends are due to machine A having a slower GPU running only at 1GHz compared to machine B at 1.8GHz. Another thing to note is that machine A can only handle inputs up to size 16 while machine B can handle inputs up to size 19. This is due to machine B having more VRAM of 8GB compared to machine A only having 2GB of VRAM. There are also no records regarding size 18 for machine B due to the simulator taking longer than the 20 minute restriction. The situation on input size 18 is due to the output neuron having large number of spikes, e.g. 800, even if the neuron only fires at exactly 450 spikes. This makes the simulator enter a large number of loops, and eliminating such loops are recommended for future work.

It is also noteworthy that the total memory consumption of the GPU is dependent on a hard-coded value called $MAX-Q$ which is the maximum number of possible outcomes the SNP system can have for any configuration. $MAX - Q$ is used for allocating GPU memory of the vectors and matrices of the simulation. A computation for a certain input can run with lower consumed memory if $MAX - Q$ is lowered but may also cause the simulator to fail due to insufficient memory allocation.

V. Conclusions and Future Work

In this work, certain optimisations are made with regards to input limit and memory footprint of the transition matrix of the CuSNP in [4]. A generalised prefix scan was used to simulate SNP systems with more than 1024 neurons unlike in [4]. A modification of CSR is given to form dense matrices in our CuSNP. Changes to the algorithms using the transition

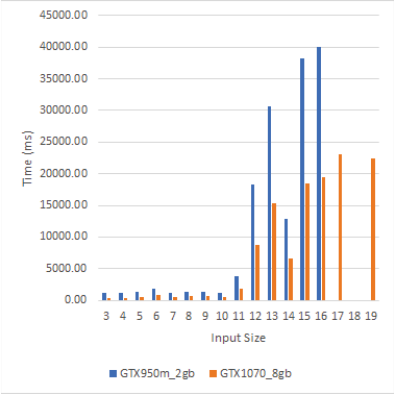

Figure 7. Nonuniform Simulator Runtime

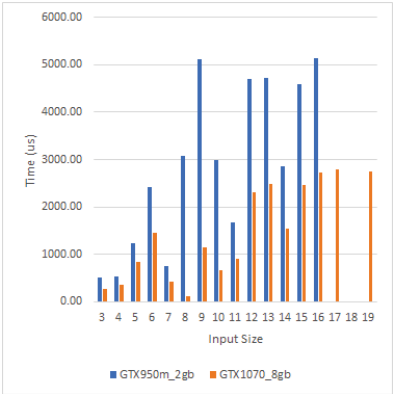

Figure 8. Compute Net Gain Comparison

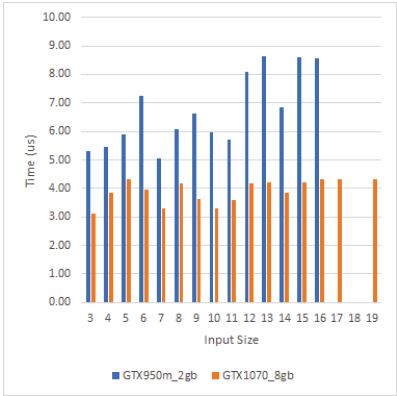

Figure 9. Generating Applicability Vector Comparison

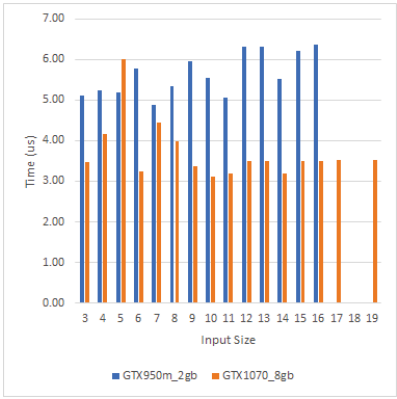

Figure 10. Computation of q comparison

matrices are also presented that shows how the new dense matrices are used during simulation. One motivation in this work is to increase the size of the SNP systems that CuSNP can simulate. In this way, making sure that the memory usage in the GPU of the transition matrices are reduced was necessary. As expected, additional computations for the sparse matrices to our dense matrices result in slower runtimes. We intend to optimise the calculation of q next. In this work and in [4] the value of q is fixed, i.e. independent of any configuration, for the entire simulation. Dynamic calculation of q, upper bounded by the value in this work, can allow simulations of larger SNP sytems. It is also interesting to use other matrix representations and their transformations. Further optimisations in the implementation of CuSNP can also improve runtime, such as halting the simulation immediately depending on the spikes found or not found in the output neuron.

REFERENCES

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

[2] J. P. Carandang, F. G. C. Cabarle, H. N. Adorna, N. H. S. Hernandez, and M. Á. Martínez-del-Amor. Handling Non-determinism in Spiking Neural P Systems: Algorithms and Simulations. *Fundamenta Informaticae*, 164(2-3):139–155, 2019.

[3] J. P. Carandang, J. Villaflores, F. G. C. Cabarle, and H. N. Adorna. CuSNP: Improvements on GPU Simulations of Spiking Neural P Systems in CUDA. In *16th Philippine Computing Science Congress*, pages 77–84, 2016.

[4] J. P. Carandang, J. M. B. Villaflores, F. G. C. Cabarle, H. N. Adorna, and M. Á. Martínez-del-Amor. CuSNP: Spiking Neural P Systems Simulators in CUDA. *Romanian Journal of Information Science and Technology*, 20(1):57–70, 2017.

[5] M. Ionescu, G. Păun, and T. Yokomori. Spiking neural p systems. *Fundamenta Informaticae*, 71(2,3):279–308, Feb. 2006.

[6] A. Leporati, G. Mauri, C. Zandron, G. Păun, and M. J. Pérez-Jiménez. Uniform solutions to SAT and Subset Sum by Spiking Neural P systems. *Natural computing*, 8(4):681, 2009.

[7] M. Á. Martínez-del-Amor, M. García-Quismondo, L. F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, and M. J. Pérez-Jiménez. Simulating P systems on GPU devices: a survey. *Fundamenta Informaticae*, 136(3):269–284, 2015.

[8] M. Á. Martínez-del-Amor, D. Orellana-Martín, F. G. C. Cabarle, M. J. Pérez-Jiménez, and H. N. Adorna. Sparse-matrix Representation of Spiking Neural P Systems for GPUs. *Proc. 15th Brainstorming Week on Membrane Computing 2017*, pages 161–170, 2017.

[9] G. Păun. *Membrane Computing: An Introduction*. Springer Berlin Heidelberg, 2002.

[10] S. Sengupta, A. E. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Workshop on edge computing using new commodity architectures*, pages 26–27, 2006.

[11] X. Zeng, H. N. Adorna, M. A. Martínez-del-Amor, L. Pan, and M. J. Pérez-Jiménez. Matrix representation of Spiking Neural P systems. *Lecture Notes in Computer Science*, 6501:377–391, 2010.