Simulating a Family of Tissue P Systems Solving SAT on the GPU

Miguel A. Martínez-del-Amor, Jesús Pérez-Carrasco, Mario J. Pérez-Jiménez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: `mdelamor@us.es, jesusperezcarrasco@gmail.com, marper@us.es`

Summary. In order to provide efficient software tools to deal with large membrane systems, high-throughput simulators are required. Parallel computing platforms are good candidates, since they are capable of partially implementing the inherently parallel nature of the model. In this concern, today GPUs (Graphics Processing Unit) are considered as highly parallel processors, and they are being consolidated as accelerators for scientific applications. In fact, previous attempts to design P systems simulators on GPUs have shown that a parallel architecture is better suited in performance than traditional single CPUs.

In 2010, a GPU-based simulator was introduced for a family of P systems with active membranes solving `SAT` in linear time. This is the starting point of this paper, which presents a new GPU simulator for another polynomial-time solution to `SAT` by means of tissue P systems with cell division, trading space for time. The aim of this simulator is to further study which ingredients of different P systems models are well suited to be managed by the GPU.

Keywords: Membrane Computing, tissue P systems, SAT, GPU Computing

1 Introduction

Membrane Computing [15] is a recent branch of *Natural Computing*, which defines massively parallel and non-deterministic computing devices abstracted from living *eukaryotic cells*. These devices are called membrane systems, or simply, *P systems* (named after its creator, Gheorghe Păun) [12]. Today, researchers have only one method to work with P systems, which is the usage of simulators running on conventional electronic computers. However, these simulators are normally unfit for very large P systems models. The main reason is that they are not throughput-oriented, so they consume large amounts of time and memory resources on a computer. Therefore, the necessity of efficient simulators arises [15].

In the last years, the trend has been oriented to implement P systems parallelism on parallel platforms, such as *accelerators* (special parallel devices). In fact, *the advent of the accelerators in High Performance Computing offers fresh avenues for developing new and efficient simulators* [2]. One of the most important accelerators nowadays is the *GPU (Graphics Processing Unit)*. It is the core of graphics cards and, thanks to the fast growth of video and game market, typically contains hundreds of slight processors. Their evolution has also led to a new programming model based on data parallelism. This permits to use GPUs for general purpose applications (*GPGPU* or *GPU computing*) [8].

So far, many GPU-based simulators have been developed for several P systems models: *active membranes* [2], *PDP systems* [9], *Spiking Neural P systems* [1], among others. These simulators are flexible for the corresponding P system model, supporting a wide variety of P systems. However, this feature causes a negative effect on performance [10]. An alternative line was initiated in 2010 with the introduction of a specific (*ad-hoc*) simulator for a P system based efficient solution to `SAT` by using GPUs [3, 4]. The solution is based on P systems with active membranes, and the simulator achieves speedups of up to 90x, compared to the CPU counterpart. The obtained results lead to a new open question, related with the efficiency of P system simulators: fixed a problem (e.g. `SAT`), which is the fastest P system based solution simulated on the GPU? In order to answer this question, we first need to analyze which elements of P systems are better suited to be handled by the GPU. In fact, this can help to define new methods to design more efficient simulators.

In this paper, we consider another efficient solution to `SAT` based on *tissue P systems with cell division*. A simulator based on GPUs for this solution is presented. We provide an analysis of performance of the new simulator, together with a performance comparison between the cell-like and the tissue-like simulators.

The paper is structured as follows: Section 2 introduces the model of tissue P systems with cell division and the solution to `SAT`; Section 3 surveys the typical GPU architecture and the peculiarities of GPU computing; Section 4 depicts the design of the new simulator; Section 5 provides the performance analysis of the developed simulator and the mentioned comparisons; and finally, Section 6 ends the paper with conclusions and open research lines.

2 Tissue-like P systems

In this paper, we work on computational devices inspired by the cell inter–communication in tissues, and adding the ingredient of cell division rules. Cell division is an elegant process that enables organisms to grow and reproduce by means of the production of two daughter cells from a single parent cell.

Tissue P systems with cell division are inspired by the cell-like model of P systems with active membranes [13]. In these models, the cells are not polarized; cells obtained by division have the same labels as the original cell, and if a cell is

divided, its interaction with other cells or with the environment is locked during the division process.

First, we recall some preliminaries. An *alphabet* Γ is a non–empty set whose elements are called *symbols*. A *multiset* m over an alphabet Γ is a pair $m = (\Gamma, f)$ where f is a mapping from Γ into $\mathbb{N}$. If $m = (\Gamma, f)$ is a multiset then its *support* is defined as $supp(m) = \{x \in \Gamma \mid f(x) > 0\}$. A multiset is finite if its support is a finite set. If $supp(m) = \{a_1, \ldots, a_k\}$ then the multiset m will be denoted as $m = a_1^{f(a_1)} \ldots a_k^{f(a_k)}$ (here the order is irrelevant), and we say that $f(a_1) + \cdots + f(a_k)$ is the cardinal of m, denoted by $|m|$. The empty multiset is denoted by λ.

Let $m_1 = (\Gamma, f_1)$ and $m_2 = (\Gamma, f_2)$ multisets over Γ. The *union* of m_1 and m_2, denoted by $m_1 + m_2$, is the multiset (Γ, g), where $g = f_1 + f_2$, that is, $g(x) = f_1(x) + f_2(x)$ for each $x \in \Gamma$. The *relative complement* of m_2 in m_1, denoted by $m_1 \setminus m_2$ is the multiset (Γ, g), where $g(x) = f_1(x) - f_2(x)$ if $f_1(x) \geq f_2(x)$ and $g(x) = 0$ otherwise.

Definition 1. *A tissue P system with cell division of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$, where:*

1. *Γ, Σ and $\mathcal{E}$ are finite alphabets such that $\Sigma \subsetneq \Gamma$ and $\mathcal{E} \subseteq \Gamma$.*
2. *Γ has two distinguished objects* `yes` *and* `no`*, and $\{\texttt{yes}, \texttt{no}\} \cap \mathcal{E} = \emptyset$.*
3. *$\mathcal{M}_1, \ldots, \mathcal{M}_q$ are finite multisets over $\Gamma \setminus \Sigma$.*
4. *At least one copy of objects* `yes` *and* `no` *are present in some initial multisets $\mathcal{M}_1, \ldots, \mathcal{M}_q$, but none of them are present in $\mathcal{E}$.*
5. *$\mathcal{R}$ is a finite set of rules of the following forms:*
   (a) Communication rules*: $(i, u/v, j)$, for $i, j \in \{0, 1, 2, \ldots, q\}, i \neq j$, u, v finite multisets over Γ, and $|u + v| \neq 0$;*
   (b) Division rules*: $[a]_i \rightarrow [b]_i[c]_i$, where $i \in \{1, 2, \ldots, q\}$, $i \neq i_{out}$ and $a, b, c \in \Gamma$.*
6. *$i_{in} \in \{1, 2, \ldots, q\}$, and $i_{out} \in \{0, 1, \ldots, q\}$.*

A *tissue P system with cell division* $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$ of degree $q \geq 1$ can be viewed as a set of q cells, labeled by $1, \ldots, q$, with an environment labeled by 0 such that: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are finite multisets over Γ representing the objects (elements in Γ) initially placed in the q cells of the system; (b) $\mathcal{E}$ is the set of objects located initially in the environment of the system, all of them appearing in an *arbitrary number of copies*; and (c) i_{in} represents the input cell, and $i_{out} \in \{0, 1, \ldots, q\}$ represents the *region* (a distinguished cell when $i_{out} \in \{1, \ldots, q\}$, or the environment when $i_{out} = 0$) which will encode the output of the system.

When applying a rule $(i, u/v, j)$, the objects of the multiset u are sent from region i to region j and, simultaneously, the objects of multiset v are sent from region j to region i. When applying a division rule $[a]_i \rightarrow [b]_i[c]_i$, under the influence of object a, the cell with label i is divided into two cells with the same label. In the first copy, object a is replaced by object b, and in the second one,

object a is replaced by object c; all the other objects are replicated and copies of them are placed in the two new cells. The output cell i_{out} cannot be divided.

The rules of a tissue P system with cell division are applied in a non-deterministic maximally parallel manner. At each step, all the cells which can evolve must evolve in a maximally parallel way (at each step, we apply a multiset of rules which is maximal, no further rule can be added), with the following important remark: if a cell divides, only the division rule is applied to that cell at that step; the objects inside that cell do not evolve by means of communication rules.

A *configuration* at any instant of Π is described by all multisets of objects over Γ associated with all the cells present in the system, and the multiset of objects over $\Gamma \setminus \mathcal{E}$ associated with the environment at that moment. Given a finite multiset m over Σ, the *initial configuration* with input m is $(\mathcal{M}_1, \dots, \mathcal{M}_{i_{in}} + m, \dots, \mathcal{M}_q; \emptyset)$. A configuration is a *halting configuration* if no rule of the system is applicable to it.

We say that configuration $\mathcal{C}_1$ yields configuration $\mathcal{C}_2$ in one *transition step* if we can pass from $\mathcal{C}_1$ to $\mathcal{C}_2$ by applying the rules from $\mathcal{R}$ following the previous remarks. A *computation* of Π is a sequence of configurations such that: (a) the first term of the sequence is an initial configuration of the system; (b) each remaining term of the sequence is obtained from the previous one by applying the rules of the system in a maximally parallel manner with the restrictions previously mentioned; and (c) if the sequence is finite (called *halting computation*), then the last term of the sequence is a halting configuration.

A tissue P system with cell division is a *recognizer system* if all computations halt, and if $\mathcal{C}$ is a computation of Π, then either object `yes` or object `no` (but not both) must have been released into the environment, and only at the last step of the computation. We say that $\mathcal{C}$ is an *accepting* (respectively, *rejecting*) *computation* if object `yes` (respectively, object `no`) appears in the environment associated with the corresponding halting configuration of $\mathcal{C}$.

2.1 An efficient solution to SAT by means of tissue P systems with cell division

This section presents an efficient solution to the `SAT` problem by means of family of recognizer tissue P systems with cell division (see [14] for details).

For each pair of natural numbers $m, n \in \mathbf{N}$, we will consider the recognizer tissue P system with cellular division $\Pi(\langle m, n \rangle) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{M}_2, R, 2)$ of degree 2, defined as follows:

- The input alphabet is $\Sigma = \{x_{i,j}, \overline{x}_{i,j} : \ 1 \leq i \leq n, \ 1 \leq j \leq m\}$
- The working alphabet is

$$\begin{aligned}\Gamma = \Sigma & \cup \{a_i, t_i, f_i \mid 1 \leq i \leq n\} \cup \{r_i \mid 1 \leq i \leq m\} \cup \\ & \cup \{T_i, F_i \mid 1 \leq i \leq n\} \cup \{T_{i,j}, F_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m+1\} \cup \\ & \cup \{b_i \mid 1 \leq i \leq 2n+m+1\} \cup \{c_i \mid 1 \leq i \leq n+1\} \cup \\ & \cup \{d_i \mid 1 \leq i \leq 2n+2m+nm+1\} \cup \\ & \cup \{e_i \mid 1 \leq i \leq 2n+2m+nm+3\} \ \cup \ \{f, g, \texttt{yes}, \texttt{no}\}\end{aligned}$$

- The environment alphabet is $\mathcal{E} = \Gamma - \{\texttt{yes}, \texttt{no}\}$.
- The set of labels is $\{1, 2\}$.
- The initial multisets associated with the cells are $\mathcal{M}_1 = \{\texttt{yes}, \texttt{no}, b_1, c_1, d_1, e_1\}$ and $\mathcal{M}_2 = \{f, g, a_1, a_2, \ldots, a_n\}$.
- The input cell is the one labeled by 2, and the output region is the environment.
- The set $\mathcal{R}$ is formed by the following rules:

1. **Division rule:**
   (a) $[\ a_i\ ]_2 \rightarrow [\ T_i\ ]_2 [\ F_i\ ]_2$, for $i = 1, 2, \ldots, n$.
2. **Communication rules:**
   (b) $(1, b_i/b_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (c) $(1, c_i/c_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (d) $(1, d_i/d_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (e) $(1, e_i/e_{i+1}, 0)$, for $i = 1, \ldots, 2n + 2m + nm + 2$.
   (f) $(1, b_{n+1}c_{n+1}/f, 2)$.
   (g) $(1, d_{n+1}/g, 2)$.
   (h*) $(1, f^2/f, 0)$.
   (h) $(2, c_{n+1}T_i/c_{n+1}\ T_{i,1}, 0)$, for $i = 1, \ldots, n$.
   (i) $(2, c_{n+1}F_i/c_{n+1}\ F_{i,1}, 0)$, for $i = 1, \ldots, n$.
   (j) $(2, T_{i,j}/t_i\ T_{i,j+1}, 0)$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.
   (k) $(2, F_{i,j}/f_i\ F_{i,j+1}, 0)$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.
   (l) $(2, b_i/b_{i+1}, 0)$.
   (m) $(2, d_i/d_{i+1}, 0)$, for $i = n + 1, \ldots, 2n + m$.
   (n) $(2, b_{2n+m+1}\ t_i\ x_{i,j}/b_{2n+m+1}\ r_j, 0)$.
   (o) $(2, b_{2n+m+1}\ f_i\ \overline{x}_{i,j}/b_{2n+m+1}\ r_j, 0)$, for $1 \leq i \leq n$ and $1 \leq j \leq m$.
   (p) $(2, d_i/d_{i+1}, 0)$, for $i = 2n + m + 1, \ldots, 2n + m + nm$.
   (q) $(2, d_{2n+m+nm+j}\ r_j/d_{2n+m+nm+j+1}, 0)$, for $j = 1, \ldots, m$.
   (r) $(2, d_{2n+2m+nm+1}/f\ \texttt{yes}, 1)$.
   (s) $(2, \texttt{yes}/\lambda, 0)$.
   (t) $(1, e_{2n+2m+nm+3}\ f\ \texttt{no}/\lambda, 0)$.

Let $\varphi = C_1 \wedge \cdots \wedge C_m$ be a propositional formula in **CNF**[1] such that the set of variables of the formula is $Var(\varphi) = \{x_1, \ldots, x_n\}$, and consists of m *clauses* $C_j = y_{j,1} \vee \cdots \vee y_{j,k_j}$, $1 \leq i \leq m$, where $y_{j,j'} \in \{x_i, \neg x_i : 1 \leq i \leq n\}$ are the literals of φ. Without loss of generality, we can assume that the formula is in simplified expression.

Next, we consider a polynomial encoding (cod, s) of the SAT problem in the family $\mathbf{\Pi} = \{\Pi(t) \mid t \in \mathbb{N}\}$. The function *cod* associates to the previously described propositional formula φ, that is an instance of SAT with parameters n (number of variables) and m (number of clauses), the following multiset of objects

[1] Conjunctive Normal Form

$$cod(\varphi) = \bigcup_{i=1}^{m} \{x_{i,j} : x_i \in C_j\} \cup \{\overline{x}_{i,j} : \neg x_i \in C_j\}$$

In this case, object $x_{i,j}$ represents that variable x_i belongs to clause C_j.

The *size* function, s, is defined as follows $s(\varphi) = \langle m, n \rangle = \frac{(m+n)\cdot(m+n+1)}{2} + m$. The system of the family $\mathbf{\Pi}$ to process the instance φ will be the tissue P system $\Pi(s(\varphi))$ with input multiset $cod(\varphi)$.

The execution of the system $\Pi(s(\varphi))$ with input $cod(\varphi)$ is structured in six phases:

- *Valuations generation phase*: in this phase all the possible relevant truth valuations are generated for the set of variables of the formula $\{x_1, \ldots, x_n\}$. It is implemented by using division rules (a), whereby each object x_i produces two new cells, one having the object T_i, that codifies the true value of the variable x_i, y and the other having the object T_i, that codifies the false value of the variable x_i. Thus, 2^n cells are obtained in n computation steps. These cells are labeled by 2, and each one codifies each possible truth valuation of the set of variables $\{x_1, \ldots, x_n\}$. Meanwhile, the objects f, g are replicated in each created cell. This phase spends n computation steps.
- *Counters generation phase*: simultaneously, and using the rules (b), (c), (d) and (e), the counters b_i, c_i, d_i, e_i of the cell labeled by 1, are evolving such that in each computation step the number of objects in each one are doubling. Thereby, through this process and after n steps, we get 2^n copies of the objects b_{n+1}, c_{n+1}, and d_{n+1}. Objects $b's$ will be used to check which clauses are satisfied for each truth valuation. Objects $c's$ are used to obtain a sufficient number of copies of t_i, f_i (namely, m). Objects $d's$ will be used to check if there is at least one valuation satisfying all clauses. Finally, objects $e's$ will be used to produced, in its case, the object `no` at the end of the computation.
- *Checking preparation phase*: this phase aims at preparing the system for checking clauses. For this, at step $n+1$ of the computation, and by the application of the rules (f) and (g), the counters b_{n+1}, c_{n+1}, d_{n+1} of the cell 1 is exchanged for the objects f and g of the 2^n cells 2. Thus, after this step, each cell labeled by two has a copy of the objects b_{n+1}, c_{n+1}, d_{n+1}, while the cell 1 has 2 copies of the objects f and g.
  Subsequently, the presence of an object c_{n+1} in each one of the 2^n cells labeled by 2 allows to generate the objects $T_{i,1}$ and $F_{i,1}$. By the application of rules (j) and (k), these objects allows the emergence of m copies of t_i and m copies of f_i, according to the values of truth or falsity that a cell 2 assigns to a variable x_i. This process spends $n+m$ steps since there is only one object c_{n+1} in each cell 2 and, moreover, for each $i = 1, \ldots, n$, the rules (j) and (k) are applied exactly m consecutively times. Simultaneously, in the first steps of this process, the application of the rule (h^*) makes the cell labeled by 1 to appear only one copy of the object `yes`.
  Simultaneously in this phase, the counters b_i, d_i and e_i are evolving by the applications of the corresponding rules.

- *Checking clauses phase*: in this phase, the clauses that are true for every truth valuation are determined, and encoded by a cell labeled by 2. This phase starts at the computation step $(n+1)+(n+m)+1 = 2n+m+2$. Using the rules (n) and (o), the true clauses are checked for each valuation encoded by a cell., so that the appearance of an object r_j in a cell 2 means that the corresponding valuation makes true the clause C_j. Bearing in mind that a single copy of the object b_{2n+m+1} is in each cell, the phase takes nm computation steps.
  Thus, the configuration $\mathcal{C}_{2n+m+nm+1}$ is characterized by the following:
  - It contains exactly 2^n cells labeled by 2. Each one contains the object $d_{2n+m+nm+1}$, and copies of objects r_j for each clause C_j made true by the encoded valuation in the cell.
  - It contains a unique cell labeled by 1, containing a copy of objects `yes`, `no`, f, g and the counter $e_{2n+m+nm+2}$.

  This phase consumes m computation steps.
- *Formula checking phase*: in this phase it is determined if there exists any valuation making true the m clauses of the formula. For this, the rules of type (q) are used, analyzing in an ordered way (first the clause C_1, after that clause C_2, and so on) if the clauses of the formula are being satisfied by the represented valuation in the corresponding cell labeled by 2. For example, from counter $d_{2n+m+nm+1}$ appearing in every cell 2, the appearance of the object r_1 (the valuation makes true clause C_1) permits to generate in that cell the object $d_{2n+m+nm+2}$. This object, in turn, permits to evolve object $d_{2n+m+nm+3}$ if in that cell appears the object r_2. In this manner, a valuation represented by a cell labeled by 2 makes true the formula φ if and only if the object $d_{2n+m+nm+m+1}$ appears in the content of that cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$.
- *Output phase*: in this phase the system will provide the corresponding output, depending on the analysis in the formula checking phase.
  If the formula φ is satisfiable, then there is some cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$ that contains an object $d_{2n+m+nm+m+1}$. In this case, the application of rule (r) sends an object f and the object `yes` to the cell 1. The object `yes` therefore disappears from cell 1, and consequently, rule (t) can not be applied. In the next computation step, the application of the rule (s) produces an object `yes` in the environment (for the first time during the whole computation) and the process ends.
  If the formula φ is not satisfiable, then there no exist any cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$ containing an object $d_{2n+m+nm+m+1}$. In this case, the rule (r) is not applicable, and in the next computation step, the counter e_i evolves, providing an object $e_{2n+m+nm+m+3}$ in the cell 1. This object permits the application of rule (t), since the objects `no` and f remains in the cell 1. In this way, the object `no` is sent in the next computation step, and the computation finalizes.

It can be easily proved that the family $\mathbf{\Pi} = \{\Pi(\langle m,n\rangle) : \ n,m \in \mathbf{N}\}$, defined above, is polynomially uniform by deterministic Turing machines. For this, it is enough to keep in mind that the systems of the family have benn defined through

recursive expressions, and the amount of resources needed to describe the system $\Pi(\langle m,n\rangle)$ is quadratic in $\max\{m,n\}$. Indeed:

1. Size of the alphabet: $6nm + 12n + 7m + 12 \in \Theta(nm)$.
2. Number of initial cells: $2 \in \Theta(1)$.
3. Number of initial objects: $n + 8 \in \Theta(n)$.
4. Number of rules: $4nm + 10n + 3m + 16 \in \Theta(nm)$.
5. Upper limit of rule length: $5 \in \Theta(1)$

3 GPU computing

The *GPU (Graphics Processor Unit)* is a specialized chip designed to manipulate computer graphics efficiently. In fact, it is an essential part of most current computers. Their highly parallel structure is based on hundreds of simple computing cores, making them more effective than common CPUs for processing large blocks of data in parallel [8]. Thus, the GPU is being consolidated as a device suitable for *High Performance Computing*, as it was foreseen by Elster [5] and other authors [8].

3.1 CUDA programming model

In 2007, NVIDIA announced *CUDA (Compute Unified Device Architecture)* [17], a programming model totally abstracted from the hardware of the GPU. Based on C, the programmer only has to think on threads and arrays, together with some performance aspects. This easy way to build large applications has led to a rapidly evolution of GPU computing [11]. As a result, CUDA has been successfully utilized for developing P systems simulators [1, 2, 3, 4, 9].

CUDA provides an heterogeneous computing system, consisting of a host (the CPU) and several devices (GPUs) [7]. The idea is to execute on device program sections with large amount of data parallelism. These sections are written in separated C functions called *kernel*. Each kernel is executed on the GPU by a grid of threads. As shown in Figure 1, threads are grouped in *blocks*. Threads belonging to the same block are easily synchronized by barrier operations (when a thread reaches the barrier, wait for the rest to continue).

The memory model is also an aspect to consider in the CUDA programming model. This memory hierarchy is explicitly and manually managed. The *global memory* is the largest (but slowest) in the GPU. It is accessed by the host, and also by any thread, as it is the communication channel between the host and the device. The smallest (but fastest) memory is the *shared memory*. It is local to each block, but the content is volatile through kernels calls, and the CPU cannot read it. Finally, there is a variety of atomic operations to update single data elements in any memory in a concurrently and synchronously way.

An efficient way to structure an algorithm in CUDA is by maximizing the usage of the memory hierarchy, as follows:

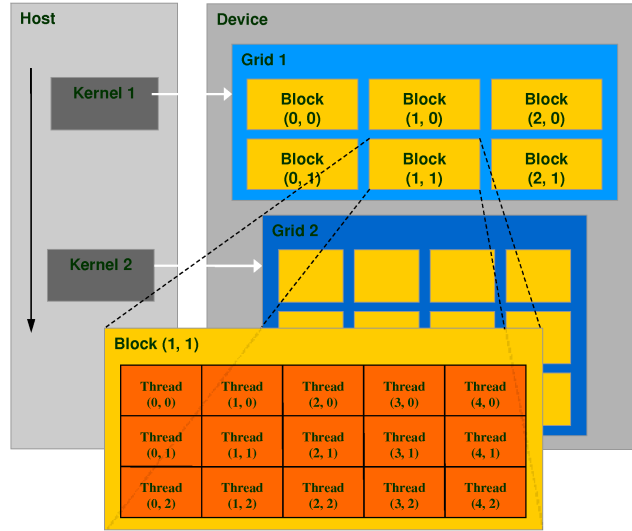

Fig. 1: Overview of CUDA programming model.

1. The threads of each block read its corresponding data portion from global memory to shared memory (which is inevitable because the host only can put the data in global memory).
2. Threads work with the data directly on shared memory.
3. Threads copy these data back to global memory (so the host can retrieve the result).

3.2 A modern GPU architecture

The GPU used in our work is the NVIDIA Tesla C1060. We use this model since it was used in our previous work, and the aim is to compare results. The Tesla C1060 is based on a scalable processor array which has 240 *SPs* (streaming-processor) cores organized as 30 *SMs* (streaming multiprocessor) and 4 GB of off-chip GDDR3 memory called device memory. The applications start at the host side which communicates with the device side through a bus, which is a PCI Express x16 bus standard.

The SM is the processing unit and an unified graphics and computing multiprocessor. Every SM contains the following units: eight SPs arithmetic cores, one double precision unit, an instruction cache, a read only constant cache, 16-Kbyte on-chip read/write shared memory, a set of 16384 32-bit registers, and access to the off-chip memory (device/local memory). The SM also has two SFUs that execute more complex floating point operations such as reciprocal square root,

sine or cosine with low latency. The arithmetic units are capable to execute three instructions per clock cycle, and they are fully pipelined, running at 1,296 GHz, yielding a peak theoretical 933 GFLOPS[2] (240 SP * 3 instructions * 1,296 GHz).

The local and global (device) memory spaces are not cached, which means that every memory access to global memory (or local memory) generates an explicit memory access. A multiprocessor takes 4 clock cycles to issue one memory instruction. Accessing local or global memory incurs an additional 400 to 600 clock cycles of memory latency [7], that is more expensive than accessing share memory and registers (only the mentioned 4 cycles).

A SM is a hardware device specifically designed with multithreaded capabilities. Each SM manages and executes up to 1024 threads in hardware with zero scheduling overhead. Each thread has its own thread execution state and can execute an independent code path. The SMs execute threads in a *SIMT (Single-Instruction Multiple-Thread)* fashion [7]. Basically, in the SIMT model all the threads execute the same instruction on different piece of data. SMs create, manage, schedule and execute threads in groups of 32 threads (which is the branching granularity of NVIDIA GPUs). This set of 32 threads is called *warp*. Each SM can handle up to 32 warps. Individual threads of the same warp must be of the same type and start together at the same program address, but they are free to branch and execute independently at cost of performance.

4 Parallel simulator on the GPU

In this section we describe the developed CUDA simulator. We first explain the data structures and the phases that compound the simulation algorithm. Secondly, the parallel simulator based on CUDA is depicted.

This simulation framework is named *TSPCUDASAT* and published under GNU GPLv3 license. It is enclosed to the software project *PMCGPU (Parallel simulators for Membrane Computing on the GPU)* [18], where the source code of the simulators is available for download.

4.1 Sequential simulation and data structures

For an easier implementation, the simulation algorithm has been divided into five (simulation) phases. Note that they are different in number than the denoted phases of the theoretical model (Section 4 and [14]). This is done to unify phases in the software design. Each of these simulation phases are implemented in code as separated functions whenever is possible. They correspond to the application of certain rules, as explained below:

- *Generation phase*: it performs the application of rules from (a) to (e) of the systems (Section 2.1). Therefore, it comprises the two first phases of the theoretical model: valuations generation phase and counters generation phase.

[2] FLOPS stands for *FLoating-point Operations Per Second*. GFLOPS are giga FLOPS.

- *Exchange phase*: it simulates the application of rules (f) and (g). It comprises the first part of the checking preparation phase.
- *Synchronization phase*: it applies the rules from (h) to (m), so comprising the second part of the checking preparation phase.
- *Checking phase*: it performs the application of rules from (n) to (p). Thus, it is the checking clauses phase we identified in the theoretical model.
- *Output phase*: it applies rules from (q) to (t). It then performs both the formula checking phase and the output phase identified in the theoretical model.

The sequential simulator implements these five simulation phases directly in source code, which is in C++. Each one works directly with the data structures depicted below. The input of the simulator is the same than the one used in the simulator for the cell-like solution [3, 4]. A DIMACS CNF file[3] is provided, and the simulator outputs the response of the computation. Therefore, it acts merely as a `SAT` solver, but the implementation follows the computation of the systems from the family of tissue P systems. Recall that the aim is not to provide a `SAT` solver, but to study P system simulations on the GPU by comparing different solutions to the same problem.

Furthermore, we have adopted a set of enhancements to improve the performance of the sequential simulator. After several tests, we have shown that the best optimizations are:

- As the Exchange phase is very simple, it is then implemented after the Generation phase loop, within the same function.
- We apply the full Synchronization phase to one cell before going to the next one. This allows us to exploit data locality in cache memories.
- In the Checking phase, we orderly insert the objects r_j, for $1 \leq j \leq m$, in the corresponding array whenever they are created. Thus, the Output phase can be easily performed, in such a way that it is not necessary to loop all the objects coming from the input multiset (literals). Now it is enough to check if there exists the m objects r_j.

4.2 Data structures

For this solution, the representation of a tissue P system $\Pi(\langle m, n \rangle)$ is twofold. As the model differentiates between cells labeled by 1 and 2, the design decision was to also have a different data structure representing each cell type in the system. The elements of the cells are encoded within 32-bit integers.

First, cell 1 is represented as an array having a constant dimension of 5 elements. That is, the multiset for cell 1 has the maximum amount of 5 objects: the three counters, b, c and d (which are initially in this cell), and the two objects *yes* and *no* (the final answer to the problem).

Second, the cells labeled by 2 are also represented by a one-dimensional array. All of them are stored inside this large array, since it is initially allocated to store

[3] One of the most adopted input formats by `SAT` solvers.

the maximum amount of cells (2^n). By studying the computation, we conclude that the maximum number of objects appearing in a cell 2 is $(2n) + 4 + |cod(\varphi)|$, where:

- $|cod(\varphi)|$ elements for the initial multiset,
- n elements for objects $T_{i,j}$ and $F_{i,j}$, for $1 \leq i \leq n$ and $1 \leq j \leq m$.
- n elements for objects t_i and f_i, for $1 \leq i \leq n$.
- 4 elements for counter objects a, b, c and d. They will be replaced for counter objects f and g.

The objects are represented similarly to the previous simulator for the cell-like based solution [3]. They are encoded at bit-level within integers of 32 bits, that store the following (8 bits for each field): the name of the object (x or $\overline{x}$), multiplicity of the object (as the multiplicity can exceed 2^8, this field can eventually be joined to the next one), variable (index i) and clause (index j).

1. The name of the object (x or $\overline{x}$)
2. Multiplicity of the object. As there are objects whose multiplicity can exceed 2^8, this field can eventually be joined to the next one (variable).
3. Variable (index i).
4. Clause (index j).

4.3 Design of the parallel simulator

The parallel simulator is designed to also fully reproduce a computation of the systems from the family of tissue P systems. That is, there is no a hybrid[4] solution providing simulation shortcuts to the computation as in [3]. The design of this parallel simulator is driven by the structure of phases explained above, using separate CUDA kernels to speedup the execution of each one.

A similar CUDA work distribution used in other simulators for cell-like solutions [2, 3] is applied. This general assignment is summarized in Figure 2. Each thread block corresponds to each cell labeled by 2 created in the system (up to 2^n cells). However, unlike the previous simulator for the cell-like solution, we do not assign a thread per literal. The assignment of each thread, this time, is different for each simulation phase. The work mapping per phase is therefore as follows:

- *Generation phase*: the number of thread blocks is iteratively increased together with the amount of cells created in each computation step. We distribute cells along the two-dimensional grid through successive kernel calls. Each thread block contains $(2n) + 4 + |cod(\varphi)|$ threads. That is, the amount of elements assigned to each cell in the global array storing multisets. Threads are then used to copy each individual elements of the corresponding cell when it is divided.

[4] A *hybrid simulator* does not perform exactly the same computational steps as the theoretical P system, but achieves the same answer.

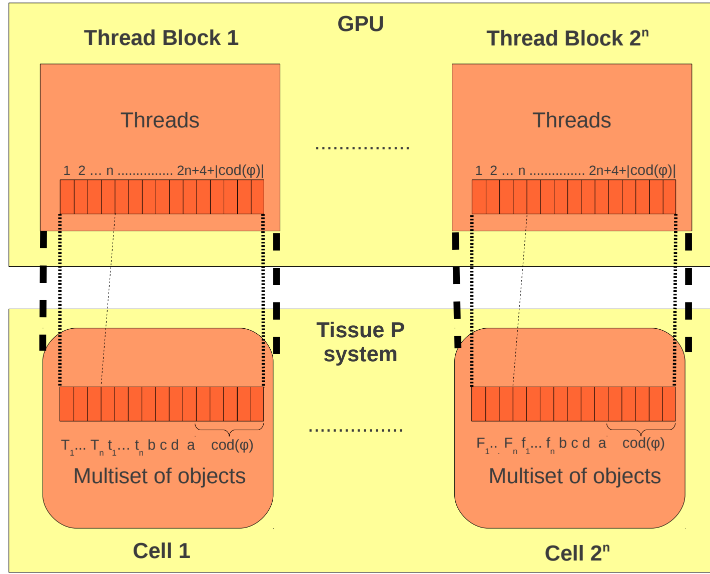

Fig. 2: General design of the parallel simulator.

- *Exchange phase*: it is executed at the kernel for Generation phase, using the same amount of thread blocks, but only the corresponding threads perform the exchange.
- *Synchronization phase*: the thread blocks are assigned to the cells labeled by 2, and the number of threads is n (number of variables). If we use the same amount of threads than in Generation phase, most of them will be idle: it is preferred to launch less threads, but performing effective work. We have experimentally corroborated this fact.
- *Checking phase*: the number of thread blocks is again assigned to be the number of cells labeled by 2. However, for this phase we use a block size of $|cod(\varphi)|$. That is, each thread is used to execute, in parallel, rules of type (n) and (o). The result at the SAT problem resolution level, each thread checks if the corresponding literal makes true its clause, depending on the truth assignment encoded by the cell assigned to the thread block.
- *Output phase*: rules of type (q) are sequentially executed in a separate kernel, again using $|cod(\varphi)|$ threads per block, and 2^n thread blocks.

For this solution, we have applied a small set of improvements, focused on the GPU implementation, to improve the performance of the parallel simulator. We have identified that the simulator runs twice faster than the non-enhanced simulator. We will use the enhanced version of the parallel simulator to perform the comparisons. These improvements are oriented to two performance aspects of GPU computing [11]:

1. The first enhancement type is to *emphasize the parallelism*, which aims to increase the number of threads per block (to the recommended amount from 64 to 1024).
2. The second enhancement type is to *exploit streaming bandwidth.* To do this, the data is first loaded into shared memory and operated there, avoiding global memory (expensive) accesses.

Next, we show the specific enhancement we have carried out for each phase:

- *Generation phase*: no enhancement were implemented here, since the implementation already satisfies the first optimization type. The second type will require a more sophisticated implementation, like the one presented in [4].
- *Exchange phase*: this phase is joined with the generation phase, but has no further enhancements.
- *Synchronization phase*: the two enhancement types are implemented here. The second enhancement type is carried out by using shared memory to avoid global memory accesses. The first type is performed by increasing the number of threads per block. For our simulator, we can assume that n (number of variables, and the number of threads per block) is a small number, since the number of cells grows exponentially with respect to it. For example, let be $n = 32$. Then, 2^{32} cells will be created, what require $2^{32}(68 + |cod(\varphi)|)$ bytes (in gigabytes: $272 + 4|cod(\varphi)|$). This number obviously exceeds the amount of available device memory. We therefore need to increase the number of threads per block, since $n < 32$ means to not fulfilling a CUDA warp. A solution here is to assign more than one cell to each thread block. This amount is $\frac{256}{n}$, being 256 the optimum number of threads per block. It allows us to reach a number of threads close to the optimum one. However, we have to take care also of having enough shared memory to load the data of every assigned cell.
- *Checking phase*: since $|cod(\varphi)|$ can be greater than 32, we then keep this number as the number of threads per block. However, we use shared memory to speedup the accesses to the elements of the array.
- *Output phase*: as in the previous phase, we also use shared memory, and the number of threads per block is kept to $|cod(\varphi)|$.

5 Performance analysis

In this section we show the performance tests carried out for the introduced simulator and for the cell-like based simulator [3]. All experiments are run on a Linux 64-bit server, with a 4-core (2 GHz) dual socket Intel i5 Xeon Nehalem processor, 12 GBytes of RAM, and two NVIDIA Tesla C1060 (240 cores at 1.30 GHz, 4 GBytes of memory).

We have developed two benchmarks (called *test 1* and *test 2*, respectively) to analyze the performance behavior of our simulators in two ways: increasing the number of threads per thread block, and increasing the number of thread blocks

per grid. They are the same than the two used for the cell-like simulators [3]. Both benchmarks have been generated by WinSAT program [16]. WinSAT is able to generate random SAT instances in DIMACS CNF format file by configuring several parameters: the number of variables (n), the number of clauses (m) and the number of literals per clause (we fix k for our experiments).

5.1 Tissue-like simulator

In this subsection, we will see the comparisons of performance between the two simulators developed for the family of tissue-like P systems under study: the sequential simulator (from now on, *tsp-sat-seq*), and the parallel simulator on the GPU (*tsp-sat-gpu*). For this analysis we will use one of the two tests mentioned above: the first one increasing the number of objects (fixing membranes to 2048), and the second increasing the number of variables (and so number of cells) and fixing the number of literals (and so input objects) to 256.

In the first case we can see that, even for small number of objects per membrane, *tsp-sat-gpu* runs faster than *tsp-sat-seq*. A different number of objects does not produce a great impact into the performance of the parallel simulator.

In the second case, we can observe that the kernels of *tsp-sat-gpu* runs faster than *tsp-sat-seq*. However, the performance gain is increased with the amount of cells 2 created by the system. For 64 membranes, the speedup is of 2x, but for 2 M cells it is of 8.3x.

Figure 3 shows the performance behavior of the tissue-like simulators for test 1. Only the time employed by kernels are considered for *tsp-sat-gpu*. We can see that, even for small number of objects per membrane, *tsp-sat-gpu* runs faster than *tsp-sat-seq*. A different number of objects does not produce a great impact into the performance of the parallel simulator. Note that in Section 4, we have introduced a different CUDA design for each phase. In this sense, the synchronization phase has been optimized to assign more cells to a thread block in order to increase the number of threads. However, the speedup is increased together with the number of objects per membrane. This means that the resources of the GPU are better utilized (e.g. 4 objects/threads does not fulfill a warp). We report the maximum speedup for 32 objects (a warp), which is of 11.6x. For 2 objects is 4x, and for 256, 6.1x.

Figure 4 shows the results for test 2, considering only kernel runtime for *tsp-sat-gpu*. For this case, we can observe that again, the kernels of *tsp-sat-gpu* runs faster than *tsp-sat-seq*. However, the performance gain is increased with the amount of cells 2 created by the system. For 64 membranes, the speedup is of 2x, but for 2 M cells it is of 8.3x.

Finally, Figure 5 shows the speedup achieved by the simulator *tsp-sat-gpu*, taking into account also the amount of time consumed by the data management (allocation and transfer). It is observed that, since the data management time is fixed for all the sizes, the speedup exceeds 1 only after 128 K cells. Systems with smaller number of cells are executed slower in the GPU because of the data

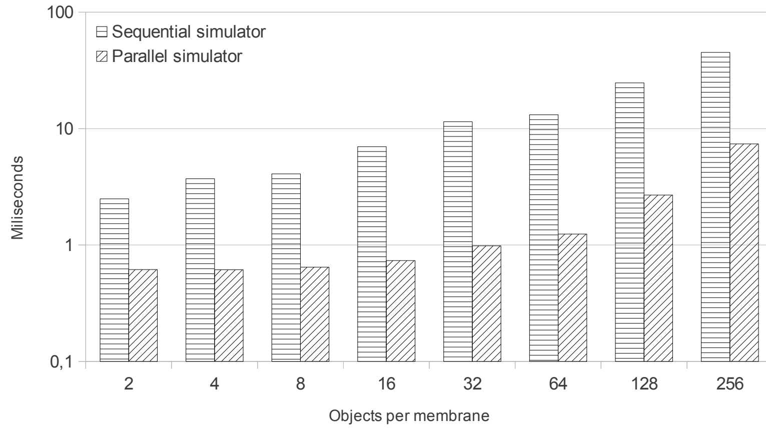

Fig. 3: Simulation performance for *tsp-sat-seq* and *tsp-sat-gpu*: Test 1 (2048 membranes)

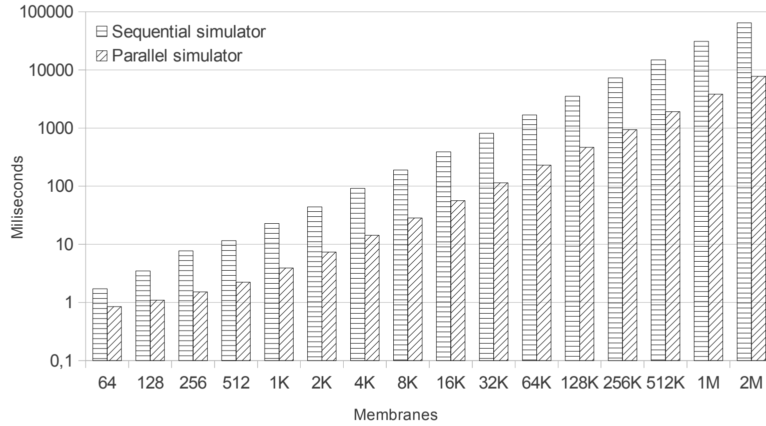

Fig. 4: Simulation performance for *tsp-sat-seq* and *tsp-sat-gpu*: Test 2 (256 Objects/Membrane)

management. However, for very large systems, the speedup is as large as only considering kernels. The maximum speedup is given for 4 M cells, up to 10x.

5.2 Cell-like vs tissue-like

Next, we compare the two simulators developed for the two solutions to `SAT` using P systems with active membranes (let call it *am-sat-gpu*) and tissue P systems with cell division (*tsp-sat-gpu*). Here we study which model is better suited to be simulated on the GPU.

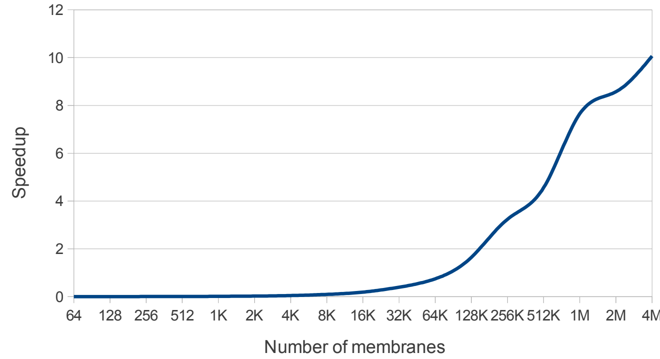

Fig. 5: Speedup achieved running test (256 Objects/Cell) for *tsp-sat-gpu* and *tsp-sat-seq*. GPU data management is also considered.

First of all, we should analyze the differences between them to better understand the different behaviors. We highlight the following:

- Computational steps: given $m, n \in \mathbb{N}$, representing the number of clauses and variables respectively, the cell-like P systems take $5n + 2m + 3$ steps, and the tissue P systems require $2n + 2m + nm + 1$. Thus, the computation of the tissue-like solution is longer (in number of steps), if $m > 3 + \frac{2}{n} \simeq 3$.
- Phases: *am-sat-gpu* is based on 4 phases (implemented in 3 kernels), whereas *tsp-sat-gpu* uses 5 phases (implemented in 4 kernels).
- Memory requirements: each membrane in *am-sat-gpu* is represented by a number of 32-bit integers equals to $|cod(\varphi)|$, but the tissue-like simulators use for them $2n + 4 + |cod(\varphi)|$. Thus, *tsp-sat-gpu* uses, in total, $(2n + 4)2^n$ bytes more.

Figure 6 compares both solutions using Test 2. It can be observed that the kernels of *am-sat-gpu* outperforms *tsp-sat-gpu*, even using optimizations for the last one. This improvement implies a speedup of 2.9x. However, if we take into account the data management in the GPU, we can see that the behavior of them is almost similar. The simulator *am-sat-gpu* runs just a bit faster, but for 2 M membranes, the speedup is almost 2x. This makes us to think that the data implementation of *am-sat-gpu* can be improved, since it requires an inferior amount of data. Recall that *am-sat-gpu* has not any GPU oriented enhancements, as in *tsp-sat-gpu*.

We finish this comparison by reporting their corresponding maximum speedup with their sequential counterparts, which is of 63x and 10x for the cell-like and tissue-like simulators, respectively. Therefore, using the GPU for the cell-like solutions allows to get better performance gain.

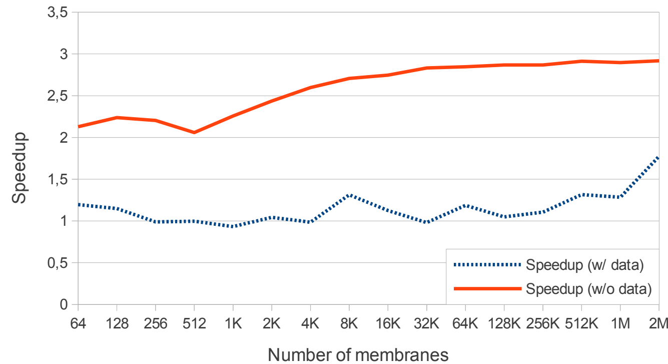

Fig. 6: Achieved speedup for both *tsp-sat-gpu* and *am-sat-gpu* simulators, considering (*w/ data*) or not considering (*w/o data*) the time for data management.

5.3 Characterizing the simulation on the GPU

Next, we characterize the simulations carried out in this work. From the comparison of the simulators for the cell-like and the tissue-like solutions, we have observed that the cell-like simulations are better carried out by the GPU. Thus, we have identified two properties that have helped to improve the performance of these GPU simulators:

- *Charges*: the model of P systems with active membranes associates charges to the membranes. They can be used to store information over the computation as well. If they are considered (and effectively used) for a given solution (e.g. to encode the truth assignment for `SAT`), less memory would be required (remind that the tissue-like simulator requires $2^n(2n + 4)$ bytes more). In fact, the information encoded by charges can save objects that may or not may appear simultaneously in membranes, what saves also memory, and so, the number of threads to launch, working with much less objects.
- *Rules with no cooperation*: the model of P systems with active membranes defines rules with no cooperation, that is, the number of objects appearing in their left-hand sides is always 1. This property helps threads to be assigned to each rule, what also means to work with each object in parallel. Rules permitting cooperation (as in tissue P systems) require to take care of which objects are accessed by rules (and threads). It would be also interesting to study each type of rule (i.e., division, communication for tissue-like, and division, dissolution, send-in, send-out and evolution for cell-like) separately. Recall that a more flexible and general simulator for active membranes [2], the constraints of send-in, send-out, division and dissolution rules have to be considered for each membrane, what degrades parallelism on the GPU (it implies using local locks). However, in the model of tissue P systems these restrictions are not presented. A flexible simulator for tissue models can be implemented in a future

to study what is better: usage of charges but restricting types of rules, or not using charges (i.e. more objects per membrane) and more (but less restrictive) parallel rules.

6 Conclusions

In this paper we have presented a recent result on the parallel simulation with GPUs of an efficient solution to `SAT` by tissue P systems with cell division. The CUDA simulator design is similar to the one used in the previous simulator for a solution based on P systems with active membranes. Each thread block is assigned to each cell labeled by 2. However, the number of objects to be placed inside each cell in the memory representation is increased.

Experiments show that the CUDA simulator outperforms the sequential one by 10x. It can be seen that solving the same problem (`SAT`) under different P system variants leads to different speedups on the GPU (up to 2.9x for the cell-like simulator against the tissue-like). Indeed, we show that the usage of charges can help to save space devoted to objects, and rules without cooperation to increase thread parallelism.

Future work will be focused on developing new GPU-based simulators for other P systems models, and on improving the existing ones. In addition, further research can be carried out concerning the parallel simulation of particular P systems features, identifying which of them can be easily combined and efficiently simulated by the GPU. In this way, novel approximations for parallel simulators development can be performed also at the P systems area. An approach is to define a P system model combining all the good features for GPU simulators (let call it *GP systems*, or GPU oriented P systems). Then, the creation of a GPU based simulator for GP systems would be straightforward, considering the corresponding GPU oriented optimizations. However, it would be important to define a translation protocol from other P systems models to GP systems.

Acknowledgments

The authors acknowledge the support of “Proyecto de Excelencia con Investigador de Reconocida Valía” of the “Junta de Andalucía” under grant P08-TIC04200, and the support of the project TIN2012-37434 of the “Ministerio de Economía y Competitividad” of Spain, both co-financed by FEDER funds.

References

1. F. Cabarle, H. Adorna, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez. Improving GPU Simulations of Spiking Neural P Systems, *Romanian Journal of Information Science and Technology*, **15**, 1 (2012), 5–20.

2. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Simulation of P systems with Active Membranes on CUDA, *Briefings in Bioinformatics*, **11**, 3 (2010), 313–322.
3. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Simulating a P system based efficient solution to SAT by using GPUs, *Journal of Logic and Algebraic Programming*, **79**, 6 (2010), 317–325.
4. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, M. Ujaldón. The GPU on the simulation of cellular computing models, *Soft Computing*, **16**, 2 (2012), 231–246.
5. A. C. Elster. High-performance computing: Past, present and future, *L*NCS, **2367** (2002), 433–444.
6. M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Agustín Riscos-Núñez. An overview of P-Lingua 2.0, *LNCS*, **5957** (2010), 264–288.
7. D. Kirk, W. Hwu. *Programming Massively Parallel Processors: A Hands On Approach*, MA (USA), 2010.
8. M. Harris. Mapping computational concepts to GPUs, *ACM SIGGRAPH 2005 Courses*, NY (USA), 2005.
9. M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez. Population Dynamics P systems on CUDA, *LNCS*, **7605** (2012), 247-266.
10. V. Nguyen, D. Kearney, G. Gioiosa. Balancing performance, flexibility, and scalability in a parallel computing platform for Membrane Computing applications, *LNCS*, **4860** (2007), 385-413.
11. J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips. GPU Computing, *Proceedings of the IEEE*, **96**, 5 (2008), 879-899.
12. G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and TUCS Report No 208.
13. G. Păun. Attacking **NP**-complete problems, In *Unconventional Models of Computation, UMC'2K*, 2000, 94-115.
14. G. Păun, M.J. Pérez-Jiménez, A. Riscos-Núñez. Tissue P System with cell division. *International Journal of Computers, Communications* & *Control*, **3**, 3 (2008), 295-303.
15. G. Păun, G. Rozenberg, A. Salomaa (eds.). *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
16. M. Qasem. WinSAT website, 2009,`http://www.mqasem.net/sat/winsat`.
17. *Official NVIDIA CUDA website*. `http://www.nvidia.com/cuda`
18. *The PMCGPU project website*. `http://sourceforge.net/p/pmcgpu`