View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by idUS. Depósito de Investigación Universidad de Sevilla

# Population Dynamics P Systems on CUDA

Miguel A. Martínez-del-Amor[1], Ignacio Pérez-Hurtado[1], Adolfo Gastalver-Rubio[1], Anne C. Elster[2], and Mario J. Pérez-Jiménez[1]

[1] Research Group on Natural Computing,
Department of Computer Science and Artificial Intelligence,
University of Seville,
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
{mdelamor,perezh,marper}@us.es, adolaurion@hotmail.co.jp
[2] HPC-Lab, Department of Computer and Information Science,
Norwegian University of Science and Technology,
Sem Sælands vei 9, NO-7491, Trondheim, Norway
elster@ntnu.no

**Abstract.** Population Dynamics P systems (PDP systems, in short) provide a new formal bio-inspired modeling framework, which has been successfully used by ecologists. These models are validated using software tools against actual measurements. The goal is to use P systems simulations to adopt a priori management strategies for real ecosystems.

Software for PDP systems is still in an early stage. The simulation of PDP systems is both computationally and data intensive for large models. Therefore, the development of efficient simulators is needed for this field. In this paper, we introduce a novel simulator for PDP systems accelerated by the use of the computational power of GPUs. We discuss the implementation of each part of the simulator, and show how to achieve up to a 7x speedup on a NVIDA Tesla C1060 compared to an optimized multicore version on a Intel 4-core i5 Xeon for large systems. Other results and testing methodologies are also included.

**Keywords:** Ecological Modeling, Population Dynamics, Membrane Computing, Parallel Simulation, GPU Computing, CUDA.

## 1 Introduction

Membrane Computing [19] is part of the broader field of natural or bio-inspired computing. The related computational models are called *P systems*. They are hierarchically distributed models inspired by how membranes compartmentalize living cells into "protected reactors". The model consists of simultaneous applications of rules (abstraction of chemical reactions) over multisets of objects (abstraction of chemical compounds) [20]. Membrane Computing covers both the study of the theoretical basis for the models as well as the applications of the model to various fields including computational Systems Biology [7,22], and Ecosystem Dynamics [3,4,8]. *Population Dynamics P Systems*, or PDP systems, is a P system based framework for modeling population dynamics [3,17] . It

enables simultaneous evolution of a high number of species, as well as the management of a large number of auxiliary objects. It also facilitates model development that can be easily interpreted by simulation software.

So far, several algorithms have been developed in order to capture the semantics defined by the modeling framework. A comparison on the performance of these algorithms can be found in [9]. These algorithms select rules according to their associated probabilities, while keeping the maximal parallelism semantics of P systems. In [17], a new simulation algorithm is presented, called *DCBA* (Direct distribution based on Consistent Blocks Algorithm). It overcomes a common problem on the previous algorithms, regarding a distorted selection of rules. Furthermore, the DCBA was initially implemented and validated using the *P-Lingua* software framework [11,24] which resulted in a simulation Java library (pLinguaCore). However, the simulations were slow (taking hours to run a single simulation), since the pLinguaCore library is not focused on efficiency.

A more efficient implementation based on C++ and OpenMP was presented in [16], taking advantage of modern multicore architectures. These preliminary results indicate that the simulation of PDP systems are memory bound. *GPU computing* [15] has been successfully used to implement other P systems simulators [2,5,6]. By using *CUDA* [13,23], the simulators are accelerated taking advantage of the many-core GPU architecture.

This paper, introduces our new CUDA-optimized PDP systems simulator. We describe how the data structures have been optimized, and how also the code is adapted and restructured in different parts. A performance analysis is also provided, comparing the results with further optimized sequential and multicore versions written in C++/OpenMP.

This paper is structured as follows: Section 2 gives an overview of the P systems based framework that our simulator implements. Section 3 explains the simulation algorithm. Section 4 contains some concepts about the CUDA programming model. Section 5 explains the details of the implementation using CUDA. Section 6 shows some performance results by using a random generator of PDP systems. Finally, conclusions and future work are discussed in Section 7.

## 2 The P Systems Based Framework

**Definition 1.** *A PDP system of degree* $(q, m)$ *& time* $T$ $(q, m, T \geq 1)$ *is a tuple*

$$\Pi = (G, \Gamma, \Sigma, T, R_E, \mu, R, \{f_{r,j} : r \in R, 1 \leq j \leq m\}, \{\mathcal{M}_{ij} : 1 \leq i \leq q, 1 \leq j \leq m\})$$

*where:*

- $G = (V, S)$ *is a directed graph. Let* $V = \{e_1, \ldots, e_m\}$ *whose elements are called environments;*
- $\Gamma$ *is the working alphabet and* $\Sigma \subsetneqq \Gamma$ *is an alphabet representing the objects that can be present in the environments;*

- *$T$ is a natural number that represents the simulation time of the system;*
- *$R_E$ is a finite set of communication rules between environments of the form $r \equiv (x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$, where $x, y_1, \ldots, y_h \in \Sigma$, $(e_j, e_{j_l}) \in S$ $(1 \leq l \leq h)$ and $p_r$ is a computable function from $\{1, \ldots, T\}$ to $[0, 1]$ depending on $x, j, j_1, \ldots, j_h$. If for any rule, $p_r$ is the constant function 1, then we can omit it. These functions verify the following: for each $e_j \in V$ and $x \in \Sigma$, the sum of functions associated with the rules whose left-hand side is $(x)_{e_j}$ is the constant function 1.*
- *$\mu$ is a membrane structure consisting of $q$ membranes injectively labelled by $1, \ldots, q$. The skin membrane is labelled by 1. We also associate electrical charges from the set $\{0, +, -\}$ with membranes.*
- *$R$ is a finite set of evolution rules of the form $r \equiv u[\, v\,]_i^{\alpha} \to u'[\, v'\,]_i^{\alpha'}$, where $u, v, u', v' \in \Gamma^*$, $i$ $(1 \leq i \leq q)$, $u + v \neq \lambda$ and $\alpha, \alpha' \in \{0, +, -\}$.*
  - *If $(x)_{e_j}$ is the left-hand side of a rule from $R_E$, then none of the rules of $R$ has a left-hand side of the form $u[v]_1^{\alpha}$, having $x \in u$.*
- *For each $r \in R$ and for each $j$ $(1 \leq j \leq m)$, $f_{r,j} : \{1, \ldots, T\} \longrightarrow [0, 1]$ is a computable function verifying the following: for each $u, v \in \Gamma^*$, $i$ $(1 \leq i \leq q)$, $\alpha, \alpha' \in \{0, +, -\}$ and $j$ $(1 \leq j \leq m)$ the sum of functions associated with $j$ and the rules whose left-hand side is $u[v]_i^{\alpha}$ and whose right-hand side has polarization $\alpha'$, is the constant function 1.*
- *For each $j$ $(1 \leq j \leq m)$, $\mathcal{M}_{1j}, \ldots, \mathcal{M}_{qj}$ are strings over $\Gamma$, describing the multisets of objects initially placed in the $q$ regions of $\mu$, within the environment $e_j$. It is usual to manage multisets through strings. A finite multiset $m = \{a_1^{f(a_1)}, \ldots, a_k^{f(a_k)}\}$ can be represented by the string $a_1^{f(a_1)} \ldots a_k^{f(a_k)}$ over the alphabet $\{a_1, \ldots, a_k\}$. Nevertheless, all permutations of this string precisely identify the same multiset $m$. Throughout this paper, we speak about "the finite multiset $m$" where $m$ is a string, and meaning "the finite multiset represented by the string $m$".*

That is, a system defined as above can be viewed as a set of $m$ environments $e_1, \ldots, e_m$ interlinked by the edges from the directed graph $G$. Each environment $e_j$ contains a P system, $\Pi_j = (\Gamma, \mu, R, \mathcal{M}_{1j}, \ldots, \mathcal{M}_{q,j})$, of degree $q$, where every rule $r \in R$ has a computable function $f_{r,j}$ associated with it. For each environment $e_j$, we denote by $R_{\Pi_j}$ the set of rules with probabilities obtained by coupling each $r \in R$ with the corresponding function $f_{r,j}$.

A *configuration* of the system at any instant $t$ is a tuple of multisets of objects present in the $m$ environments and at each of the regions of each $\Pi_j$, together with the polarizations of the membranes in each P system. We assume that all environments are initially empty and that all membranes initially have a neutral polarization. We assume a global clock exists, sychromnizing all membranes and the application of all the rules (from $R_E$ and from $R_{\Pi_j}$ in all environments).

The P system can pass from one configuration to another by using the rules from $\bigcup_{j=1}^{m} R_{\Pi_j} \cup R_E$ as follows: at each transition step, the rules to be applied are selected according to the probabilities assigned to them, and all applicable rules are simultaneously applied in a maximal way.

An evolution rule $r \in R$, of the form $u[\,v\,]_i^\alpha \to u'[\,v'\,]_i^{\alpha'}$, is applicable to each membrane labelled by $i$, whose electrical charge is $\alpha$, and it contains the multiset $v$, and its father contains the multiset $u$. When such rule is applied, the objects of the multisets $u$ and $v$ are removed from the father of membrane $i$ and membrane $i$, respectively. Simultaneously, the objects of the multiset $u'$ are added to the father of membrane $i$, and objects of multisets $v'$ are introduced in membrane $i$. The application also replaces the charge of membrane $i$ to $\alpha'$.

A rule $r \in R_E$, of the form $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \dots (y_h)_{e_{j_h}}$, is applicable to the environment $e_j$ if it contains object $x$. When such rule is applied, object $x$ passes from $e_j$ to $e_{j_1}, \dots, e_{j_h}$ possibly modified into objects $y_1, \dots, y_h$ respectively. At any moment $t$ $(1 \leq t \leq T)$ for each object $x$ in environment $e_j$, if there exist communication rules whose left-hand side is $(x)_{e_j}$, then one of these rules will be applied. If more than one such a rule can be applied to an object, the system selects one randomly, according to their probability which is given by $p_r(t)$.

For each $j$ $(1 \leq j \leq m)$ there is just one further restriction, concerning the consistency of charges: in order to apply several rules of $R_{\Pi_j}$ simultaneously to the same membrane, all the rules must have the same electrical charge on their right-hand side.

Following the properties verified by the probabilistic functions, rules in $R$ and $R_E$ can be classified into *blocks of rules*, as showed in definitions 2, 3 and 4.

**Definition 2.** *The left and right-hand sides of the rules are defined as follows:*

**(a)** *Given a rule $r \in R$ of the form $u[v]_i^\alpha \to u'[v']_i^{\alpha'}$ where $1 \leq i \leq q$, $\alpha, \alpha' \in \{0, +, -\}$ and $u, v, u', v' \in \Gamma^*$:*
- *The left-hand side of $r$ is $LHS(r) = (i, \alpha, u, v)$. The charge of $LHS(r)$ is $charge(LHS(r)) = \alpha$. The length of $LHS(r)$ is $|u| + |v|$, what indicates the cooperation degree of the rule.*
- *The right-hand side of $r$ is $RHS(r) = (i, \alpha', u', v')$. The charge of $RHS(r)$ is $charge(RHS(r)) = \alpha'$. The length of $RHS(r)$ is $|u'| + |v'|$.*

**(b)** *Given a rule $r \in R_E$ of the form $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$, the left-hand side of $r$ is $LHS(r) = (e_j, x)$, and the right-hand side of $r$ is $RHS(r) = (e_{j_1}, y_1) \cdots (e_{j_h}, y_h)$.*

**Definition 3.** *Rules from $R$ can be classified into consistent blocks associated with $(i, \alpha, \alpha', u, v)$ as follows:*

$$B_{i,\alpha,\alpha',u,v} = \{r \in R : LHS(r) = (i, \alpha, u, v) \wedge charge(RHS(r)) = \alpha'\}$$

**Definition 4.** *Rules from $R_E$ can be classified into (consistent) blocks associated with $(e_j, x)$ as follows: $B_{e_j,x} = \{r \in R_E : LHS(r) = (e_j, x)\}$.*

Recall that, according to the semantics of our model, the sum of probabilities of all the rules belonging to the same block is always equal to 1; in particular, rules with probability equal to 1 form individual blocks. Note that rules that have exactly the same left-hand side (LHS) belongs to the same block, but rules with overlapping (but different) left-hand sides are classified into different blocks. The latter leads to object *competition*, what is a critical aspect to manage with the simulation algorithms.

**Definition 5.** *Two blocks* $B_{i_1,\alpha_1,\alpha'_1,u_1,v_1}$ *and* $B_{i_2,\alpha_2,\alpha'_2,u_2,v_2}$ *are mutually consistent with each other, if and only if* $(i_1 = i_2 \wedge \alpha_1 = \alpha_2) \Rightarrow (\alpha'_1 = \alpha'_2)$.

## 3 The DCBA

The goal of the DCBA (Direct distribution based on Consistent Blocks Algorithm) [17] is to perform a proportional distribution of objects among competing blocks (with overlapping LHS), determining in this way the number of times that each rule in $\bigcup_{j=1}^{m} R_{\Pi_j} \cup R_E$ is applied. *I.e.* the algorithm simulates the computational steps of a PDP systems. Algorithm 3.1 describes the main loop of the DCBA. It follows the same general scheme as its predecessors, *DNDP* and *BBB* [18] where the simulation of a computing step is structured in two stages: The first stage (selection), selects which rules are to be applied (and how many times) on each environment. The second stage (execution), implements the effects of applying the previously selected rules, yielding the next configuration of the PDP system. Note that, although every $\Pi_j$ has the same set of rules $R$, the probability functions may be different for each environment. See [17] for a more detailed explanation and examples of how to apply this algorithm.

As shown in Algorithm 3.1, the selection stage consists of three phases: Phase 1 distributes objects to the blocks in a certain proportional way, Phase 2 assures the *maximality* by checking the maximal number of applications of each block, and Phase 3 translates block applications to rule applications by calculating random numbers using the multinomial distribution.

**Algorithm 3.1.** DCBA MAIN PROCEDURE

**Require:** A Population Dynamics P system of degree $(q, m)$, $T \geq 1$ (time units), and $A \geq 1$ (*Accuracy*). The initial configuration is called $C_0$.

1: *INITIALIZATION* ▷ (Algorithm 3.2)
2: **for** $t \leftarrow 1$ to $T$ **do**
3: Calculate probability functions $f_{r,j}(t)$ and $p(t)$.
4: $C'_t \leftarrow C_{t-1}$
5: *SELECTION* of rules:
- *PHASE 1*: distribution ▷ (Algorithm 3.3)
- *PHASE 2*: maximality ▷ (Algorithm 3.4)
- *PHASE 3*: probabilities ▷ (Algorithm 3.5)

6: *EXECUTION* of rules. ▷ (Algorithm 3.6)
7: $C_t \leftarrow C'_t$
8: **end for**

The INITIALIZATION procedure (Alg. 3.2) constructs a static distribution table $\mathcal{T}_j$ for each environment. Two variables, $B^j_{sel}$ and $R^j_{sel}$, are also initialized, in order to store the selected multisets of blocks and rules, respectively.

**Observation 1.** *Each column label of the tables* $\mathcal{T}_j$ *contains the information of the corresponding block left-hand side.*

**Observation 2.** *Each row of the tables $\mathcal{T}_j$ contains the information related to the object competitions: for a given object, its row indicates which blocks are competing for it (those columns having non-null values).*

**Algorithm 3.2.** INITIALIZATION

1: Construction of the *static distribution* table $\mathcal{T}$:
   - Column labels: consistent blocks $B_{i,\alpha,\alpha',u,v}$ of rules from $R$.
   - Row labels: pairs $(o,i)$ and $(x,0)$, for all object $o \in \Gamma$, $x \in \Sigma$ and membrane $i$, being 0 the identifier of the environments of the P system.
   - For each row labelled by $(o,i)$ and column labelled by block $B_{i,\alpha,\alpha',u,v}$: place $\frac{1}{k}$ if $o$ appears within $i$ ($0 \leq i \leq q$) with multiplicity $k$ in the LHS of $B_{i,\alpha,\alpha',u,v}$.
2: **for** $j = 1$ **to** $m$ **do** ▷ (Construct the *static expanded* tables $\mathcal{T}_j$)
3: $\mathcal{T}_j \leftarrow \mathcal{T}$. ▷ (Initialize the table with the original $\mathcal{T}$)
4: For each rule block $B_{e_j,x}$ from $R_E$, add a column labelled by $B_{e_j,x}$ to the table $\mathcal{T}_j$; place the value 1 at row $(x,0)$ for that column.
5: Initialize the multisets $B^j_{sel} \leftarrow \emptyset$ and $R^j_{sel} \leftarrow \emptyset$
6: **end for**

The distribution of objects among the blocks with overlapping LHS (competing blocks) is performed in selection Phase 1 (Algorithm 3.3). The expanded static tables $\mathcal{T}_j$ are used for this purpose in each environment, together with three different filter procedures. FILTER 1 discards the columns of the table corresponding to non-applicable blocks due to mismatch charges in the LHS and in the configuration $C'_t$. Then, FILTER 2 discards the columns with objects in the LHS not appearing in $C'_t$. Finally, in order to save space in the table, FILTER 3 discards empty rows. These three filters are applied at the beginning of Phase 1, and the result is a *dynamic table* $\mathcal{T}^t_j$ (for the environment $j$ and time step $t$).

The semantics of the modeling framework requires a set of mutually consistent blocks before distributing objects to the blocks. For this reason, after applying FILTERS 1 and 2, the mutually consistency is checked. Note that this checking can be easily implemented by a loop over the blocks. If it fails, meaning that an inconsistency was encountered, the simulation process is halted, providing a warning message to the user. Nevertheless, it could be interesting to find a way to continue the execution by non-deterministically constructing a subset of mutually consistent blocks. Since this method can be exponentially expensive in time, it is optional for the user whether to activate it or not.

Once the columns of the *dynamic table* $\mathcal{T}^t_j$ represent a set of mutually consistent blocks, the distribution process starts. This is carried out by creating a temporal copy of $\mathcal{T}^t_j$, called $\mathcal{TV}^t_j$, which stores the following products:

- The normalized value with respect to the row: this is a way to *proportionally* distribute the corresponding object along the blocks. Since it depends on the multiplicities in the LHS of the blocks, the distribution, in fact, penalizes the blocks requiring more copies of the same object. This is inspired in the amount of energy required to gather individuals from the same species.
- The value in the dynamic table (i.e. $\frac{1}{k}$): this indicates the number of possible applications of the block with the corresponding object.

– The multiplicity of the object in the configuration $C'_t$: this performs the distribution of the number of copies of the object along the blocks.

**Algorithm 3.3.** SELECTION PHASE 1: DISTRIBUTION

**for** $j = 1$ **to** $m$ **do** ▷ (For each environment $e_j$)
  Apply filters to table $\mathcal{T}_j$ using $C'_t$, obtaining $\mathcal{T}_j^t$. The filters are applied as follows:
    **a.** $\mathcal{T}_j^t \leftarrow \mathcal{T}_j$
    **b.** FILTER 1 $(\mathcal{T}_j^t, C'_t)$.
    **c.** FILTER 2 $(\mathcal{T}_j^t, C'_t)$.
    **d.** Check *mutual consistency* for the blocks remaining in $\mathcal{T}_j^t$. **If** there is at least one inconsistency **then** report the information about the error, and optionally halt the execution (in case of not activating step *3.*)
    **e.** FILTER 3 $(\mathcal{T}_j^t, C'_t)$.
  *(OPTIONAL)* Generate a set $S_j^t$ of sub-tables from $\mathcal{T}_j^t$, formed by sets of *mutually consistent* blocks, in a maximal way in $\mathcal{T}_j^t$ (by the inclusion relationship). Replace $\mathcal{T}_j^t$ with a randomly selected table from $S_j^t$.
  $a \leftarrow 1$
  **repeat**
    **for all** rows $X$ in $\mathcal{T}_j^t$ **do**
      $RowSum_{X,t,j} \leftarrow$ total sum of the non-null values in the row $X$.
    **end for**
    $\mathcal{TV}_j^t \leftarrow \mathcal{T}_j^t$ ▷ (A temporal copy of the dynamic table)
    **for all** non-null positions $(X, Y)$ in $\mathcal{T}_j^t$ **do**
      $mult_{X,t,j} \leftarrow$ multiplicity in $C'_t$ at $e_j$ of the object at row $X$.
      $\mathcal{TV}_j^t(X, Y) \leftarrow \lfloor mult_{X,t,j} \cdot \frac{(\mathcal{T}_j^t(X,Y))^2}{RowSum_{X,t,j}} \rfloor$
    **end for**
    **for all** not filtered column, labelled by block $B$, in $\mathcal{T}_j^t$ **do**
      $N_B^a \leftarrow \min_{X \in rows(\mathcal{T}_j^t)}(\mathcal{TV}_j^t(X, B))$ ▷ (The minimum of the column)
      $B_{sel}^j \leftarrow B_{sel}^j + \{B^{N_B^a}\}$ ▷ (Accumulate the value to the total)
      $C'_t \leftarrow C'_t - LHS(B) \cdot N_B^a$ ▷ (Delete the LHS of the block.)
    **end for**
    FILTER 2 $(\mathcal{T}_j^t, C'_t)$
    FILTER 3 $(\mathcal{T}_j^t, C'_t)$
    $a \leftarrow a + 1$
  **until** $(a > A) \vee$ *(all the selected minimums at step 15 are* 0*)*
**end for**

**Algorithm 3.4.** SELECTION PHASE 2: MAXIMALITY

**for** $j = 1$ **to** $m$ **do** ▷ (For each environment $e_j$)
  Set a random order to the blocks remaining in the last updated table $\mathcal{T}_j^t$.
  **for all** block $B$, following the previous random order **do**
    $N_B \leftarrow$ number of possible applications of $B$ in $C'_t$.
    $B_{sel}^j \leftarrow B_{sel}^j + \{B^{N_B}\}$ ▷ (Accumulate the value to the total)
    $C'_t \leftarrow C'_t - LHS(B) \cdot N_B$ ▷ (Delete the LHS of block $B$, $N_B$ times.)
  **end for**
**end for**

After the object distribution process, the number of applications for each block is calculated by selecting the minimum value in each column. This number is then used for consuming the LHS from the configuration. However, this application could be not maximal. The distribution process can eventually deliver objects to blocks that are restricted by other objects. As this situation may occur frequently, the distribution and the configuration update process is performed $A$ times, where $A$ is an input parameter referring to *accuracy*. The more the process is repeated, the more accurate the distribution becomes, but the performance of the simulation decreases. We have experimentally checked that $A = 2$ gives the best accuracy/performance ratio. In order to efficiently repeat the loop for $A$, and also before going to the next phase (maximality), it is interesting to apply FILTERS 2 and 3 again.

After phase 1, it may be the case that some blocks are still applicable to the remaining objects. This may be caused by a low $A$ value or by rounding artifacts in the distribution process. Due to the requirements of P systems semantics, a maximality phase is now applied (Algorithm 3.4). Following a random order, a maximal number of applications is calculated for each block still applicable.

After the application of phases 1 and 2, a maximal multiset of selected (mutually consistent) blocks has been computed. The output of the selection stage has to be, however, a maximal multiset of selected rules. Hence, Phase 3 (Algorithm 3.5) passes from blocks to rules, by applying the corresponding probabilities (at the local level of blocks). The rules belonging to a block are selected according to a multinomial distribution $M(N, g_1, \ldots, g_l)$, where $N$ is the number of applications of the block, and $g_1, \ldots, g_l$ are the probabilities associated with the rules $r_1, \ldots, r_l$ within the block, respectively.

**Algorithm 3.5.** SELECTION PHASE 3: PROBABILITY

1: **for** $j = 1$ **to** $m$ **do** ▷ (For each environment $e_j$)
2: **for all** block $B^{N_B} \in B^j_{sel}$ **do**
3: Calculate $\{n_1, \ldots, n_l\}$, a random multinomial $M(N_B, g_1, \ldots, g_l)$ with respect to the probabilities of the rules $r_1, \ldots, r_l$ within the block.
4: **for** $k = 1$ **to** $l$ **do**
5: $R^j_{sel} \leftarrow R^j_{sel} + \{r_k^{n_k}\}$.
6: **end for**
7: **end for**
8: Delete the multiset of selected blocks $B^j_{sel} \leftarrow \emptyset$. ▷ (Useful for the next step)
9: **end for**

Finally, the execution stage (Algorithm 3.6) is applied. This stage consists on adding the RHS of the previously selected multiset of rules, as the objects present on the LHS of these rules have already been consumed. Moreover, the indicated membrane charge is set.

**Algorithm 3.6.** EXECUTION

1: **for** $j = 1$ **to** $m$ **do** ▷ (For each environment $e_j$)
2: **for all** rule $r^n \in R^j_{sel}$ **do** ▷ (Apply the RHS of selected rules)
3: $C'_t \leftarrow C'_t + n \cdot RHS(r)$
4: Update the electrical charges of $C'_t$ from $RHS(r)$.
5: **end for**
6: Delete the multiset of selected rules $R^j_{sel} \leftarrow \emptyset$. ▷ (Useful for the next step)
7: **end for**

# 4 The CUDA Programming Model

With the commercial sector's demands for video and gaming, it was foreseen by Elster [10] and others that graphics processor development would lead to devices suitable for High Performance Computing (HPC). With the introduction of NVIDIA's CUDA [13] and AMD's Stream SDK environments in 2007, the GPUs became more easily programmable and the era of GPGPU (General-Purpose Computing on Graphics Processing Units) truly began. GPUs can now be considered affordable computing solutions for speeding up computationally demanding applications. GPUs today typically have several hundred computational cores. By parallelizing and optimizing our codes for these cores, we have shown that GPUs can speed up applications ranging from smoothed hydrodynamics (SPH) [14] and real-time gradient vector flows to linear programming [21] and can even be used as an accelerator to compress I/O data for faster I/O speeds [1]. As was mentioned earlier, we have also successfully used GPUs to implement other P systems simulators [2,5,6].

## 4.1 Compute Unified Device Architecture

NVIDIA's *CUDA* (Compute Unified Device Architecture) provides developers with a high-level programming model that allows developers to take full advantage of the NVIDIA's powerful GPU hardware. To a CUDA programmer [23], the computing system is heterogeneous, consisting of a host (the CPU), and one or more massively parallel many-core devices (GPU systems). In modern software applications, there are often program sections that exhibit rich amount of data parallelism, a property where many arithmetic operations can be safely performed simultaneously on the program data structures. GPUs accelerate the execution of these applications by harvesting a large amount of data parallelism.

CUDA is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). The SM creates, manages, and executes concurrent threads (extra light weight processes) in hardware with with virtually no overhead [23]. This allows very fine-grained decomposition of problems by assigning, for instance, one thread to each data element. Each *threads* has a unique *threadIdx*, and is grouped in 1D, 2D, or 3D *blocks* which are organized in 1D or 2D *grids*. All the threads execute the same code, called *kernel*.

Since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core, current GPUs,

are limited to a maximum of 1024 threads. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. These features make CUDA an interesting choice for developing simulators for the area of Membrane Computing, as previously demonstrated for active membranes [5], a P systems based solution to SAT [6] and for spiking neural P systems [2].

## 5 DCBA Implementation on the GPU

The DCBA was first implemented inside the pLinguaCore framework [17,11,24]. This version (hereafter *pdp-plcore-sim*) was validated by a real ecosystem model [17], reproducing the same data as the actual measurements. However, the performance was slow since it as part of the pLinguaCore was written in Java.

Our first approach for making our implementation more efficient, was to develope a *stand alone* simulator written in C++. We then improved performance further by using OpenMP to take advantage of modern multi-core architectures, such as the Intel's i5 Nehalem and i7 Sandy (*pdp-omp-sim*). *Pdp-omp-sim* achieved speedups of up to 2.5x on a 4-core Intel i7. These preliminary results indicated that the simulations of PDP systems are memory bound.

Our previous simulator, *pdp-omp-sim*, is the starting point of the new implementation using CUDA. In this new simulator (let call it *pdp-gpu-sim*), the code and the data structures have been optimized, saving up to 27% of memory. We have also adapted *pdp-omp-sim* to these, achieving better speedups (1.25x for large systems).

Normally, the end user (*i.e.* ecological experts and model designers) runs many simulations on each set of parameters to extract statistical information of the probabilistic model. This can be automated by adding a outermost loop for simulations in Algorithm 3.1. This loop is easily parallelized. Indeed, our tests of *pdp-omp-sim* conclude that parallelizing by simulations or a hybrid technique (simulations plus environments) yields the largest speedups.

At first glance, these two levels of parallelism (simulations and environments) could fit the double parallelism of the CUDA architecture (thread blocks and threads). For example, we could assign each simulation to a block of threads, and each environment to a thread (since they require synchronization at each time step). However, the number of environments depends inherently on the model. Typically, 2 to 20 environments are considered, which is not enough for fulfilling the GPU resources. Number of simulations typically range from 50 to 100, which is sufficient for thread blocks, but still a poor number compared to the several hundred cores available on modern GPUs.

We therefore also parallelize the execution of rule blocks. Our simulator can hence utilize a huge number of thread blocks by distributing simulations (parallel simulations, as memory can store them) and environments in each one, and process each rule block by each thread. Since there are normally more rule blocks (thousand of them) than threads per thread block (up to 512), we create 256 threads which iterate over the rule blocks in tiles. This design is graphically

shown on Figure 1. Each phase of the algorithm has been designed following the general CUDA design explained above, and implemented separately as individual kernels. Thus, simulations and environments are synchronized by the successive calls to the kernels.

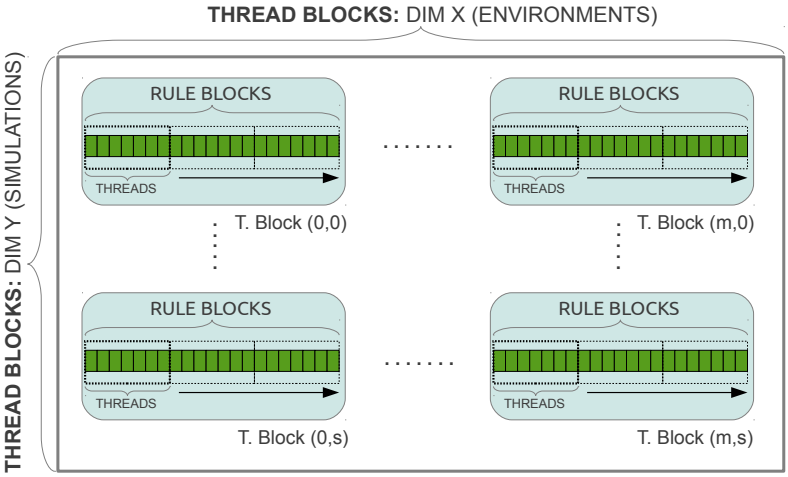


**Fig. 1.** General design of our CUDA-based simulator: 2D grid, and 1D thread blocks. Threads loop the rule blocks in tiles.

### 5.1 Implementation of Selection Phase 1

The main challenge at this phase is the construction of the expanded static table $\mathcal{T}_j$. The size of this table is of order $O(|B|\cdot|\Gamma|\cdot(q+1))$, where $|B|$ is the number of rule blocks, $|\Gamma|$ is the size of the alphabet (amount of different objects), and $q+1$ corresponds to the number of membranes plus the space for the environment.

A full implementation of $\mathcal{T}_j$ can be expensive for large PDP systems. Moreover, it is a sparse matrix, having null values in the majority of the positions: competitions for one object appears for a relatively small number of blocks. This problem was overcome in the *pdp-plcore-sim* by using a hash table storing only non-null values. For *pdp-omp-sim*, the idea was to avoid the construction of $\mathcal{T}_j$, by translating the operations over the table to operations directly to the rule blocks information (using the observations made in section 3):

- Operations over columns: they can be transformed to operations for each rule block and the objects appearing in the multisets of the LHS.
- Operations over rows: they can be translated similarly to operations over rows, but the partial results into a global array (one position per row).

Phase 1 can be implemented as described in Algorithm 5.1. Note that Filter 3 is not needed any more. Although the full table is not created, some auxiliary data structures are used to virtually simulate it (we say it uses a *virtual table*):

- $activationVector$: the information of filtered blocks is stored here as boolean values. The full global size is of order $O(|B| * m * nsim)$, where $m$ is the number of environments and $nsim$ the number of simulations carried out in parallel. This vector is actually implemented passing from boolean to bits.
- $addition$: the total calculated sums for rows are stored here, one number per each pair object and region. Its size is of order $O(|\Gamma| * (q+1) * m * nsim)$.
- $MinN$: the minimum numbers calculated per column are stored here. This is needed in order to substract the corresponding number of applications to $C'_t$ in each loop for the $A$ value. The full global size is of order $O(|B| * m * nsim)$.
- $BlockSel$: the total number of applications for each rule block is stored here. The full global size is of order $O(|B| * m * nsim)$.
- $RuleSel$: the total number of applications for each rule is stored here. The full global size is of order $O(((|R| * m) + |R_E|) * nsim)$, where $|R|$ is the number of rules and $|R_E|$ the number of communication rules.

**Algorithm 5.1.** Implementation of selection Phase 1 with virtual table

1: **for** $j = 1, \ldots, m$ **do** ▷ For each environment
2: **for all** block $B$ **do**
3: $activationVector[B] \leftarrow true$
4: **if** $charge(LHS(B))$ is different to the one presented $C'_t$ **then**
5: $activationVector[B] \leftarrow false$ ▷ (Apply Filter 1)
6: **else if** one of the objects in $LHS(B)$ does not exist in $C'_t$ **then**
7: $activationVector[B] \leftarrow false$ ▷ (Apply Filter 2)
8: **end if**
9: **end for**
10: Check the mutually consistency of blocks.
11: **repeat**
12: **for all** block $B$ having $activationVector[B] = true$ **do** ▷ (Normalization 1)
13: **for** each object $o^k$ appearing in $LHS(B)$, associated to region $i$ **do**
14: $addition[o,i] \leftarrow addition[o,i] + k$
15: **end for**
16: **end for**
17: **for all** block $B$ having $activationVector[B] = true$ **do** ▷ (Normalization 2)
18: $MinN[B] \leftarrow Min_{[o^k]_i \in LHS(B)}(\frac{1}{k^2} * \frac{1}{addition[o,i]} * C'_t[o,i])$.
19: $BlockSel[B] \leftarrow BlockSel[B] + MinN[B]$.
20: **end for**
21: **for all** block $B$ having $activationVector[B] = true$ **do** ▷ (Updating)
22: $C'_t \leftarrow C'_t \ - \ LHS(B) * MinN[B]$
23: **end for**
24: Apply Filter 2 again (as described in step 6).
25: $a \leftarrow a + 1$
26: **until** $a = A$ **or** *for each active block* $B$, $MinN[B] = 0$
27: **end for**

The implementation on the device has been constructed directly from Algorithm 5.1. Phase 1 has been implemented using several kernels, avoiding the overload of only one:

– Kernel for Filters (from line 2 to 10 in Algorithm 5.1): FILTERS 1 and 2 are implemented here by using our general CUDA design (Figure 1).
– Kernel for Normalization (from line 11 to 20): the two parts (row additions and minimum calculations) of the normalization step is implemented in a kernel. The two parts are synchronized by *synchtreads* CUDA instruction. The work assigned to threads is divergent; that is, each thread works with one rule block, but writes information for each object appearing in the LHS. Therefore, the writes to *addition* are carried out by atomic operations.
– Kernel for Updating and FILTER 2 (from line 21 to 26). As before, the work of each thread is divergent. Thus, the update of the configuration is also implemented with atomic operations.

### 5.2 Implementation of Selection Phase 2

Phase 2 is the most challenging part when parallelizing by blocks. The selection of blocks at this phase is performed in an inherently sequential way: we need to know how many objects a block can consume before selecting the next one. In our solution, Phase 2 is implemented by one kernel, using our general CUDA design.

The random order to the blocks is *simulated* by the CUDA thread scheduler: each thread calculates the position in the order of its rule block by using the *atomicInc* operation. Since it does not perform a real random order, random numbers are going to be used soon in next versions. Our first approach (let designate it *ph2-simorder-oneseq*) for phase 2 was to launch 257 threads: 256 threads to calculate the "random" order, and an extra thread to iterate the blocks in that order, selecting and consuming the LHS. Since this approach is still sequentially executed in the GPU, an improved version was constructed.

Our new version (designated *ph2-simorder-dyncomp*) dynamically checks the blocks that are really competing for objects, and calculates which blocks can be selected in parallel, and which depend on the selection of the others. To do this,

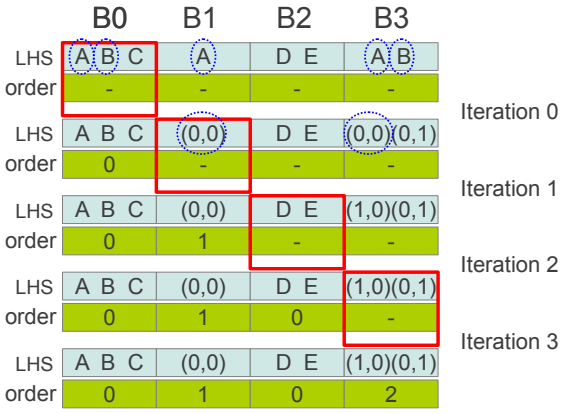


**Fig. 2.** Sample of our *ph2-simorder-dyncomp* kernel execution

some previous computations are needed. Two arrays are used, one for storing the information of the LHS, and another to store the order of selection (rule blocks having the same order number will be selected in parallel). Both of the arrays are implemented using the GPUs shared memory to speedup this computation. Shared memory on the GPU is one of the on-chip memory spaces which is shared by all the SPs of a SM. Access times to the shared memory are comparable to those of a L1-cache on a traditional CPU. (GPUs also feature high-speed DRAM memory (device memory) with higher latency than on-chip memory (typically hundreds of times slower). The device memory is subdivided in read-write, non-cached (global and local) and read-only, cached (texture) areas.)

Figure 2 shows a sample *ph2-simorder-dyncomp* kernel execution. We iterate for each rule block (using the pre-calculated random order). First, the rule blocks check if they have common objects with block $B0$. In the example, block $B1$ has object $A$, and block $B3$ has objects $A$ and $B$. They annotate this competition with the pair $(block, object)$, using the indexes of the array. The current block also calculates the selection order by checking if it has some depending objects. If so, the order is increased by one. For the first iteration, block $B0$ is assigned order 0, but in iteration 1, block $B1$ is assigned order 1 (competing with block $B0$). The rest of the iteration can be seen in Figure 2.

Our experiments shows that *ph2-simorder-dyncomp*, that includes extra computations but allows to execute independent blocks in parallel, achieves up to 20% of performance improvement from *ph2-simorder-oneseq*.

### 5.3 Implementation of Selection Phase 3

Phase 3 calculates the number of times a rule is applied using a binomial distribution, and the selected block number, both implemented in one kernel.

For random binomial number generation, we have made a *CUDA library* based on *CuRAND*, called *curng_binomial*. This module implements the BINV algorithm proposed by *Voratas Kachitvichyanukul* and *Bruce W. Schmeiser* [12]. Algorithm BINV executes with speed proportional to $n \cdot p$ and has been improved by exploiting properties listed in the paper [12]. Also, it has got the best results assuming a normal probability approximation when $n \cdot p > 10$.

In depth, the library implements an *inline device function* which executes binomial randomization (BINV) when $n \cdot p \leq 10$ and normal randomization (CuRAND) otherwise. Our implementation generates binomial random numbers while running the kernel; thus, they are not generated previously.

The implementation of the phase is directly translated from the pseudocode of the DCBA. Also, it has got the best parallelism exploiting until now, comparing to other phases of the algorithm.

### 5.4 Implementation of Execution (Phase 4)

Phase 4 is implemented as directly shown in the DCBA pseudocode using our general CUDA design. In this case, we go to another level of parallelism for threads, that now works with each rule. As before, threads iterate the rules by

tiles, and adding the corresponding RHS (if it has a number of applications $N_r > 0$). Finally, since this operation is divergent (from rules to add objects), we use atomic operations again to update the configuration of the system.

## 6 Performance Results

In order to test the performance of our simulators, we constructed a random generator of PDP systems (designated *pdps-rand*). These randomly created PDP systems have no biological meaning. The purpose is to stress the simulator in order to analyze the implemented designs with different topologies. *pdps-rand* is parametrized in such a way that it can create PDP systems of a desired size.

We benchmark our *pdp-gpu-sim* and *pdp-omp-sim* (for 1, 2 and 4 cores) by first analyzing the scalability when increasing the size of the system in several ways. We then profile the simulators, showing the percentage of time taken by each phase separately. 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 DDR3 RAM and two NVIDIA Tesla C1060 graphics cards (240 cores at 1.30 GHZ , 4 GBytes of memory). GPU cores are typically slower than CPU cores.

Figure 3 shows the scalability of the simulator when the number of different objects appearing in the LHS (cooperation degree (see Definition 2)) increases. We can assume that, the greater the cooperation degree, the greater the number of competing blocks generated by *pdps-rand*. The figure shows the simulation time (in milliseconds) for one computation step running 50 simulations of PDP systems with 10 environments, 50000 rule blocks and 5000 different objects. The randomly generated PDP systems are sorted by the mean LHS length, showing that *pdp-gpu-sim* works better for lengths smaller than 3. The speedup achieved by *pdp-gpu-sim* is 6.6x and 2.3x for lengths of 1 and 2 against *pdp-omp-sim* with one core, and 4.5x and 1.9x against *pdp-omp-sim* with 4 cores, respectively.

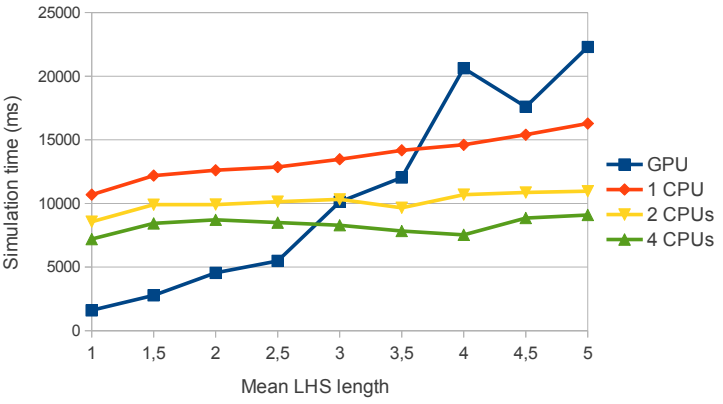


**Fig. 3.** Scalability when increasing the mean LHS length of rules

The second test analyses the performance when increasing the parallelism level of the CUDA threads within thread blocks, that is, the number of rule blocks. The speedup achieved by *pdp-gpu-sim* versus *pdp-omp-sim* is showed in Figure 4. The number of simulations is fixed to 50, and the environments to 20 (hence, a total of 1000 thread blocks). The number of objects is proportionally increased together with the number of rule blocks, in such a way that the ratio for number of rule blocks and number of objects is always 2. The mean LHS length is 1.5 (this is typical for many real ecosystem models, as seen in the literature). The speedup gets stable to around 7x on the number of rule blocks for the GPU versus CPU. For the multicore versions with 2 and 4 CPUs, the speedups are maintained to 4.3x and 3x, respectively. In our experiments, this number is also achieved when running with 1000000 rule blocks.

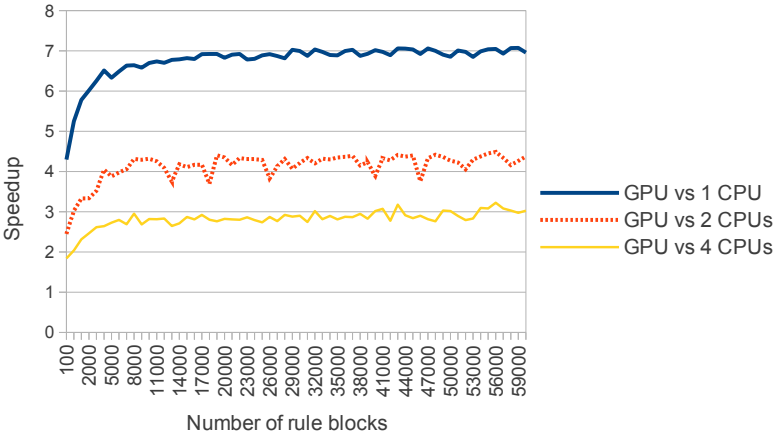


**Fig. 4.** Scalability of the simulators when increasing the number of rule blocks

The third test is for the second parallelism level in CUDA, concerning thread blocks. It is directly related with the number of environments and simulations. The result is shown in Figure 5. In this experiment, the number of rule blocks is fixed to 10000, the number of objects to 7024 and the mean LHS length is 2. The number of environments is fixed to 1 when increasing the simulations, and vice versa. As it can be seen, for low values, the speedup is demoted below 1. These values come from the fact of insufficient number of thread blocks to fulfill the GPU resources. Another trend shown is that as the number of simulations increases, the advantage of parallelizing by simulations increases. The same effect is observed for environments. This trend is stabilized to 3.5x for high values. However the parallelism over simulations is better carried out by the GPU, giving lower speedups for environments.

As stated in [16], parallelizing by simulations yields the largest speedups on multicore platforms. Therefore, we finalize the first benchmark by comparing these results with the GPU. Rule blocks are fixed to 50000, environments to

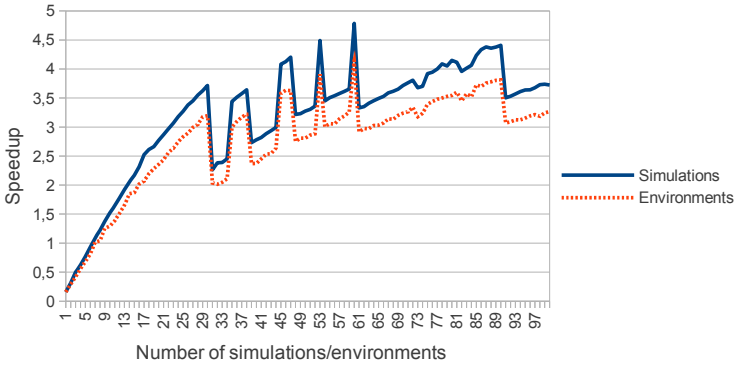


**Fig. 5.** Scalability – increasing the number of simulations and environments

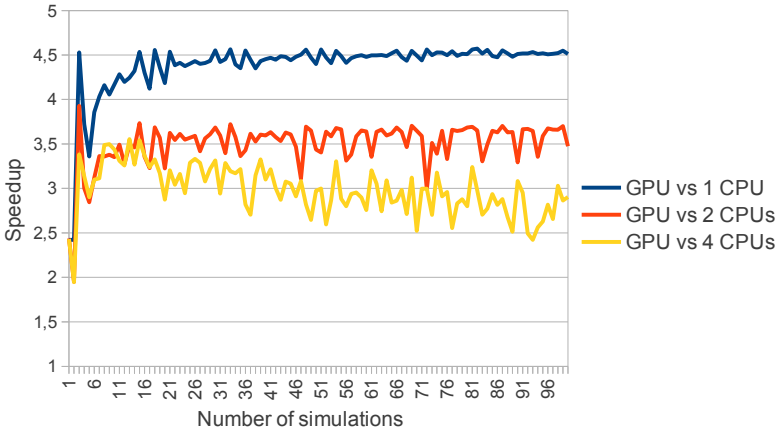


**Fig. 6.** Scalability of the simulators when increasing the number of simulations

20, objects to 5000 and mean LHS length to 1.5. As shown in Figure 6, the GPU achieves better runtime than the multicore implementations. The speedup is maintained to 4.5x using one core, 3.5x for 2 cores, and 2.7x for 4 cores.

The results of the second benchmark are shown in Table 1. This profile has been calculated running the simulator with 10000 rule blocks, 20 environments, 50 simulations, 5000 objects and two different mean LHS lengths, 1.5 (test A) and 3 (test B). Phase 1 is the most complex part in the simulation (taking more than 50% of the runtime on the CPU). In test A, the GPU implementation offers for phase 1 up to 14x of speedup. Therefore, the percentage of the execution time is decreased to 30%.

Following with Test A, Phase 2 takes only the 12% of the execution time on CPU. However, the GPU can only accelerate this phase by 2x. Therefore, this phase becomes the most expensive when executing the simulator on the GPU (47%). Our novel implementation, *ph2-simorder-dyncomp*, is close (time-wise) to the sequential implementation. Indeed, as mentioned above, this phase

**Table 1.** Profiling the simulators for GPU and 1 core CPU

| | Test A (mean LHS length 1.5) | | | Test B (mean LHS length 3) | | |
|---|---|---|---|---|---|---|
| | % CPU | % GPU | Speedup | % CPU | % GPU | Speedup |
| Phase 1 | 53.7% | 30.1% | 14.23x | 55.3% | 12% | 8.52x |
| Phase 2 | 12.6% | 47% | 2.13x | 18.4% | 82.8% | 0.4x |
| Phase 3 | 22.6% | 13.7% | 13.2x | 14% | 2.2% | 11.72x |
| Phase 4 | 11.1% | 9.2% | 9.7x | 12.3% | 3% | 7.43x |

is the most challenging to parallelize. Special efforts have to be considered here. On the other side, Phases 3 and 4 are relatively lightweight, and are successfully accelerated (up to 13x and 9.7x, respectively). Hence, our library random binomial generation based on CuRAND is well suited for Phase 3.

Finally, as shown in Figure 3, the performance of *pdp-gpu-sim* decreases as the mean LHS length is increased. For Test B, the overall speedup decreases from 7.9x (Test A) to 1.8x (Test B). The percentage of time consumed by Phase 2 is dramatically increased for the GPU, taking up to 83%. Thus, the competition degree of rule blocks is a limiting factor in performance, which fully correlates with the achieved results.

## 7 Conclusions and Future Work

In this paper we have presented the first GPU-based version of a simulator for PDP systems with CUDA. We improved the memory utilization of both our GPU-based version and our previous OpenMP-based version that we benchmarked against. We benchmarked a set of randomly generated PDP systems (without biological meaning), achieving speedups of up to 7x for large sizes on NVIDIA Tesla C1060 GPU over our multi-core version.

We used a general CUDA design for the GPU part of our simulator: environments and simulations are distributed through thread blocks, and rule blocks among threads. Phases 1, 3 and 4 were efficiently executed on the GPU, however Phase 2 was poorly accelerated, since it is inherently sequential.

For future work, the simulator is going to be reconnected with the P-Lingua framework. The pLinguaCore library is able to parse P-Lingua files, simulate P systems computations, and translate P-Lingua files to other file formats. In this respect, a new file format is going to be designed to become the input of our simulator. That way, the same P-Lingua files can contain the input data for the Java simulator inside pLinguaCore and the CUDA simulator. Moreover, we would like to validate our simulator with real ecosystems models. This would also help us to adopt model-oriented heuristics to improve the CUDA design. Finally, it would be useful to design a communication protocol and a Graphics User Interface in order to connect the simulation pipeline with the end-user.

This simulator represents our first attempt on simulating PDP systems using the GPU. Hence, improvements can be done to our new CUDA designs (i.e. iterating from objects to blocks (gather strategy)), optimizing the code, and/or

making more efforts on Phase 2. Future work will also take advantage of the new GPU architectures, such as the NVIDIA Kepler, and AMD APU, which deliver better performance than the Tesla C1060 we used for our experiments here.

**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 TIN2009-13192 of the "Ministerio de Educación y Ciencia" of Spain, both co-financed by FEDER funds. NVIDIA's support and donations to our HPC-Lab at NTNU is also gratefully acknowledged.

## References

1. Aqrawi, A.A., Elster, A.C.: Bandwidth Reduction through Multithreaded Compression of Seismic Images. In: IEEE International Symposium on Parallel & Distributed Processing (IPDPSW 2011), pp. 1730–1739 (2011)
2. Cabarle, F., Adorna, H., Martínez-del-Amor, M.A., Pérez-Jiménez, M.J.: Improving GPU Simulations of Spiking Neural P Systems. Romanian Journal of Information Science and Technology 15(1), 5–20 (2012)
3. Cardona, M., Colomer, M.A., Margalida, A., Palau, A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Sanuy, D.: A computational modeling for real ecosystems based on P systems. Natural Computing 10(1), 39–53 (2011)
4. Cardona, M., Colomer, M.A., Margalida, A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Sanuy, D.: A P System Based Model of an Ecosystem of Some Scavenger Birds. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) WMC 2009. LNCS, vol. 5957, pp. 182–195. Springer, Heidelberg (2010)
5. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P systems with Active Membranes on CUDA. Briefings in Bioinformatics 11(3), 313–322 (2010)
6. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulating a P system based efficient solution to SAT by using GPUs. Journal of Logic and Algebraic Programming 79(6), 317–325 (2010)
7. Cheruku, S., Păun, A., Romero-Campero, F.J., Pérez-Jiménez, M.J., Ibarra, O.H.: Simulating FAS-induced apoptosis by using P systems. Progress in Natural Science 17(4), 424–431 (2007)
8. Colomer, M.A., Lavín, S., Marco, I., Margalida, A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Sanuy, D., Serrano, E., Valencia-Cabrera, L.: Modeling Population Growth of Pyrenean Chamois (Rupicapra p. pyrenaica) by Using P-Systems. In: Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., Salomaa, A. (eds.) CMC 2010. LNCS, vol. 6501, pp. 144–159. Springer, Heidelberg (2010)
9. Colomer, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos, A.: Comparing simulation algorithms for multienvironment probabilistic P system over a standard virtual ecosystem. Natural Computing, doi:10.1007/s11047-011-9289-2
10. Elster, A.C.: High-Performance Computing: Past, Present, and Future. In: Fagerholm, J., Haataja, J., Järvinen, J., Lyly, M., Råback, P., Savolainen, V. (eds.) PARA 2002. LNCS, vol. 2367, pp. 433–444. Springer, Heidelberg (2002)

11. García-Quismondo, M., Gutiérrez-Escudero, R., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos-Núñez, A.: An Overview of P-Lingua 2.0. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) WMC 2009. LNCS, vol. 5957, pp. 264–288. Springer, Heidelberg (2010)
12. Kachitvichyanukul, V., Schmeiser, B.W.: Binomial random variate generation. Communications of the ACM 31(2), 216–222 (1988)
13. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands on Approach, MA, USA (2010)
14. Krog, Ø.E., Elster, A.C.: Fast GPU-Based Fluid Simulations Using SPH. In: Jónasson, K. (ed.) PARA 2010, Part II. LNCS, vol. 7134, pp. 98–109. Springer, Heidelberg (2012)
15. Harris, M.: Mapping computational concepts to GPUs. In: ACM SIGGRAPH 2005 Courses, NY, USA (2005)
16. Martínez-del-Amor, M.A., Karlin, I., Jensen, R.E., Pérez-Jiménez, M.J., Elster, A.C.: Parallel Simulation of Probabilistic P Systems on Multicore Platforms. In: Proceedings of the Tenth Brainstorming Week on Membrane Computing, vol. II, pp. 17–26 (2012)
17. Martínez-del-Amor, M.A., Pérez-Hurtado, I., García-Quismondo, M., Macías-Ramos, L.F., Valencia-Cabrera, L., Romero-Jiménez, A., Graciani, C., Riscos-Núñez, A., Colomer, M.A., Pérez-Jiménez, M.J.: DCBA: Simulating Population Dynamics P Systems with Proportional Object Distribution. In: Proceedings of the Tenth Brainstorming Week on Membrane Computing, vol. II, pp. 27–56 (2012)
18. Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos-Núñez, A., Sancho-Caparrini, F.: A simulation algorithm for multienvironment probabilistic P systems: A formal verification. International Journal of Foundations of Computer Science 22(1), 107–118 (2011)
19. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000), TUCS Report No 208
20. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
21. Spampinato, D.G., Elster, A.C.: Linear optimization on modern GPUs. In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009), pp. 1–8 (2009)
22. Terrazas, G., Krasnogor, N., Gheorghe, M., Bernardini, F., Diggle, S., Cámara, M.: An Environment Aware P-System Model of Quorum Sensing. In: Cooper, S.B., Löwe, B., Torenvliet, L. (eds.) CiE 2005. LNCS, vol. 3526, pp. 479–485. Springer, Heidelberg (2005)
23. NVIDIA CUDA programming guide 4.0 (2012), `http://www.nvidia.com/cuda`
24. The P-Lingua web page, `http://www.p-lingua.org`