Analysis of P systems simulation on CUDA

Ginés D. Guerrero, Jose M. Cecilia, José M. García [1]
Miguel A. Martínez–del–Amor, Ignacio Pérez–Hurtado, Mario J. Pérez–Jiménez [2]

Abstract—
GPUs (Graphics Processing Unit) have been consolidated as a massively data-parallel coprocessor to develop many general purpose computations, and enable developers to utilize several levels of parallelism to obtain better performance of their applications. The massively parallel nature of certain computations leads to use GPUs as an underlying architecture, becoming a good alternative to other parallel approaches. P systems or membrane systems are theoretical devices inspired in the way that living cells work, providing computational models and a high level computational modeling framework for biological systems. They are massively parallel distributed, and non-deterministic systems. In this paper, we evaluate the GPU as the underlying architecture to simulate the class of recognizer P systems with active membranes. We analyze the performance of three simulators implemented on CPU, CPU-GPU and GPU respectively. We compare them using a presented P system as a benchmark, showing that the GPU is better suited than the CPU to simulate those P systems due to its massively parallel nature.

Key words— **CUDA, P System, Membrane Computing.**

I. Introduction

TH e dominant trend in the design of microprocessor in recent years has been the increase of chip-level parallelism. This trend is noticed in the design of multicore CPUs (having 2-4 scalar cores), and everything points out to continue this trend increasing parallelism on towards "manycore" chips that provide higher level of parallelism. GPUs (Graphics Processing Units) have been at the leading edge of increasing chip-level parallelism for some time and they have become fundamentally manycore processors. Modern GPUs, such as NVIDIA Tesla C1060, contain up to 240 scalar processing units providing huge level of parallelism and being directly programmed using C language and CUDA extensions [14], [6].

Huge parallelism available on GPUs is used by the developers to enhance performance of their applications. Although many general purpose applications are difficult to be parallelized by developers due to its data-dependent and synchronizations features, other applications are massively parallel by its definition and they are well-suited to extract all the parallelism on GPUs.

Membrane computing (or cellular computing) is an emerging branch within Natural Computing. The main idea is to consider biochemical processes taking place inside living cells from a computational point of view, in a way that gives us a new non-deterministic model of computation by using cellular machines. The devices of this model are called *P systems* [15], and they consist of a cell-like membrane structure, where the compartments contain multisets of objects which evolve according to given rules in a synchronous non-deterministic maximally parallel manner.

There are different models of computation that have been investigated in this area. They are theoretically designed to solve diverse problems [2] [3]. In this work we deal with P systems capable of constructing an exponential workspace (expressed by the number of membranes and objects) in polynomial time. They are based on the model of P systems with active membranes and membrane division, that abstracts the way of obtaining new membranes through the process of *mitosis*. This model has been successfully used to design (uniform) solutions to well-known **NP**-complete problems, such as SAT [11] and *Subset Sum* [9] problems.

In this paper, we analyze GPUs as underlying architecture for a massively parallel simulation for the class of recognizer P systems with active membranes using CUDA. The simulation is divided in two main stages: *Selection stage* and *Execution stage*. We analyze three simulators which are executed on different platforms. The first simulator is fully executed on CPU, the second simulator is partly executed on the GPU (the selection stage is executed on the GPU and the execution stage is executed on the CPU), and finally the third simulator is totally implemented on the GPU. We test the simulator with a presented P system which exploits the intrinsic parallelism that P systems naturally have, and we demonstrate that GPU is better suited than CPU to simulate those P system as long as its intrinsic parallelism increase.

The rest of the paper is structured as follows. In Section 2 we describe the underlying architecture used for our experiments. Section 3, introduces the model of recognizer P systems with active membranes. In Section 4 we introduce the characteristics of the simulators. In Section 5 we describe the P system for testing our simulator. Finally, Section 6 shows the performance analysis of the simulators. The paper ends with some conclusions and ideas for future work in Section 7.

II. Underlying Architecture

In this section we briefly introduce the NVIDIA Tesla C1060 used in our experiments and the CUDA parallel programing model.

[1]Grupo de Arquitectura y Computación Paralela. Dpto. Ingeniería y Tecnología de Computadores. Universidad de Murcia. E-mail: {gines.guerrero, chema, jmgarcia}@ditec.um.es
[2]Research Group on Natural Computing. Department of Computer Science and Artificial Intelligence. University of Sevilla. E-mail: {mdelamor, perezh, marper}@us.es

A. Hardware Background

The Tesla C1060 [5] is based on scalable processor array which has 240 streaming-processor (SP) cores organised as 30 streaming multiprocessor (SMs) and 4GB of off-chip GDDR3 memory called *device memory or global memory*. The applications start at host side (CPU side) which communicates with device side (GPU side) through PCI-Express x16 bus (PCI-Express delivers up to 4 GB/sec of peak bandwidth per direction, and up to 8 GB/s concurrent bandwidth). The SM is the processing unit and it is unified graphics and computing multiprocessor. Every SM contains, among others, eight SPs arithmetic cores, a set of 16384 32-bit registers, a 16-Kbyte read/write on-chip shared memory that has very low access latency, and access to the global memory. The arithmetic units are capable to execute three instructions per clock cycle, and they are fully pipelined, running at 1,296 GHz, yielding 933 GFLOPS (240 SP * 3 instructions *1,296GHZ) of peak theoretical for the GPU.

A SM is a hardware device specifically designed with multithreaded capabilities. It 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 Single-Instruction Multiple-Thread (SIMT) fashion [5]. The SMs create, manage, schedule and execute threads in groups of 32 threads, that are called *Warp*. Each SM can handle up to 32 Warps (1024 threads in total). 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.

The execution flow begins with a set of Warps ready to be selected. The instruction unit selects one of them, which is ready for issue and executing instructions. The SM maps all the threads in an active Warp to the SP cores, and each thread executes independently with its own instructions and register state. Some threads of the active Warp can be inactived due to branching or predication, and it is a critical point in the optimisation process. The maximum performance is achieved when all the threads in an active Warp takes the same path. If the threads of a Warp diverge, the Warp serially executes each branch path taken, disabling threads that are not on that path, and when all the paths complete, threads reconverge to the original execution path [5].

B. Software Background

Parallel computing programs on GPUs are programed using the C and C++ programing language along with CUDA extensions (Compute Unified Device Architecture)[14]. In the CUDA parallel programing [13][6], an application consits of a sequential code (*host* code) that may execute parallel programs known as *kernels* on a parallel *device*. The host program executes on the CPU and the kernels execute on the GPU.

A kernel is a SPMD (Single Program, Multiple Data) computation executed by large number of threads running in parallel. The programmer organizes these threads into a grid of *thread blocks*. A thread block in CUDA is a set of threads that execute the same program (kernel) and cooperate to obtain a result through barrier synchronization and a per-block shared memory space, private to that block.

The programmer declares the number of threads block per grid and also the number of threads per thread block. The blocks in a grid are declared in one or two dimensions, and all of them have their own and unique identifier. Similarly, threads in a block can be declared in one, two or three dimensions, having their own and unique identifier too. Besides, the maximum number of threads in a block is 512. Using a combination of *thread id* and *block id*, threads can access to different data addresses and also to select the program code that they run.

Thread blocks in CUDA programing model are seen as virtual multiprocessors, since they have a fixed allocation of per-block shared memory and each thread in a block has a fixed register footprint [12]. The communication between thread blocks is performed through global memory and the synchronization among them is only obtained whenever the kernel ends.

III. Recognizer P systems Background

A. Membrane Computing

Membrane Computing is a vivid research area initiated in 1998 by Gh. Păun [7]. The main idea was to abstract the structure and functioning of a cell in order to extract computing models [8]. Many of them have been proved to be computational completeness (they are equivalent in power to Turing machines), and others are being used in the field of computational Systems Biology as modeling tools for biological systems.

The devices of the models considered in this framework are called *P systems*, and they are based on the fundamental concept of biological membranes: distributed (compartmentalized) parallel non-deterministic computing devices that process multisets of abstract objects by means of various types of rules.

The P system consists of a set of *syntactic* components: a *membrane structure* (it is formed by a rooted tree of membranes arranged hierarchically inside a root membrane called *skin*, delimiting *regions*), *multiset of objects* (corresponding to chemical substances present in the compartments (membranes) of a cell), and *rules* (corresponding to chemical reactions that can take place inside the cell).

A computation of a P system is a (finite or infinite) sequence of instantaneous steps called *configurations*, assuming a global clock that synchronize the execution. The computation starts always with a *initial configuration* of the system, where the input

data is encoded. The *transition* from one configuration to the next one is performed by applying rules to the objects placed inside the regions. Whenever it is not possible to apply more rules to the existing objects and membranes of a given configuration, the computation halts (it reaches a *halting configuration*). The result of a computation of the system is encoded by the multiset associated with a specific output membrane (or the environment) in a halting configuration. Non-determinism is presented in a P system when there are more than one possible transition from one configuration, resulting in a tree of computations with several of possible paths.

Finally, recognizer P systems were introduced in [10], and constitute the natural framework to study the efficient solvability of decision problems, which require either an affirmative (*yes*) or negative (*no*) answer. In this sense, we consider recognizer P systems as P systems with external output such that the *yes* or *no* answer is sent to the environment in every halting configuration.

B. P systems with active membranes, membrane division and polarization

Polynomial time solutions to **NP**-complete problems in membrane computing are achieved by trading time for space. It is inspired by the ability of cells to produce an exponential number of new membranes in polynomial time. There are many ways a living cell can produce new membranes: *mitosis* (cell division), *autopoiesis* (membrane creation), *gemmation*, etc. Following these inspirations a number of different models of P systems has arisen [8].

In this paper we focus on the model of P systems with active membranes, objectary membrane division and polarization. It is one of the most studied models in Membrane Computing, and one of the first models presented by Gh. Păun [8]. P systems with active membranes are formed by a membrane structure, where a label and a polarization is associated to each membrane. The membrane structure is a rooted tree, and the root is called *skin*. In this model, every *objectary membrane* (the leaves of the tree) is able to divide itself by reproducing its content into a new membrane.

Here we provide a short recall of its features (see [8] for details). The model of P system with active membranes is a construct of the form $\Pi = (O, H, \mu, \omega_1, \ldots, \omega_m, R)$, where $m \geq 1$ is the initial degree of the system; O is the alphabet of *objects*, H is a finite set of *labels* for membranes; μ is a membrane structure, consisting of m membranes injectively labelled with objects of H, $\omega_1, \ldots, \omega_m$ are strings over O, describing the *multisets of objects* placed in the m regions of μ; and R is a finite set of *rules*, where each rule is of one of the following forms:

(a) $[a \to v]_h^\alpha$ where $h \in H$, $\alpha \in \{+, -, 0\}$ (electrical charges), $a \in O$ and v is a string over O describing a multiset of objects associated with membranes and depending on the label and the charge of the membranes (*evolution rules*).

(b) $a\,[\,]_h^\alpha \to [b]_h^\beta$ where $h \in H$, $\alpha, \beta \in \{+, -, 0\}$, $a, b \in O$ (*send-in communication rules*). An object is introduced in the membrane, possibly modified, and the initial charge α is changed to β.

(c) $[a]_h^\alpha \to [\,]_h^\beta b$ where $h \in H$, $\alpha, \beta \in \{+, -, 0\}$, $a, b \in O$ (*send-out communication rules*). An object is sent out of the membrane, possibly modified, and the initial charge α is changed to β.

(d) $[a]_h^\alpha \to b$ where $h \in H$, $\alpha \in \{+, -, 0\}$, $a, b \in O$ (*dissolution rules*). A membrane with a specific charge is dissolved in reaction with a (possibly modified) object.

(e) $[a]_h^\alpha \to [b]_h^\beta\,[c]_h^\gamma$ where $h \in H$,$\alpha, \beta, \gamma \in \{+, -, 0\}$, $a, b, c \in O$ (*division rules*). A membrane is divided into two membranes. The objects inside the membrane are replicated, except for a, that may be modified in each membrane.

Rules are applied according to the following principles:

- All the objects which are not involved in any of the operations to be applied remain unchanged.
- Rules associated with label h are used for all membranes with this label, no matter whether the membrane is an initial one or whether it was generated by division during the computation.
- Rules from (a) to (e) are used as usual in the framework of membrane computing, i.e. in a maximal parallel way. In one step, each object in a membrane can only be used by at most one rule (non-deterministically chosen), but any object which can evolve by a rule must do it (with the restrictions indicated below).
- Rules (b) to (e) cannot be applied simultaneously in a membrane in one computation step.
- An object a in a membrane labelled with h and with charge α can trigger a division, yielding two membranes with label h, one of them having charge β and the other one having charge γ. Note that all the contents present before the division, except for object a, can be the subject of rules in parallel with the division. In this case we consider that in a single step two processes take place: "first" the contents are affected by the rules applied to them, and "after that" the results are replicated into the two new membranes.
- If a membrane is dissolved, its content (multiset and interior membranes) becomes part of the immediately external one. The skin is never dissolved neither divided.

Note that P systems with active membranes can be seen as devices with two levels of parallelism: among membranes where every membrane works independently (except for when there are communication across them), and among objects inside a membrane where the rules are applied to the existing multiset of objects in a maximal parallel way.

IV. Simulating P systems with active membranes

In our designs, we have based on the simulator for P systems with active membranes presented in PLinguaCore by I. Pérez–Hurtado et al [4], considering the requirements explained in section III-B. In this design, the simulation is divided into two stages: *selection stage* and *execution stage*. The selection stage consists of the search for the rules to be executed in each membrane, and the execution stage consists of the execution of these rules.

The input data for the selection stage consists of the description of the membranes with their multisets (strings over the working alphabet O, labels associated with the membrane in H, etc...) and the set of rules R to be selected. The output data of this stage is the set of selected rules per membrane which will be executed on the execution stage.

The execution stage applies the rules previously selected on the selection stage, and the membranes can vary including new objects, dissolving membranes, dividing membranes, etc obtaining a new configuration of the simulated P system. This new configuration would be the input data for the next step of the selection stage. Therefore, it is an iterative process until a halting configuration is reached.

Depending on the underlying architecture where each stage is developed (CPU, CPU and GPU or only GPU), we have developed three simulators.

A. Sequential simulator

This simulator is developed in C++ language, and it is based on the JAVA simulator presented in PLinguaCore [4]. It is adapted to have the same constraints and data structures than the CUDA simulators explained in the next subsections to make a fair comparison among them. This sequential version of the simulator obtains up to 120x of speed up compared with the simulator presented in [4].

B. Massively parallel simulator with selection stage on the GPU

This simulator develops the selection stage on the GPU [1] and the execution stage on the CPU. The selection stage is implemented as a CUDA *kernel*, identifying each membrane as a thread block where each thread represents an object of the alphabet O. Each thread block runs in parallel looking for the set of rules that has to execute, and each individual thread is responsible for identifying if there are some rules associated with the object that it represents.

The execution stage (developed on the CPU) receives the selected rules from device memory (GPU memory) through PCI-Express bus to execute them, obtaining a new P system configuration. If the obtained configuration is not a halting configuration, the selection stage is called again receiving the new P system configuration from main memory to device memory through PCI-Express bus.

This data movement through PCI-Express is too expensive in terms of performance and it affects to the overall performance of the simulator, as we expose in section VI.

C. Massively parallel simulator with both stages on the GPU

In this simulator we develop both stages of the simulation on the GPU to avoid data transfers through PCI-Express bus, and also to increase the performance of the execution stage. We use different CUDA *kernels* to implement this simulator.

The selection kernel is the same kernel than previously presented, but it also includes the execution of the evolution rules. It is due to two main reasons: the evolution rules do not implies communication among membranes, and they are executed in a maximal manner.

However, the rest of the rules only can be executed one per membrane, and they entail communication among them. Therefore, we design the execution of those rules as different CUDA kernels, one kernel per each kind of rule (send-in communication, send-out communication, dissolution and division), giving a result of the *execution stage*. We use different kernels for the execution stage because, otherwise, we should implement a bigger kernel with branches to identify each kind of rule to be applied, and this model decreases the performance of our application.

This simulator reduces data transfers through PCI-Express bus since both stages are developed on the GPU.

V. Test P systems for Performance Analysis

In this section, we design a test P system that extracts parallelism among objects and membranes to study the performance of the simulators. This P system is based on evolution rules for extracting parallelism from the objects of a membrane, and a division rule to create new membranes in every step, with no communication (without send-out/send-in rules) and no dissolution. The rooted membrane tree has two levels: the skin (with label 1) and the objectary membranes (with label 2).

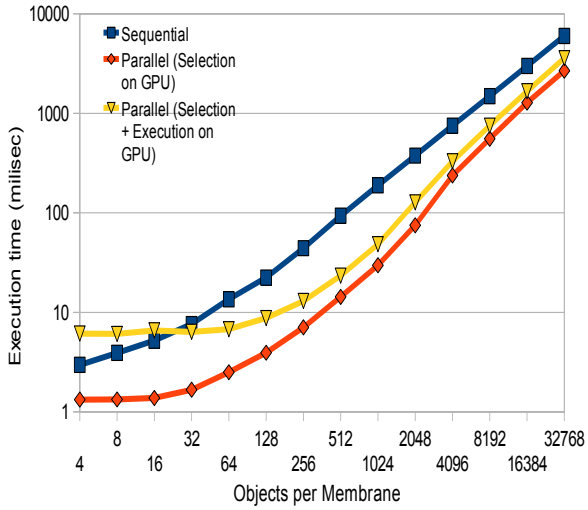
We define the following rules:

(a) Evolution rules: $[o\{i\} \rightarrow o\{i\}]_2^0, 0 <= i < n$
(b) Division rule: $[d]_2^0 \rightarrow [d]_2^0\,[d]_2^0$

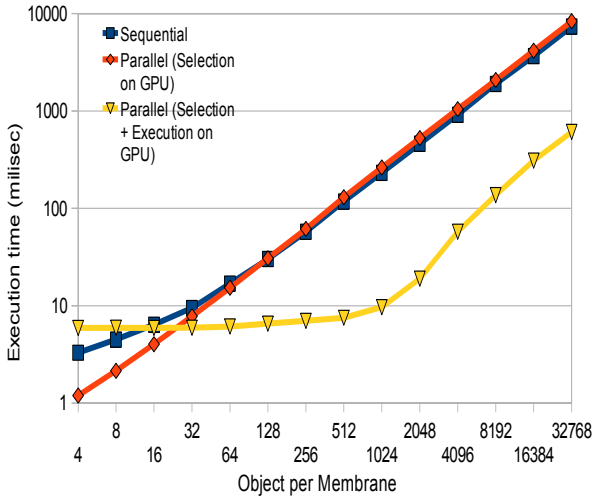
This P system allow us to take control of the number of objects in the system by modifying the n parameter. Furthermore, the number of rules changes along with the number of objects, and the number of membranes in every step is defined by 2^s, where s is the step number. Lastly, the number of evolution rules selected and executed per membrane in every step is invariable, since one object evolves always to the same.

VI. Experiments

This section presents the experiments of the simulations, making a comparison between the three simulators previously presented. For our tests, we use a



(a) Selection time for the three simulators



(b) Execution time for the three simulators

Fig. 1. P system simulation performance for the three simulators

benchmark based on the P system explained in the section V. It is covering both ways of parallelism that P systems naturally have by its definition. Firstly, it tests the parallelism between membranes, increasing the number of membranes until reaching a configuration with 16384 membranes, and it also tests the parallelism between objects increasing the number of objects within each membrane exponentially until 32768 objects per membrane.

We use CUDA version 2.1 and Tesla C1060 GPU in our experiments. GPU experiments we performed on a computer with an Intel core2 Quad Q9550 system running at 2.83GHz with 4GB of main memory. The performance of the CPU simulator used as comparison was measured with single-thread code executing on the same CPU of the CUDA simulator. The CPU simulator was compiled with the `-O3` option.

Figure 1 shows the performance of the selection stage and execution stage for all simulators presented in this work in a log scale.

On one hand, figure 1(a) shows the performance for the selection stage. As previously explained, the simulator fully executed on the GPU (V2 simulator) executes the evolution rules in the selection stage. Therefore, its performance decreases compared with the simulator with only selection on the GPU (V1 simulator). Nevertheless, V1 simulator and V2 simulator improve the performance of the sequential simulator in the selection stage, obtaining up to 7x of speed up.

On the other hand, figure 1(b) shows the performance for the execution stage. In this case, the V2 simulator obtains better performance than the others two simulators (developed both of them on the CPU) when they are dealing with more than 32 objects per membrane. 32 objects per membrane implies 32 threads per block which is the Warp size on Tesla C1060. As long as the GPU resources are better used, the V2 simulator performance increases obtaining up to 24x of speed up compared to the other two simulators.

Number of Objects	Selection on GPU	Selection + Execution on GPU	speed-up factor
4	3.33	0,41	8.12
16	5.49	0.41	13.39
64	13.61	0.41	33.19
256	42.39	0.45	94.2
1024	151.1	0.58	260.51
4096	569,87	1.09	522.81
16384	2235,26	2.96	755.15
32768	4453,17	5.37	829.26

TABLE I

Transfer time through PCI-Express bus in milliseconds on the Tesla C1060 for our Simulators. We show the speed-up factor for transfer time depending on the number of objects per membrane. The number of membranes is 16384 in all cases.

However, both simulators implemented on the GPU have an extra overhead which is the data transfers through PCI-Express bus. Table I shows the consumed time by the PCI-Express communications in both simulators. The V2 simulator greatly reduce the time consumed on those transactions by the V1 simulator, obtaining an overall best performance of the simulation as it is showed in figure 2.

VII. Conclusions and Future Work

In this paper we have analyzed three simulators for the class of recognizer P systems with active membranes. The first simulator is fully implemented on the CPU, the second simulator is implemented using CPU and GPU, and finally the third simulator is fully implemented on the GPU. Our experimental results demonstrate that a fully implementation on the GPU obtains a peak performance when simulating membrane systems due to the double parallel nature

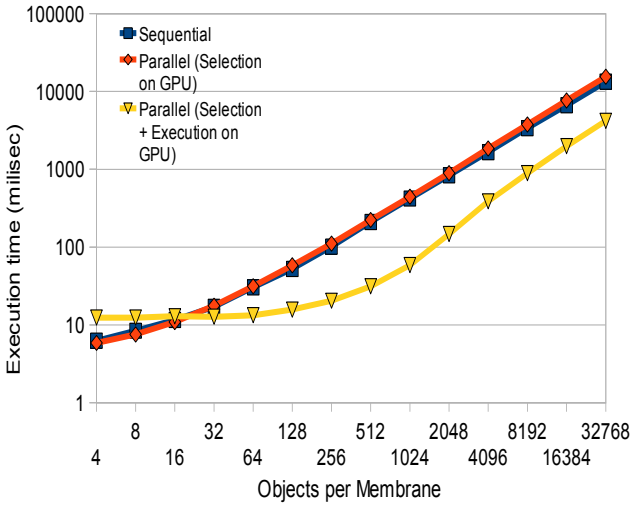


Fig. 2. Total simulation time of the three simulators, including the cost of transferring data through PCI-Express bus

that these systems present by their definition. The first level of parallelism is presented by the objects inside the membranes which fits with the parallelism among threads exposed on GPUs, and the second one is presented between membranes which we represent with the thread blocks on CUDA programing model.

P systems are very interesting tools to deal with **NP**-complete problems, by taking as inspiration the operation of living cells and cell reproduction (creating an exponential number of new membranes in polynomial time). Furthermore, P systems are being also used recently in the field of computational systems biology as a complement to other classical approaches, i.e. to model the behaviour of some ecosystems [2] or certain processes inside cells like apoptosis [3]. Hence, we consider that obtaining efficient simulations of P systems is really interesting for scientific research.

It is also important to remark that this simulator is limited by the available resources on the GPU as well as the CPU (RAM, Device Memory, CPU, GPU). In the following versions, we will reduce the memory requirements to better utilize the resources of the GPU. For this task we can use spare matrix in our simulator's design.

Although the massively parallel environment that provides the GPUs is good enough for the simulator, we need to go beyond. The newest cluster of GPUs provides a higher massively parallel environment, so we will attempt to scale to those systems to obtain better performance in our simulated codes and also more memory space for our simulations.

In forthcoming versions, we will study the possibility to simulate other kind of P systems, such as probabilistic or stochastic P system models, which are useful to attack other kind of problems within the framework of computational systems biology.

Acknowledgements

The first three authors acknowledge the support of the project from the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, and also by the Spanish MEC and European Commission FEDER under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03. The last three authors acknowledge the support of the project TIN2006–13425 of the Ministerio de Educación y Ciencia of Spain, cofinanced by FEDER funds, and the support of the "Proyecto de Excelencia con Investigador de Reconocida Valía" of the Junta de Andalucía under grant P08–TIC04200.

References

[1] M.A. Martínez–del–Amor, I. Pérez–Hurtado, M.J. Pérez–Jiménez, Jose M. Cecilia, Ginés D. Guerrero, José M. García. Simulation of Recognizer P Systems by using Manycore GPUs. Proceedings of 7th Brainstorming Week on Membrane Computing, Vol II, Fenix Editora, 2009. In press.

[2] M. Cardona, M. Angels Colomer, M.J. Pérez-Jiménez, D. Sanuy, A. Margalida. Modeling Ecosystems Using P Systems: The Bearded Vulture,a Case Study. In Proceedings of Workshop on Membrane Computing, Edinburgh, UK (2008), pp. 137–156.

[3] S. Cheruku, A. Paun, F.J. Romero–Campero, M.J. Pérez–Jiménez, O.H. Ibarra. Simulating FAS–induced apoptosis by using P systems. Progress in Natural Science, 17, 4 (2007), 424–431

[4] M. García–Quismondo, R. Gutiérrez–Escudero, I. Pérez–Hurtado, M.J. Pérez–Jiménez. P–Lingua 2.0: added features and first applications. Proceedings of the 7th Brainstorming Week on Membrane Computing, Vol I, Fenix Editora, 2009. In press.

[5] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 28, 2 (2008), 39–55.

[6] J. Nickolls, I. Buck, M. Garland, K. Skadron. Scalable parallel programming with CUDA. Queue, 6, 2 (2008), 40–53.

[7] G. Păun. Computing with membranes. Journal of Computer and System Sciences, 61, 1 (2000), pp. 108–143, and Turku Center for Computer Science-TUCS Report No 208.

[8] G. Păun: Membrane Computing, An introduction. Springer-Verlag, Berlín (2002).

[9] M.J. Pérez–Jiménez, A. Riscos–Núñez. Solving the Subset–Sum problem by active membranes. New Generation Computing, 23 (2005), 367–384.

[10] M.J. Pérez–Jiménez, F.J. Romero–Campero. An efficient family of P systems for packing items into bins. Journal of Universal Computer Science, 10, 5 (2004), 650–670.

[11] M.J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrinini. A polynomial complexity class in P systems using membrane division. Journal of Automata, Languages and Combinatorics, 11, 4 (2006), 423–434.

[12] N. Satish, M. Harris, M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. To Appear in Proceedings of the 23rd IEEE 4, May 2009.

[13] NVIDIA CUDA Programming Guide 2.0, (2008): `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`

[14] NVIDIA CUDA. World Wide Web electronic publication: `www.nvidia.com/cuda`

[15] The P Systems Webpage. `http://ppage.psystems.eu`