2009 International Workshop on High Performance Computational Systems Biology

Simulation of P Systems with Active Membranes on CUDA

Jose M. Cecilia, Ginés D. Guerrero,
José M. García
Grupo de Arquitectura y Computación Paralela
Dpto. Ingeniería y Tecnología de Computadores
Universidad de Murcia
Campus de Espinardo, 30100 Murcia, Spain
Email: {chema, gines.guerrero, jmgarcia}@ditec.um.es

Miguel A. Martínez–del–Amor, Ignacio Pérez–Hurtado,
Mario J. Pérez–Jiménez
Research Group on Natural Computing
Dpt. of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
Email: {mdelamor, perezh, marper}@us.es

***Abstract*—P systems or membrane systems provide a high level computational modeling framework that combines the structural and dynamic aspects of biological systems in a relevant and understandable way. P systems are massively parallel distributed, and non-deterministic systems. In this paper, we describe the implementation of a simulator for the class of recognizer P systems with active membranes by using the GPU (Graphics Processing Unit). We compare the high-performance parallel simulator for the GPU to the simulator developed on a single CPU (Central Processing Unit), and we show that the GPU is better suited than the CPU to simulate P systems due to its highly parallel nature.**

I. Introduction

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* [38], and they consist of a cell-like membrane structure, where in the compartments are multisets of objects which evolve according to given rules in a synchronous non-deterministic maximally parallel manner.

There are different computing models that have been investigated in this area: transition P system, P system with active membranes, probabilistic P system, stochastic P system, etc. All of these models are theoretically designed to solve diverse problems.

Most researches in membrane computing [6] concentrate on the computational power and efficiency of the devices involved, but lately they have been focused on modeling biological phenomena within the framework of computational systems biology being complementary and an alternative to more classical approaches (i.e. ODEs, Petri Nets, etc) [29].

Moreover, P systems have been successfully used as a computational modeling tool for diverse biological processes [29]. For instance, assuming that this paradigm is based in the structure and functioning of living cells, several models and simulators have been developed for apoptotic processes [5], communication among bacteria [30], etc. Furthermore, P systems have been also successfully applied as a tool for macroscopic level processes, as the computational modeling of ecosystems [3], [4].

Up to now, there have been no *in vivo* nor *in vitro* implementations of P systems, so computation and analysis of these devices is currently performed by simulators. Therefore, P systems simulators are tools that help the researchers to extract results from a model. These simulators have to be as efficient as possible when handling large problem sizes. This is one of the main problems with current simulators for P systems.

Software applications for membrane computing normally implement sequential (or parallel with few threads) algorithms simulation [7] adapted to common CPU architectures. These kinds of algorithms do not get performance when the problem size increases. In this sense, the simulation of P systems capable of constructing an exponential workspace (expressed by the number of membranes and objects) in linear time is especially critical.

One such widely used model is P systems with active membranes (that is membrane division and polarization), these abstract the biological process of *mitosis* to obtain new membranes. This model has been successfully used to design (uniform) solutions to well-known **NP**-complete problems, such as SAT [27] and *Subset Sum* [25] problems. We deal with this model because of the difficulty and low performance it presents when simulate in conventional computers. Our aim is to analyze the efficiency of a simulator based in a massively parallel architecture by using this P system model. This will help to develop efficient simulators for the study of biological processes with P systems in future.

The massively parallel nature of a P system computation leads us to look for a massively-parallel technology where the simulator can run efficiently. The newest generation of graphics processor units (GPUs) are massively parallel processors which can support several thousand of concurrent threads. Many general purpose applications have been designed on these platforms due to its high performance [31], [34]. Current NVIDIA GPUs, for example, contain up to 240 scalar processing elements per chip [17], and they are programmed using C and CUDA [20], [35].

In this paper, we present a massively parallel simulator for the class of recognizer P systems with active membranes using CUDA. The simulator receives as input a P system which is defined by using the P-Lingua [7] programing language. The simulator is divided in two main stages: *Selection stage* and *Execution stage*. Both phases are implemented on the GPU, so the simulator is executing in parallel.

We test the simulator with a P system which exploits the intrinsic parallelism that P systems naturally have and demonstrate that a GPU is better suited than a CPU to simulate those P system as long as the problem size increase.

The rest of the paper is structured as follows. In Section 2 we describe the model of recognizer P systems with active membranes. Section 3 introduces the Compute Unified Device Architecture (CUDA), and some concepts of programming on GPUs are specified. In Section 4 we explain the design of the simulator. In Section 5 we implement the P system for testing our simulator, and we explain the tool used for its definition. Finally, Section 6 shows some results and compare them to sequential version of the simulator. The paper ends with some conclusions and ideas for future work in Section 7.

II. Recognizer Membrane Systems

A. Membrane Computing background

Membrane Computing is a vivid research area initiated in 1998 by Gh. Păun [23]. In October 2003, the Institute of Scientific Information (ISI) designed Membrane Computing as a Fast Emerging Research Front in Computer Science [39]. Also, the foundational paper [23] was nominated by the ISI as the Fast Breaking Paper of February 2003.

The main idea was to abstract the structure and functioning of a cell in order to extract computing models. Many of them have been proved to be computational complete (they are equivalent in power to Turing machines), and other higher-order structures as the organization of cells into tissues, organs and neural networks have been also considered and used to abstract other computing models (see [24] for details).

A large variety of cell-like computing models, called *P systems*, were considered in this framework based on the fundamental concept of biological membranes; the respective models are distributed (compartmentalized) parallel computing devices, processing multisets of abstract objects by means of various types of rules. Parallelism, communication, non-determinism, synchronization, dynamic architecture of the model, etc, are aspects of the theory, with biological, mathematical and computer science sources of inspiration [13].

A 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 of a cell), and *evolution 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 transitions between *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 (then, the configuration is called a *halting computation*). 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 of the computation.

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. That is, more than one rule can be selected in a given configuration, but only one of them can be executed (which leads to different configurations). Thus, a non-deterministic P system has many possible computations, where some of them are halting computations, and others are infinite.

Furthermore, a P system is *confluent* when all the computations (that is, every path of the computation tree), with the same initial configuration, sends out the same answer. Therefore, we have only to look at one computation in order to know the answer when using confluent P systems, since the rest of computations will send the same output.

Finally, P systems can be used for addressing the efficient resolution of decision problems. These kinds of problems require either a *yes* or *no* answer. In this sense, we consider recognizer P systems [26] as P systems with external output (the results of halting computations are encoded in the environment) such that:

1) the working alphabet of objects contains two distinguished elements *yes* and *no*.
2) all computations halt; and
3) if C is a computation of the system, then either object *yes* or object *no* (but not both) must have been released into the environment, and only in the last step of the computation.

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. This 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 [7].

In this paper we focus on the model of *P systems with active membranes*. It is one of the most studied models in Membrane Computing, very well-known by the P system community, and one of the first models presented by Gh. Păun [24]. P systems with active membranes is formed by a membrane structure, where a label and a polarization is associated to each membrane. In this model, every elementary membrane is able to divide itself by reproducing its content into a new membrane.

Here we provide a short recall of its features (see [24] 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 (a rooted tree), consisting of m membranes injectively labeled with elements 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 elements 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 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 labeled 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.

In the literature, recognizer P systems are associated in a natural way with P systems with *input*. The data representing an instance of the decision problem has to be provided to the P system to compute the appropriate answer. This is done by codifying each instance as a multiset placed in an *input membrane*. The output of the computation, yes or no, is sent to the environment [26].

Note that P systems with active membranes can be seen as devices with two levels of parallelism: among membranes (every membrane works independently, with the exception of when there are communication across them) and among objects inside a membrane (the rules are applied to the existing multiset of objects in a maximal parallel way). This characteristic is used in order to quickly solve **NP**-complete problems, as mentioned so far, by trading time for space. The main idea is to encode each possible instance in a distinguished membrane, and select the membranes that encode a solution to the problem in a parallel manner.

III. Parallel Computing on the GPU

Before discussing the design of our simulator for P systems with active membranes, we briefly introduce the NVIDIA’s GPU used in our tests and CUDA programing model. GPUs (Graphic Processing Units) were designed to accelerate graphics applications, using for this task programing interfaces such as OpenGL and DirectX. Due to its tremendous parallelism and arithmetic intensity, GPUs have became a massively parallel processor very attractive to develop scientific applications. NVIDIA consolidated this trend launching a line of GPUs optimized for general purpose computations called TESLA [17].

A. Hardware Background

Figure 1 shows the Tesla architecture. Particularly, the Tesla C1060 [17] is based on scalable processor array which has 240 streaming-processor (SP) cores organized

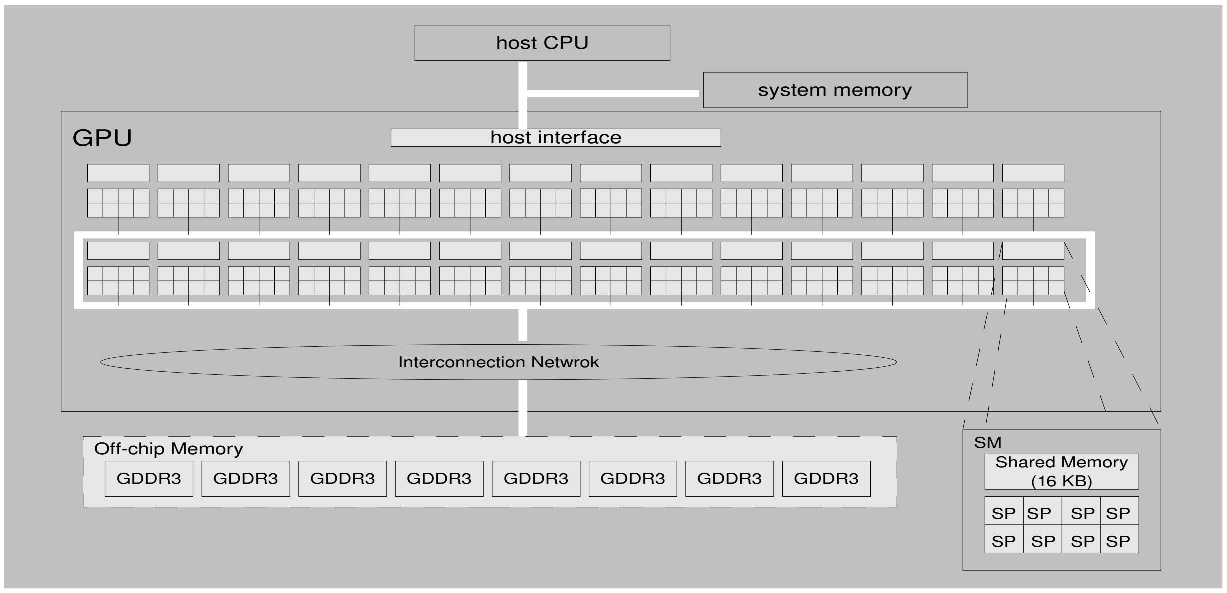


Figure 1. Tesla C1060 GPU with 240 SPs:Streaming Processors. Organized in 30 SMs: Streaming Multiprocessors

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 an unified graphics and computing multiprocessor. Every SM contains 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 video memory tied to the graphics card, whose latency is around 400-600 cycles). The SM also has two SFUs that execute more complex FP 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 933 GFLOPS (240 SP * 3 instructions *1,296GHZ) of peak theoretical for the GPU. Table I shows the major hardware constraints on Tesla C1060.

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. SMs execute threads in Single-Instruction Multiple-Thread (SIMT) fashion [17]. SMs create, manage, schedule and execute threads in groups of 32 threads. This set of 32 threads is 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, which is ready for issue and executing instructions, selects one of them. 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 inactivated due to branching or predication, and this is a critical point in the optimization 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 in that path, and when all the paths complete, threads reconverge to the original execution path [17].

Table I
MAJOR HARDWARE AND SOFTWARE CONSTRAINTS ON TESLA C1060

Hardware Parameters	Limitation
Streaming Multiprocessor (SM)	30
Streaming Processor/SM (SP)	8
32-bit Registers/SM	16384
Shared Memory/SM	16KB
Threads/SM	1024
Threads/Block	512
Threads/Warp	32
Device Memory	4GB

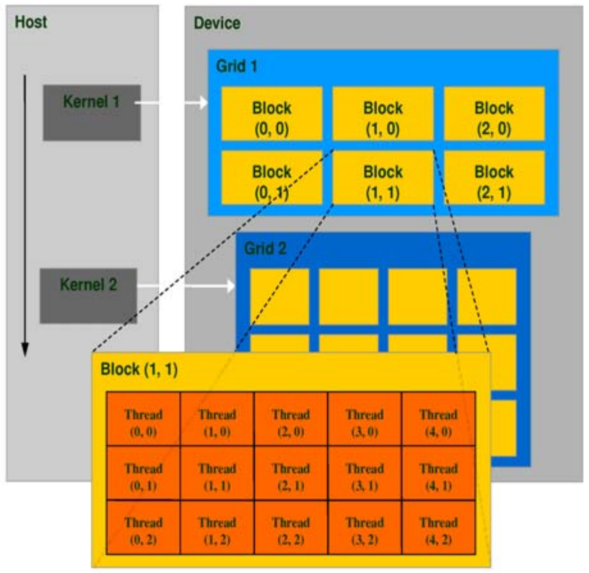


Figure 2. CUDA programming model

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)[37]. In CUDA parallel programing [20], [35], an application consists 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 (see figure 2).

A kernel is a SPMD (Single Program, Multiple Data) computation executed by large number of threads running in parallel. The programer 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 programer declares the number of threads block per grid and also the number of threads per thread block (see figure 2). 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 the 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 [34]. The communication between thread blocks is performed through global memory and the synchronization among them is only obtained whenever the kernel ends.

C. CUDA tools support

There are several tools that makes easier the task of programing in the CUDA development cycle. Firstly, the *nvcc* compiler handles all parts of the compilation flow, trying to hide the compilation details from developers and giving a wide range of compiler options. There are several compiler flags that are really useful in certain parts of the development process. For the debug purpose there is an emulation mode which is enabled with the compiler flag *-g* (this flag generates debuggable code). Furthermore, the compiler has other flags which are focused on optimizing the CUDA code. These flags are *-ptx -cubin*.

The Parallel Thread Execution (PTX) code is an assembly-like representation which is produced by the nvcc compiler, whenever a CUDA code is compiled with the `-ptx` flag enabled, and it is optimized by the CUDA runtime to get hardware-specific binaries for execution. Notice that PTX code is not the code which executes on the GPU, but it gives an approximated idea of the execution.

Besides, the nvcc compiler has the `-cubin` flag which produces a .cubin file. This file contains information about occupancy of each SM. It also shows the number of registers per thread, the amount of shared memory used by a thread block, whether the kernel is using local memory or not, and finally, the binary code of the application. This information can be used to obtain the maximum occupancy of the SM depending on the resources used by each thread block.

Other software tools have been created to support the CUDA programers and ease the CUDA development cycle, such as *CudaVisualProfiler* or *decuda* [37]. The former is a quite useful tool to profile your CUDA code. The latter is a disassembler for the Nvidia CUDA binary (.cubin) format and it helps to identify bottlenecks showing the internal instructions generated for the G8x and G9x architectures.

IV. Simulator for P Systems with Active Membranes

In this section we briefly describe the simulator of recognizer P systems with active membranes that we have designed.

Figure 3 shows the basic design of the simulator. We identify each membrane as a thread block where each thread represents an element of the alphabet O. Each thread block runs in parallel looking for the set of rules that has to select for its membrane, and each individual thread is responsible for identifying if there are some rules associated with the object that it represents (each thread select the rules that need to be executed by using the represented object).

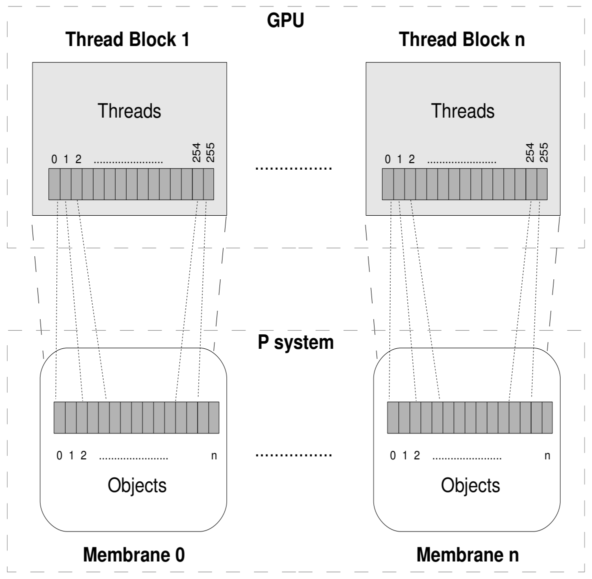


Figure 3. Design of the simulator: mapping membranes and objects with thread blocks and threads

The simulator is executed into two stages: *selection stage* and *execution stage*. This is based on the simulator for P systems with active membranes developed in PLinguaCore by I. Pérez–Hurtado et al [8]. The selection stage consists of the search for the rules to be executed in each membrane. At this stage, non-determinism is presented when it is possible to select several rules, but only one of them can be executed. For example, two evolution rules that can be executed using the same object, a division rule and a send-in rule that can be selected in the same membrane at the same time, etc.

We assume that the input P system is confluent (see section II-A) in order to avoid non-determinism. So, instead of working with the entire tree of possible computations, the simulator selects and simulates only one path. Since all paths are guaranteed to give the same answer. The computation path can be selected by choosing one set of rules to be executed with the lowest cost. We measure the cost in number of membranes and synchronizations. These are the conditions that damage the simulation performance the most. In this context, we introduce the following priorities among rules in our simulator:

1) *Dissolution rules* decrease the number of membranes (highest priority);
2) *Evolution rules* do not need any communication among membranes (which avoids synchronization);
3) *Send-out rules* do need communication between the given membrane and its parent (adding one object to its parent);
4) *Send-in rules* do need communication between the given membrane and its parent (reserving one object from its parent and adding the object to itself);
5) *Division rules* increase the number of membranes (lowest priority).

Once the rules have been selected, the *execution stage* consists of the execution of these rules. Both stages have been parallelized on the GPU as several different kernels due to the need of global synchronization among each stage.

The input data for the *selection kernel* 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 kernel is the set of selected rules, and also the modified description of the membranes where the evolution rules have been applied. We decide to execute evolution rules in this kernel due to two main reasons: the evolution rules do not implies communication (and so synchronization) among membranes, and they are executed in a maximal manner. Moreover, this decision allow to use less global memory because it does not store the selected evolution rules for the execution stage.

Besides, the rest of the rules to be applied are executed in different 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 design the execution kernel in this way 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. As result of the execution kernel, the membranes can vary including new objects, dissolving membranes, dividing membranes, etc. Therefore, we modify the input data for the next execution of *selection kernel* with the newest structure of membranes. It is an iterative process until a halting configuration or a system response is reached.

Our simulator presents two restrictions, constrained by some peculiarities in CUDA programming model: it can handle only two levels of membrane hierarchy for simplicity (the skin and the rest of elementary membranes), which is enough for solving many **NP**-complete problems; moreover, the number of objects in the alphabet must be divisible by a number smaller than 512 (the maximum number of threads per thread block), in order to distribute the objects among the threads equally.

V. Test P System Designed 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 simulator. This P system is based on evolution rules and one 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 only two levels: the skin (with label 1) and the elementary 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.

We encoded the test P system by using P-Lingua [7], which is a programing language useful for defining P system models with active membranes. In this work, we use the new version, P-Lingua 2.0 [8], to generate a binary file that our simulator can use as input. This binary file is easy-to-read by a C++ program, containing all the information of the P system (Alphabet, Labels, Rules, ...).

VI. Experiments

This section presents the results of the simulator making a comparison between a sequential simulator developed in C++ language and the simulator developed in CUDA. For our tests, we use two benchmarks based on the P system explained in section V. These benchmarks cover both ways of parallelism that P systems naturally have by its definition. The first benchmark tests the parallelism between membranes, increasing the number of membranes exponentially, and the second benchmark tests the parallelism between objects increasing the number of objects within each membrane exponentially.

We use CUDA version 2.1 in our experiments and Tesla C1060 GPU. 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 *gcc* and the `-O3` option, and the CUDA simulator was compiled and debugged by using the tools explained in section III-C. The performance for the CUDA simulator includes the cost of transferring input data from host CPU memory across the PCI-Express bus to the GPU's on board memory.

Figure 4 shows the performance between the sequential version of the simulation of P systems with active membranes and the CUDA version of this simulation in a log scale. Specifically, figure 4(a) shows the behaviour of both simulators executing the benchmark which increases the number of membranes exponentially (2^n), having a fixed number of objects per membrane (2560 objects).

On one hand, the CPU simulator increases its time from the beginning (with 4 membranes) until reaching the final configuration (with 32768 membranes) along with the problem size.

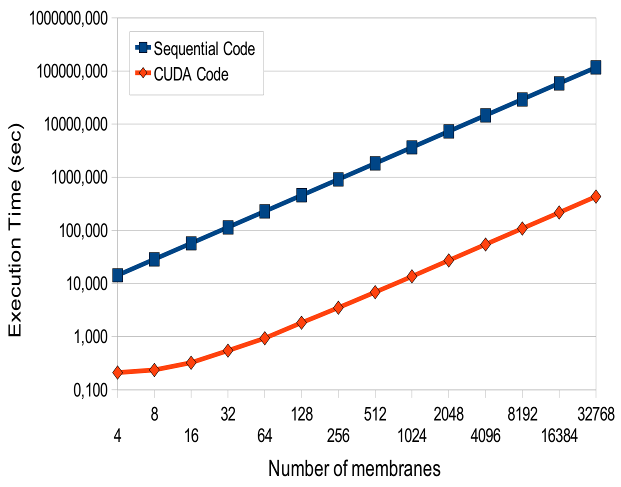
On the other hand, the CUDA simulator assigns 256 threads per block (each thread handles 10 elements per membrane). 256-thread blocks provide the overall best performance in our experiments. Figure 4(a) shows that the CUDA code increases its performance compared with the sequential code when the number of membranes increases, obtaining the overall best performance when the resources of the GPU are fully utilized (whenever the simulator has enough blocks and threads to fully utilized all the GPU cores). The difference of performance between both simulators is maintained from this point.

Figure 4(b) shows the behaviour of both simulators executing the benchmark which increases the number of objects per membrane. In this case, the number of membranes is fixed to 1024 which implies to have enough blocks to distribute the work among multiprocessors. Our simulations starts with only few objects per membrane, which implies just few threads per block in the CUDA code. In this case, figure 4(b) shows that the sequential code obtains better performance than the CUDA code until the simulations reach 32-elements per membrane. Less than 32-elements per membrane implies less than 32-threads per blocks in the CUDA code which avoid even fulfills a Warp, hence GPU resources are badly used.

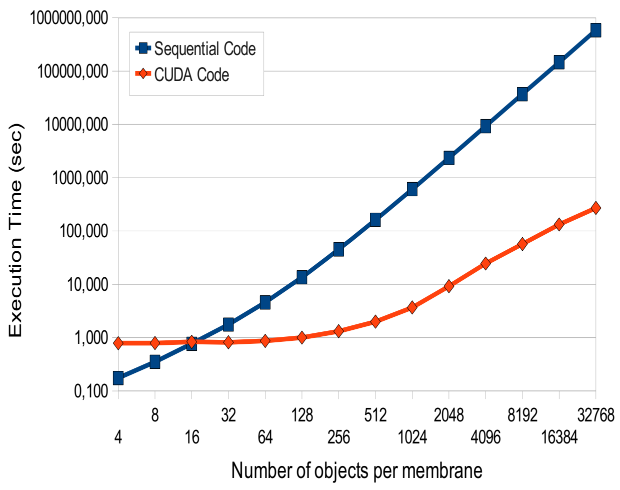
The sequential code increases its simulation time along with the number of objects. In the sequential code just one thread has to deal with all the objects in each membrane, one by one, hence the simulation time increases as much as the number of objects also increases. However, the simulation time is maintained by the CUDA code until reaching 256-objects configuration. The simulation time increases a little bit faster from this configuration because the following configurations have more objects per membranes than threads per block (uses 256-thread blocks). Therefore, objects in a membrane are equally distributed across all the threads in a block: 512-object per membrane implies 2 objects per thread, 1024-object per membrane implies 4 objects per thread, and so on. Otherwise it implies to have an overloaded thread which reduce the performance of our simulator, and leads us to conclude to have lightweight threads.

VII. Conclusions and Future Work

In this paper we have introduced the design of a simulator for the class of recognizer P systems with active membranes on the GPU. Our experimental results demonstrate that GPUs are good platforms to simulate membrane systems due to the double parallel nature that they present. 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.



(a) Varying the number of membranes



(b) Varying the number of objects per membrane

Figure 4. P system simulation performance for both sequential and CUDA simulator

Using the power and parallelism that provides GPUs to simulate P systems with active membranes is a new concept in the development of applications for membrane computing. Although GPUs are not a cellular machine, their features help the researches to accelerate their simulations allowing the consolidation of the cellular machines as an alternative to traditional machines.

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). Moreover, membrane computing has been used recently to model many biological systems and to complement other classical approaches, i.e. to simulate the behaviour of some ecosystems or certain processes inside cells like apoptosis. Hence, we consider that obtaining efficient simulations of P systems is really interesting for scientific research.

In forthcoming versions, we will adapt our simulator to simulate specific problems at maximum performance. We are also working to obtain fully simulation of P systems with active membranes, deleting the limitations showed in section IV. Furthermore, we need to include the possibility to simulate other kind of P systems in our simulator, such as probabilistic P system model or stochastic P system model, which are used to attack other kind of problems within the framework of computational systems biology.

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). They limit the size of the instances of **NP**-complete problems whose solutions can be successfully simulated. In the following versions, we will reduce the memory requirements in order to handle bigger instances of **NP**-complete problems.

Although the massively parallel environment that provides 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.

Acknowledgments

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. 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 TIC04200.

References

[1] A. Alhazov and M.J. Pérez–Jiménez, "Uniform solution of QSAT using polarizationless active membranes," in Machines, Computations, and Universality, J. Durand-Lose and M. Margenstern, Eds. Lecture Notes in Computer Science, 4664, 2007, pp. 122-133.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in SIGGRAPH '04, ACM Press, 2004, pp. 777-786.

[3] M. Cardona, M. Angels Colomer, M.J. Pérez–Jiménez, D. Sanuy, and 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.

[4] M. Cardona, M.A. Colomer, A. Margalida, I. Pérez–Hurtado, M.J. Pérez–Jiménez, and D. Sanuy, "P system based model of an ecosystem of the scavenger birds," in Proceedings of the 7th Brainstorming Week on Membrane Computing, vol. I, 2009, pp. 65-80.

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

[6] G. Ciobanu, M.J. Pérez–Jiménez, and G. Păun. Applications of membrane computing. Natural Computing Series, Springer, 2006.

[7] D. Díaz–Pernil, I. Pérez–Hurtado, M.J. Pérez–Jiménez, and A. Riscos–Núñez, "A P-Lingua programming environment for Membrane Computing," in Membrane Computing: 9th International Workshop (WMC08), Lecture Notes in Computer Science, 2009, 187-203.

[8] M. García–Quismondo, R. Gutiérrez–Escudero, M.A. Martínez–del–Amor, E. Orejuela, and I. Pérez–Hurtado, "P-Lingua 2.0: A software framework for cell-like P systems," International Journal of Computers, Communications and Control, vol. IV (3), 2009, 234-243.

[9] M. Garland, S.L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," IEEE Micro, vol. 28 (4), 2008, 13-27.

[10] N.K. Govindaraju and D. Manocha, "Cache–efficient numerical algorithms using graphics hardware," Parallel Comput., vol. 33 (10-11), 2007, 663-684.

[11] M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, and A. Riscos–Núñez, "Available membrane computing software," in Applications of Membrane Computing, G. Ciobau, Gh. Păun, M.J. Pérez–Jiménez, Eds. Natural Computing Series, 2006, pp. 411-436.

[12] M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, and A. Riscos–Núñez, "Towards a programming language in cellular computing," Electronic Notes in Theoretical Computer Science, vol. 123, 2005, 93-110.

[13] M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez, and F.J. Romero–Campero, "Computational efficiency of dissolution rules in membrane systems," International Journal of Computer Mathematics, vol. 83 (7), 2006, 593-611.

[14] M. Harris, S. Sengupta, and J.D. Owens, "Parallel prefix sum (Scan) with CUDA," GPU Gems 3, 2007.

[15] T.D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of GPUs and multicores," in Proceedings of the 22nd annual international conference on Supercomputing (ICS '08), ACM, 2008, pp. 15-25.

[16] M.D. Lam, E.E. Rothberg, and M.E. Wolf, "The cache performance and optimizations of blocked algorithms," in Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS-IV), ACM, 1991, pp. 63-74.

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

[18] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard, "Cg: a system for programming graphics hardware in a C–like language," SIGGRAPH '03, ACM, 2003, pp. 896-907.

[19] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," IPDPS, 2008, pp. 1-7.

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

[21] J. D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "Gpu computing," in Proceedings of the IEEE, vol. 96 (5), 2008, 879-899.

[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A.E. Lefohn, and T.J. Purcell, "A survey of general–purpose computation on graphics hardware," Computer Graphics Forum, vol. 26 (1), 2007, 80-113.

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

[24] G. Păun, Membrane Computing, an introduction. Springer-Verlag, Berlin, 2002.

[25] M.J. Pérez–Jiménez and A. Riscos–Núñez, "Solving the Subset–Sum problem by active membranes," New Generation Computing, vol. 23, 2005, 367-384.

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

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

[28] M.J. Pérez–Jiménez, A. Romero–Jiménez, and F. Sancho–Caparrini, "Complexity classes in models of cellular computing with membranes," Natural Computing, vol. 2 (3), 2003, 265-285.

[29] F.J. Romero-Campero, "P Systems, a Computational Modelling Framework for Systems Biology," Doctoral Thesis, University of Seville, Department of Computer Science and Artificial Intelligence, 2008.

[30] F.J. Romero-Campero and M.J. Pérez–Jiménez, "A model of the Quorum Sensing system in Vibrio Fischeri using P systems," Artificial Life, vol. 14 (1), 2008, 95-109.

[31] A. Ruiz, M. Ujaldon, J.A. Andrades, J. Becerra, K. Huang, T. Pan, and J.H. Saltz, "The GPU on biomedical image processing for color and phenotype analysis," BIBE, 2007, pp. 1124-1128.

[32] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. mei Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008, pp. 73-82.

[33] S. Ryoo, C.I. Rodrigues, S.S. Stone, J.A. Stratton, Sain-Zee Ueng, S.S. Baghsorkhi, and W.W. Hwu, “Program optimization carving for GPU computing,” J. Parallel Distrib. Comput., vol. 68 (10), 2008, 1389-1401.

[34] N. Satish, M. Harris, and M. Garland, “Designing Efficient Sorting Algorithms for Manycore GPUs,” in Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, May 2009, unpublished.

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

[36] GPGPU organization. World Wide Web electronic publication: www.gpgpu.org

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

[38] The P systems Webpage: http://ppage.psystems.eu

[39] Institute of Scientific Information, Philadelphia PA, USA: http://esi-topics.com/erf/october2003.html