# Simulating Active Membrane Systems Using GPUs

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

[1] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
{mdelamor,perezh,marper}@us.es

[2] Grupo de Arquitectura y Computación Paralela
Dpto. Ingeniería y Tecnología de Computadores
Universidad de Murcia
Campus de Espinardo, 30100 Murcia, Spain
{chema,gines.guerrero,jmgarcia}@ditec.um.es

**Summary.** Software development for cellular computing is growing up yielding new applications. In this paper, we describe a simulator for the class of recognizer P systems with active membranes, which exploits the massively parallel nature of P systems computations by using GPUs (Graphics Processing Units). The newest generation of GPUs provide a massively parallel framework to compute general purpose computations. We present GPUs as an alternative to obtain better performance in the simulation of P systems and we illustrate it by giving a solution to the N-Queens problem as an example.

## 1 Introduction

Membrane computing (or cellular computing) is an emerging branch within natural computing that was introduced by Gh. Păun [24]. 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 nondeterministic model of computation by using cellular machines.

Up to now, it has not been possible to have implementations neither *in vivo* nor *in vitro* of P systems, so handling and analysis of these devices are performed by simulators. Therefore, P systems simulators are tools that help the researchers to extract results from a model. Since the model was presented, many software applications have been produced [11]. These simulators have to be as much efficient as possible when handling large problem sizes. Thus, the massively parallel nature

of P systems computations points out to looking for a massively parallel technology where the simulator can run efficiently.

Parallel computation on clusters is the traditional environment to speed-up parallel applications. Particularly, many simulators of P systems have been designed for clusters of computers [4]. However, this computation is relatively expensive and it is available for organizations that have enough resources to buy and maintain those clusters. Nowadays, there are other cheaper solutions in the computer market that provides parallel environments. Among these solutions, the newest generation of graphics processor units (GPUs) are massively parallel processors which allow to develop a wide range of parallel applications. We also recall that other parallel computing platforms are being investigated, such as special hardware circuits [20][6].

GPUs can support several thousand of concurrent threads providing a massively parallel environment where parallel applications can obtain huge performance [14][17][29]. Current Nvidia's GPUs, for example, contain up to 240 scalar processing elements per chip [16], they are programmed using C and CUDA [32][21], and they have low cost compared with a cluster of computers.

In this paper we present a parallel simulator for the class of recognizer P systems with active membranes using CUDA. The simulator executes the P system which is defined by using the P-Lingua [5] programming language. The simulator is divided in two main stages: The *selection stage* and *execution stage*. At this point of development, the *selection stage* is executed on the GPU and the *execution stage* is executed on the CPU.

The rest of the paper is structured as follows. In Section 2 several definitions and concepts are given for a correct understanding of the paper. 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 a solution to the N-Queens problem using the simulator and P-Lingua. Finally, in Section 6 we show some results and compare them with the sequential version of the simulator. The paper ends with some conclusions and ideas for future work in Section 7.

## 2 Preliminaries

Polynomial time solutions to **NP**-complete problems in membrane computing are achieved by trading time for space. This is inspired by the capability 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, and many of them proved to be computational completeness [5].

In this paper we shall focus on the model of *P systems with active membranes*. It is one of the most studied models in Membrane Computing and one of the first

models presented by Gh. Păun [25]. 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 [25] 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; $\mu$ is a membrane structure (a rooted tree), consisting of $m$ membranes injectively labelled with elements of $H$, $\omega_1, \ldots, \omega_m$ are strings over $O$, describing the *multisets of objects* placed in the $m$ regions of $\mu$; 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 $\alpha$ is changed to $\beta$.
(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 $\alpha$ is changed to $\beta$.
(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 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 $\alpha$ can trigger a division, yielding two membranes with label $h$, one of them having charge $\beta$ and the other one having charge $\gamma$. 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 (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).

Recognizer P systems were introduced in [26], and constitute the natural framework to study the solvability of decision problems. The data representing an instance of the 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 in every halting configuration.

Furthermore, the act of simulating something generally entails representing certain key characteristics or behaviours of some physical, or abstract, system. However, an emulation tool duplicates the functions of one system by using a different system, so that the second system behaves like (and appears to be) the first system. With the current technology, we can not emulate the functionality of a cellular machine by using a conventional computer to solve **NP**-complete problems in polynomial time, but we can simulate these cellular machines, not necessarily in polynomial time, in order to aid researchers. However, depending on the underlying technology where the simulator is executed, the simulations can take too much time.

The technology used for this work is called CUDA (Compute Unified Device Architecture). CUDA is a co-designed hardware and software solution to make easier developing general-purpose applications on the Graphics Processor Unit (GPU) [34]. The GPUs, that are one of the main components of traditional computers, originally were specialized for math-intensive, highly parallel computation which is the nature of graphics applications. These characteristics of the GPU were very attractive to accelerate scientific applications which have massively parallel computations. However, the problem was the way to program general purpose applications on the GPU. This way involved to deal with GPUs designed for video games, so they have had to tune their applications using programming idioms tied to computer graphics, programing environment tightly constrained, etc [17] [14]. The CUDA extensions developed by Nvidia provides an easier environment to program general-purpose applications onto the GPU, because it is based on ANSI C, supported by several keywords and constructs. ANSI C is the standard published by the American National Standards Institute (ANSI) for the C programming language, which is one of the most used.

P systems devices are massively parallel, what fits into massively parallel nature of the GPUs with thousands of threads running in parallel. These threads are units of execution which execute the same code concurrently on different pieces of data.

## 3 Graphics Processing Unit

Driven by the video games market, programmable GPUs (Graphics Processing Units) have evolved into a highly parallel, multithreaded, manycore processor. They were designed to accelerate graphics applications, which transform three-dimensional data (coordinates of triangle vertices) into pixels that are displayed on a screen, using for this task programming interfaces such as OpenGL and DirectX. The massively parallel nature of graphics applications and its arithmetic intensity leads the researches to explore mapping more general non-graphics applications onto the GPU, creating a new programming field called GPGPU (General-Purpose on GPUs).

GPUs have become an inexpensive and readily available single-chip massively parallel system. However, GPGPU programmers had to deal with the limitations and difficulties of constrained graphics primitives to compute their non-graphics computations. The emergence of Compute Unified Device Architecture (CUDA) [34] programming model, proposed by Nvidia Corporation in 2007, has helped to develop highly-parallel applications onto the GPU easier than it was before. CUDA allows GPGPU programmers to develop their applications in a more familiar environment by using C/C++ programming language, with some extensions to manipulate special aspects of the GPU. Moreover, Nvidia consolidated this trend launching a line of GPUs optimized for general purpose computations called TESLA [16].

In this work we use a Tesla C1060 graphics processor unit (GPU) from Nvidia as hardware target for its study. This section introduces the Tesla C1060 computing architecture. In addition, it analyses the threading model of Tesla architectures, and also the most important issues in the CUDA programming environment.

### 3.1 Tesla C1060 base microarchitecture

The Tesla C1060 [16] is based on a scalable processor array which has 240 streaming-processor (SP) cores organised as 30 streaming multiprocessor (SM). The applications start at the host side (the CPU) which communicates with the device side (the GPU) through a PCI-Express x16 bus (see the top of figure 1).

The SM is the processing unit, and it is unified graphics and computing multiprocessor. Every SM contains eight SPs arithmetic cores, one double precision unit, 16-Kbyte read/write shared memory, a set of 16384 registers, and access to the off-chip memory (global/local memory). The access to shared memory is very cheap, however, the access to the off-chip memory has low performance because it is out of the chip, as it is shown on figure 1. In addition, table 1 shows all memories available on the GPU and also the cost to access them.

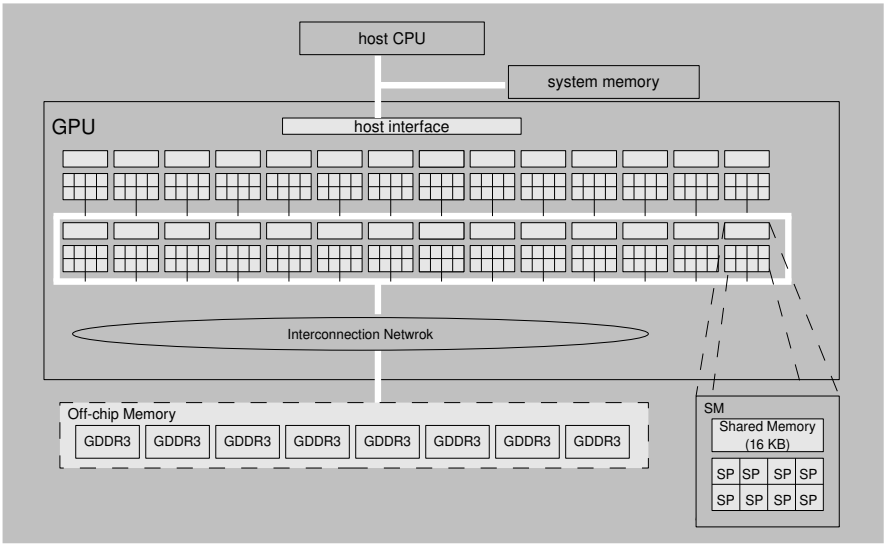


**Fig. 1.** Tesla C1060 GPU with 240 SPs: Streamming Processors, organised in 30 SMs: Streamming Multiprocessors

**Table 1.** Memory System on the Tesla C1060

| **Memory** | **Location** | **Size** | **Latency** | **Access** |
|---|---|---|---|---|
| Registers | On-Chip | 16384 32-bits Registers per SM | $\simeq 0$ cycles | R/W |
| Shared Memory | On-Chip | 16 KB per SM | $\simeq registers$ | R/W |
| Constant | On-Chip | 64 KB | $\simeq registers$ | R |
| Texture | On-Chip | Up to Global | $> 100$ cycles | R |
| Local | Off-Chip | 4 GB | 400-600 cycles | R/W |
| Global | Off-Chip | 4 GB | 400-600 cycles | R/W |

### 3.2 Parallel computing with CUDA

The GPU is seen as a cooprocessor that executes data-parallel *kernel* functions. The user creates a program encompassing CPU code (Host code) and GPU code (Kernel code). They are separated and compiled by *nvcc* (Nvidia's compiler for CUDA code) as shown in figure 2

Firstly, the host code is responsible for transfering data from the main memory (RAM or host memory) to the GPU memory (device memory), using CUDA instructions, such as *cudamemcpy*. Moreover, the host code has to state the number of threads executing the kernel function and the organization of them. Threads execute the kernel code, and they are organized into a three-level hierarchy as it is shown in figure 3. At the highest level, each kernel creates a single grid that consists of many thread blocks. Each thread block can contain up to 512 threads,

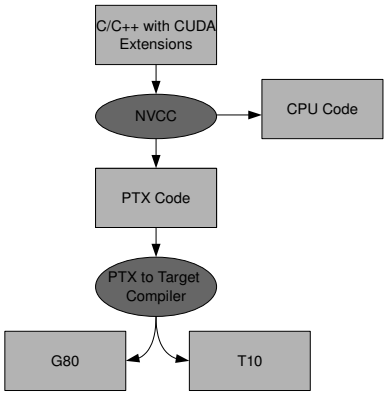


**Fig. 2.** Nvcc compilation process

which can share data through Shared Memory and can perform barrier synchronization by invoking the *–syncthreads* primitive [31]. Besides, thread blocks can not perform synchronization. The synchronization across blocks can only be obtained by terminating the kernel.

Furthermore, the host code calls the kernel function like a C function by passing parameters if it is needed, and also by specifying the number of threads per block and the number of blocks making up the grid. Each block within the grid has their own identifier [22]. This identifier can be one, two or three dimensions depending on how the programmer has declared the grid, accessed via *.x*, *.y*, and *.z* index fields. Each thread within the block have their own identifier which can be one, two or three dimensions as well. Combining thread and block identifiers, the threads can access to different data address, and also select the work that they have to do.

The kernel code is specified through the key word *__global__* and the syntax is: *__global__ kernelName <<< dimGrid, dimBlock >>> (...parameter list...)* where dimGrid and dimBlock are three-elements vectors that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively [21].

### 3.3 Threading model

A SM is a hardware device specifically designed with multithreaded capabilities. Each SM manages and executes up to 1024 threads in hardware with zero scheduling overhead. Each thread has its own thread execution state and can execute an independent code path. The SMs execute threads in a Single-Instruction Multiple-Thread (SIMT) fashion [16]. Basically, in the SIMT model all the threads execute the same instruction on different piece of data. The 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, see table 2). Individual

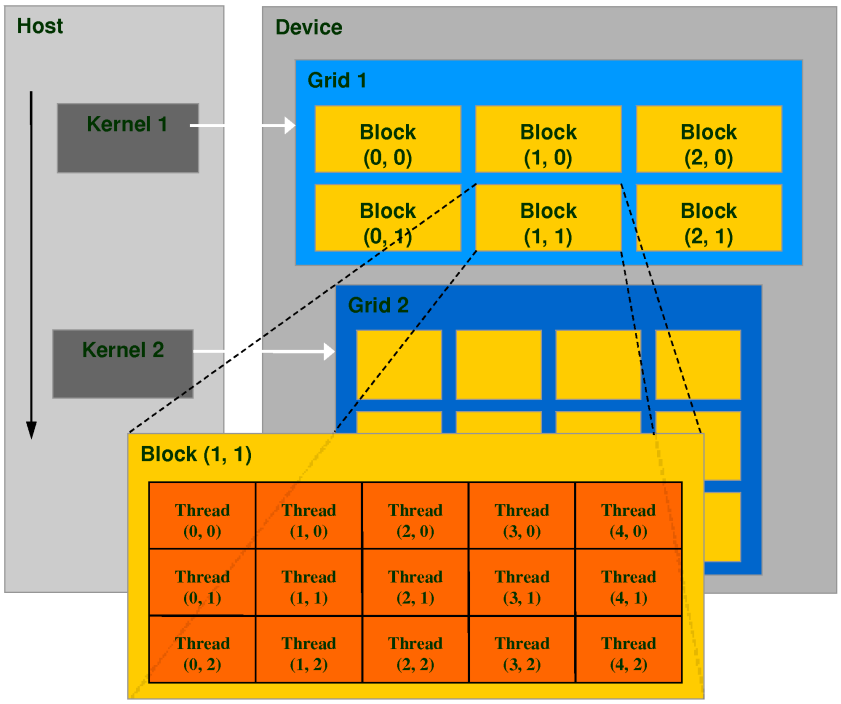


**Fig. 3.** Thread organization in CUDA programming model

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.

**Table 2.** Major Hardware and Software Limitations programing on CUDA

| Configuration Parameters | Limitation |
|---|---|
| Threads/SM | 1024 |
| Thread Blocks/SM | 8 |
| 32-bit Registers/SM | 16384 |
| Shared Memory/SM | 16KB |
| Threads/Block | 512 |
| Threads/Warp | 32 |
| Warps/SM | 32 |

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 per SP core, and each thread executes independently with its own instructions and register state. Some threads of the active Warp can be inactive due to branching or predication, and it is also another critical point in the optimisation process. The maximum performance is achieved when all the threads in an active Warp takes the same path (the same execution flow). 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, the threads reconverge to the original execution path.

## 4 Design of the Simulator for Recognizer P Systems

In this section we briefly describe the simulator of recognizer P systems with active membranes, elementary division and polarization. Firstly, we explain the previous work that we have done in order to prepare the development of the parallel simulator on the GPU. Then, we introduce the algorithm design in the CUDA programming language, and finally, we finish with our simulator's design.

### 4.1 Design of the baseline simulator

As previously mentioned, CUDA programming model is based on C/C++ language. Therefore, the first recommended step when developing applications in CUDA is to start from a baseline algorithm written in C++, where some parts can be susceptible to be parallelized on the GPU.

In this work, we have based on the simulator for P systems with active membranes developed in PLinguaCore by I. Pérez–Hurtado et al [5]. This sequential (or single-threaded) simulator is programmed in JAVA, so the first step was to translate the code to C++.

The simulator is executed into two main stages: *selection stage* and *execution stage*. The *selection stage* consists of the search for the rules to be executed in each membrane. Once the rules have been selected, 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. Only the *execution stage* changes the information of the configuration. It is the reason because *execution stage* needs synchronization when accessing to the membrane structure and the multisets. At this point of implementation, we have parallelized the *selection stage* on the GPU, and the *execution stage* is still executed on the CPU because of the synchronization problem.

We also have developed an adapted sequential simulator for the CPU (called *fast sequential* simulator), which has the same constraints as the CUDA simulator

explained in the next subsections to make a fair comparison among them. This simulator achieves much better performance than the original sequential simulator.

### 4.2 Algorithm design in CUDA

Whenever we design algorithms in the CUDA programming model, our main effort is dividing the required work into processing pieces, which have to be processed by $TB$ thread blocks of $T$ threads each. Using a thread block size of $T$=*256*, it is empirically determined to obtain the overall best performance on the Tesla C1060 [28]. Each thread block access to one different set of input data, and assigns a single or small constant number of input elements to each thread.

Each thread block can be considered independent to the other, and it is at this level at which internal communication (among threads) is cheap using explicit barriers to synchronize, and external communication (among blocks) becomes expensive, since global synchronization only can be achieved by the barrier implicit between successive kernel calls. The need of global synchronization in our designs requires successive kernel calls even to the same kernel.

### 4.3 Design of the parallel simulator

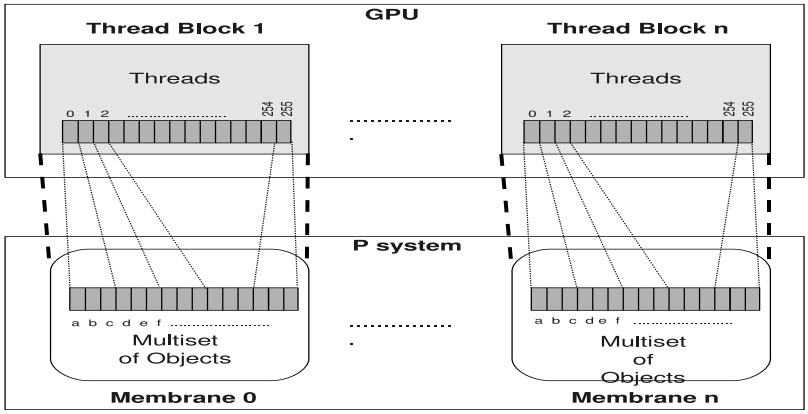


**Fig. 4.** Mapping membranes and objects with thread blocks and threads

In our design, we identify each membrane as a thread block where each thread represents at least an element of the alphabet $O$ (figure 4). 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 selecting the rules associated with the object that it represents (each thread selects the rules that need to be executed by using the represented object).

As result of the *execution stage*, the membranes can vary including news elements, dissolving membranes, dividing membranes, etc. Therefore, we have to

modify the input data for the *selection stage* with the newest structure of membranes, and then call the selection again. It is an iterative process until a halting configuration is reached.

Finally, our simulator presents some limitations, constrained by some peculiarities in the CUDA programming model. The main limitations are showed in table 3, and the following stand out among them: it can handle only two levels of membrane hierarchy for simplicity in synchronization (the skin and the rest of elementary membranes), which is enough for solving lots of **NP**-complete problems; and the number of objects in the alphabet must be divisible by a number smaller than 512 (the maximum thread block size), in order to distribute the objects among the threads equally.

**Table 3.** Main limitations in the parallel simulator

| **Parameter** | **Limitation** |
|---|---|
| Levels of membrane hierarchy | 2 |
| Maximum alphabet size | 65535 |
| Maximum label set size | 65535 |
| Maximum multiplicity of an object in an elementary membrane | 65535 |
| Alphabet size | Divisible by a number smaller than 512 |

## 5 A Case of Study: Implementing a Solution to the N-Queens problem

In this section, we briefly present a solution to the **N-Queens** problem, given by Miguel A. Gutiérrez–Naranjo et al [10], using our simulator.

### 5.1 A family of P systems for solving the N-Queens problem

The **N-Queens** problem can be expressed as a formula in conjunctive normal form, in such way that one truth assignment of the formula is considered as **N-Queens** solution. A family of recognizer P system for the SAT problem [27] can state whether exists a solution to the formula or not sending *yes* or *no* to the environment.

However, the *yes* ot *no* answer from the recognizer P system is not enough because it is also important to know the solutions. Besides, the system needs to give us the way to encode the state of the N-Queens problem.

The P system designed for solving the **N-Queens** problem is a modification of the P system for the SAT problem. It is an uniform family of deterministic recognizer P system which solves SAT as a decision problem (i.e., the P system

sends *yes* or *no* to the environment in the last computation step), but it also stores the truth assignments that makes true the formula encoded in the elementary membranes of the halting configuration.

### 5.2 Implementation

P-Lingua 1.0 [5] is a programming language useful for defining P system models with active membranes. We use P-Lingua to encode a solution to the **N-Queens** problem, and also to generate a file that our simulator can use as input. Figure 5 shows the P-Lingua process to generate the input for our simulator.

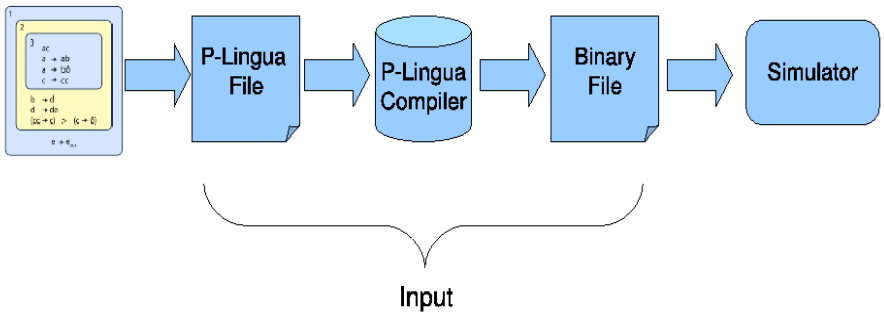


**Fig. 5.** Generation of the simulator's input

P-Lingua 2.0 [7] translates a model written in P-Lingua language into a binary file. A binary file is a file whose information is encoded in Bytes and bits (not understandable by humans like plain text), which is suitable for trying to compress the data. This binary file contains all the information of the P system (Alphabet, Labels, Rules, ...) which is executed by our simulator.

In our tests, we use the P system for solving the 3-Queens and 4-Queens problems. The former creates 512 membranes and up to 1883 different objects. The latter creates 65536 membranes and up to 8120 different objects, and now the simulator can handle it because we have decreased the memory requirement by the simulator in [18]. On one hand, the P system for 5-Queens needs to generate 33554432 membranes and 25574 objects, what leads in a memory space limitation (requires up to 1.5TB). On the other hand, note that 2-Queens is a system with only 4 membranes, what is not enough for exploiting the parallelism in P systems.

## 6 Performance Analysis

We now examine the experimental performance of our simulator. Our performance test are based on the solutions to 3-Queens and 4-Queens problems previously

explained in 5.2. They state an example of how a **NP**-complete problem can be solved by the simulator for the P systems with active membranes. We report the *selection stage* time which is executed on the GPU, and compare it with the *selection stage* for the fast sequential code. We do not include the cost of transferring input (and output) data from (and to) host CPU memory across the PCI-Express bus to the GPU's on board memory, which negatively affects to the overall simulation time. Selection is one building block of a larger-scale computation. Our aim is to get a full implementation of the simulator on the GPU. In such case, the transfers across PCI-Express bus will be close to zero.

We have used the Nvidia GPU Tesla C1060 which has 240 execution cores and 4GB of device memory, plugged in a computer server with a Intel Core2 Quad CPU and 8GB of RAM, using the 32bits ubuntu server as Operating System.

The *selection stage* on the GPU takes about *171 msec* for the 3-Queens. So it is 2.7 times faster than the *selection stage* on the CPU which takes *465 msec*. For the 4-Queens problem our simulator is 2 times faster than the fast sequential version, taking 315291 and 629849 msec in selection respectively.

Our experimental results demonstrate the results we expect to see: a massively parallel problem such as selection of the rules in a P-System with active membranes achieves faster running times on a massively parallel architecture such as GPU.

## 7 Conclusions and Future Work

In this paper, we have presented a simulator for the class of recognizer P systems with active membranes using CUDA. P system computations have a double parallel nature. The first level of parallelism is presented by the objects inside the membranes, and the second one is presented between membranes. Hence, we have simulated these P systems in a platform which provides those levels of parallelism. This platform is the GPU, with parallelism between thread blocks and threads. Besides, we have used a programming language called P-Lingua to encode P systems as input for our simulator. This tool helped us to use the P system for solving the N-Queens problem in order to test our simulator.

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

The first version of the simulator is presented for P systems with active membranes, elementary division and polarization, specifically, we have developed the *selection stage* of the simulator on the GPU. In forthcoming versions, we will include the execution version on the GPU. This issue allows a completely parallel execution on the GPU, avoiding CPU-GPU transfers in every step, which degrades system performance.

Moreover, we are working to obtain fully simulation of P systems with active membranes, deleting the limitations showed in table 3. Besides, we will include new funcionality in the simulator like not elementary division.

It is also important to point out that this simulator is limited by the resources available 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. Although developing general purpose programs on the GPU is easier than several years ago with tools such as CUDA, to extract the maximum performance of the GPU is still hard, so we need to make a deep analysis to obtain the maximum performance available for our simulator. For instance, in the following versions of the simulator we will reduce the memory requirements in order to simulate bigger instances of **NP**-complete problems and avoid idle threads, by deleting objects with zero multiplicity. For this task we can use spare matrix in our simulator's design.

The massively parallel environment that provides the GPUs is good enough for the simulator, however, 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.

Finally, we will study the adaptation of the design of P systems to the constraints of the GPU to make faster simulations. Furthermore, it would be interesting to avoid the brute force algorithms in P system computations, and start to design heuristics in the design of membrane solutions (i.e. avoiding membrane division as possible).

### Acknowledgement

The first 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. The last 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.

## References

1. A. Alhazov, M.J. Pérez–Jiménez. Uniform solution of QSAT using polarizationless active membranes. *Machines, Computations, and Universality.* Lecture Notes in Computer Science, **4664** (2007), 122–133.
2. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *SIGGRAPH '04, ACM Press*, (2004), 777–786.
3. G. Ciobanu, M.J. Pérez–Jiménez, G. Paun, (eds.) *Applications of membrane computing.* Natural Computing Series, Springer, (2006).

4. G. Ciobanu, G. Wenyuan. P systems running on a cluster of computers. *Lecture Notes in Computer Science*, **2993** (2004), 123–139.
5. D. Díaz–Pernil, I. Pérez–Hurtado, M.J. Pérez–Jiménez, A. Riscos–Núñez. A P-Lingua programming environment for Membrane Computing. *Lecture Notes in Computer Science*, **5391** (2009) , 187–203.
6. L. Fernández, V.J. Martínez, F. Arroyo, L.F. Mingo. A hardware circuit for selecting active rules in transition P systems. *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2005), pp. 415.
7. M. García–Quismondo, R. Gutiérrez–Escudero, M.A. Martínez–del–Amor, E. Orejuela, 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.
8. M. Garland, S.L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, **28**, 4 (2008), 13–27.
9. N.K. Govindaraju, D. Manocha. Cache–efficient numerical algorithms using graphics hardware. *Parallel Computing*, **33**, 10–11 (2007), 663–684.
10. M.A. Gutiérrez–Naranjo, M.A. Martínez–del–Amor, I. Pérez–Hurtado, M.J. Pérez–Jiménez. Solving the N–Queens Puzzle with P systems. *Proceedings of the 7th Brainstorming Week on Membrane Computing*, Vol. I (2009), pp. 199–210.
11. M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez. Available membrane computing software. *Applications of Membrane Computing*, Natural Computing Series, Springer–Verlag, 2006. Chapter 15 (2006), pp. 411–436.
12. M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez. Towards a programming language in cellular computing. *Electronic Notes in Theoretical Computer Science*, **123** (2005), 93–110.
13. M. Harris, S. Sengupta, J.D. Owens. Parallel prefix sum (Scan) with CUDA. *GPU Gems*, **3** (2007).
14. T.D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, M. Ujaldon. Biomedical image analysis on a cooperative cluster of GPUs and multicores. *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, ACM (2008), pp. 15–25.
15. M.D. Lam, E.E. Rothberg, M.E. Wolf. The cache performance and optimizations of blocked algorithms. *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ACM (1991), pp. 63–74.
16. E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro*, **28**, 2 (2008), 39–55.
17. W.R. Mark, R.S. Glanville, K. Akeley, M.J. Kilgard. Cg: a system for programming graphics hardware in a C–like language. *SIGGRAPH '03*, ACM (2003), pp. 896–907.
18. 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 (2009), pp. 45–58.
19. J. Michalakes, M. Vachharajani. GPU acceleration of numerical weather prediction. *IPDPS*. (2008), pp. 1–7.
20. V. Nguyen, D. Kearney, G. Gioiosa. An algorithm for non-deterministic object distribution in P systems and its implementation in hardware. *Lecture Notes in Computer Science*, **5391** (2009), 325–354.

21. J. Nickolls, I. Buck, M. Garland, K. Skadron. Scalable parallel programming with CUDA. *Queue*, **6**, 2 (2008), 40–53.
22. J. D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96, 5 (2008), pp. 879–899.
23. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A.E. Lefohn, T.J. Purcell. A survey of general–purpose computation on graphics hardware. *Computer Graphics Forum*, **26**, 1 (2007), 80–113.
24. G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and Turku Center for Computer Science-TUCS Report No 208.
25. G. Păun: *Membrane Computing, An introduction*. Springer-Verlag, Berlín (2002).
26. M.J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, **2**, 3 (2003), 265–285.
27. M.J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrini. A polynomial complexity class in P systems using membrane division. *Journal of Automata, Languages and Combinatorics*, **11**, 4 (2006), 423–434.
28. N. Satish, M. Harris, M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. To Appear in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.
29. A. Ruiz, M. Ujaldon, J.A. Andrades, J. Becerra, K. Huang, T. Pan, J.H. Saltz. The GPU on biomedical image processing for color and phenotype analysis. *BIBE*, (2007), pp. 1124–1128.
30. S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, W. mei Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (2008), pp. 73–82.
31. S. Ryoo, C.I. Rodrigues, S.S. Stone, J.A. Stratton, Sain-Zee Ueng, S.S. Baghsorkhi, W.W. Hwu. Program optimization carving for GPU computing. *J. Parallel Distrib. Comput.*, **68**, 10 (2008), 1389–1401.
32. Nvidia CUDA Programming Guide 2.0, (2008): `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`
33. GPGPU organization. World Wide Web electronic publication: `http://www.gpgpu.org`
34. Nvidia CUDA. World Wide Web electronic publication: `http://www.nvidia.com/cuda`