Simulation of Recognizer P Systems by Using Manycore 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,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 the P systems computations by using a massively parallel computer architecture, such as Compute Unified Device Architecture (CUDA) from Nvidia, to obtain better performance in the simulations. 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 [20]. 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.

Since the model was presented, many software applications have been produced [9]. The common purpose of all these software applications is to simulate P systems devices (cellular machines), and hence the designers have faced similar difficulties. However, these systems were usually focused on, and adapted for, particular cases, making it difficult to work on generalizations.

P systems simulators are tools that help the researchers to extract results from a model. These simulators have to be as much efficient as possible when handling large problem sizes. The massively-parallel nature of the P systems computations points out to looking 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 huge performance [12], [15], [23]. Current NVIDIA GPUs, for example, contain up to 240 scalar processing elements per chip [14], and they are programmed using C and CUDA [26], [17].

In this paper we present a massively parallel simulator for the class of recognizing P systems with active membranes using CUDA. The simulator executes the P system which is defined by using the P-Lingua [4] programming language. The simulator is divided in two main stages: the *selection stage* and the *execution stage*. At this development stage, the *selection stage* is executed, in a parallel fashion, 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 computationally universal [4].

For the sake of simplicity, we shall focus in this paper on a model, *P systems with active membranes.* It 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 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) $[a \rightarrow v]_h^\alpha$, where $h \in H$, $\alpha \in \{+, -, 0\}$ (electrical charges), $a \in O$ and v is a string over O describing a multiset of objects (*object evolution rules*).

(b) $a\,[\;]_h^{\alpha} \rightarrow [b]_h^{\beta}$, 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) $[a]_h^{\alpha} \rightarrow [\;]_h^{\beta} b$, 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) $[a]_h^{\alpha} \rightarrow b$, 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) $[a]_h^{\alpha} \rightarrow [b]_h^{\beta}\,[c]_h^{\gamma}$, 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 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.

Recognizing P systems were introduced in [21], and constitute the natural framework to study the solvability of decision problems, since deciding whether an instance has an affirmative or negative answer is equivalent to deciding if a string belongs or not to the language associated with the problem.

In the literature, recognizing 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 [4].

In this paper, we present a simulation tool to simulate recognizer P systems with active membranes. The act of simulating something generally entails representing certain key characteristics or behavior of some physical, or abstract, system. On the other hand, 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) [28]. 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 whose have massively parallel applications. However, the problem was the way to program 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, programming environment tightly constrained, etc [15], [12]. 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.

The P system devices are massively parallel which 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 piece of data. This idea of thread is very important and used in parallel computing.

3 Underlying Architecture

This work uses a graphics processor unit (GPU) from Nvidia as hardware target for its study: Tesla C1060. This section introduces the Tesla C1060 computing architecture, and it shows architecture parameters that can affect the performance. In addition, it analyzes the threading model of Tesla architectures depending on its computing capability, and also the most important issues in the CUDA programming environment.

3.1 Tesla 10 Base Microarchitecture

The Tesla C1060 [14] is based on scalable processor array which has 240 streaming-processor (SP) cores organized as 30 streaming multiprocessor (SMs). The applications start at the host side (the CPU) which communicates with the device side (the GPU) through a bus, which is a PCI Express bus standard (see Figure 1).

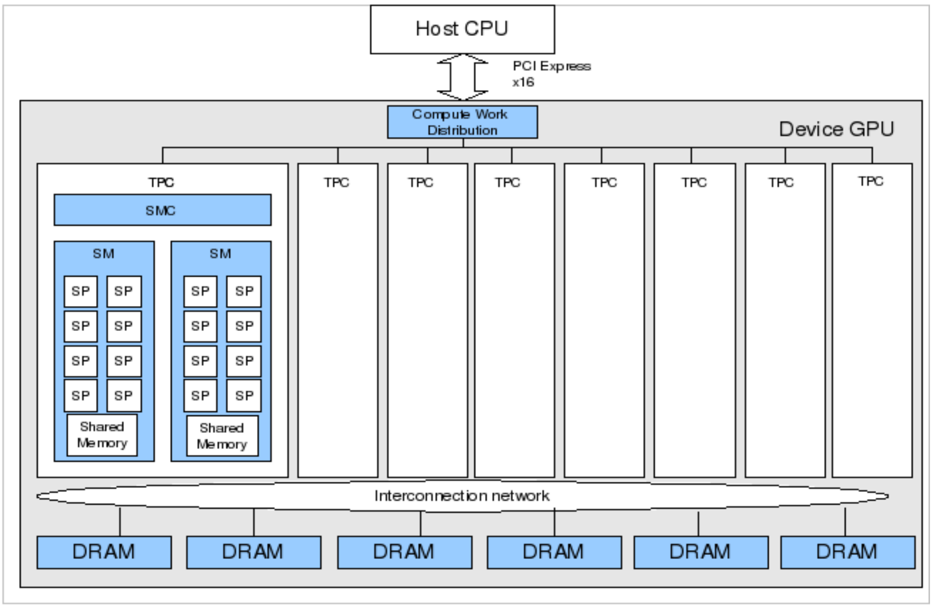

Fig. 1. Tesla Unified Architecture. TPC: Texture/processor cluster. SM: Streaming Multiprocessor, distributed among TPCs. SP: Streaming Processor.

The SM is the processing unit and it is unified graphics and computing multiprocessor. The parallel computing programs are programmed using ANSI C programming language along with CUDA extensions [28].

Every SM contains the following units: eight SPs arithmetic cores, one double precision unit, an instruction cache, a read only constant cache, 16-Kbyte read/write shared memory, a set of 16384 registers, and access to the off-chip memory (device/local memory).

The local and global (device) memory spaces are not cached, which means that every memory access to global memory (or local memory) generates an explicit memory access. A multiprocessor takes four clock cycles to issue one memory instruction for a "Warp" (see next subsection). Accessing local or global memory incurs an additional 400 to 600 clock cycles of memory latency [26], that is more expensive than accessing share memory and registers that incurs 4 cycles.

The Tesla C1060 achieves 102 GB/sec of bandwidth to the off-chip memory (running at 800 MHz). This bandwidth is not enough for the big set of cores and the possibilities to saturate it are high. To obtain the maximum bandwidth available it is needed to coalesce accesses to the device memory. The coalesced accesses are obtained whenever the accesses are contiguous 16-word lines, otherwise a fraction of this bandwidth it is obtained. Coalesced accesses will be a critical point in the optimization process.

In addition, the threads can use other memories like constant memory or texture memory. Reading from constant cache is as fast as reading from a registers, as long as all threads in the same warp read the same address. Texture Memory is optimized for 2D spatial locality (see Table 1).

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 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 [14]. 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 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 execute 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 inactive due to branching or predication, and this 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.

Table 2. Major Hardware and Software Limitations programming 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

3.3 Parallel Computing with CUDA

The GPU is, nowadays, a single-chip massively parallel system which is inexpensive and readily available. However, programming a highly-parallel system has historically been a domain of few experts [25]. The emergence of Compute Unified Device Architecture (CUDA) has helped develop highly-parallel applications easier than before. CUDA programming toolkit is an extension of ANSI C including several keywords and constructs.

The GPU is seen as a coprocessor that executes data-parallel **kernel** functions. The user creates a program encompassing CPU code (Host code) and GPU code (Kernel code). These are separated and compiled by nvcc (Nvidia's compiler for CUDA code). The host code is responsible for transfer data to and from the GPU memory (device memory) via API calls, to initiates the kernel code executed on the GPU.

The threads executes the kernel code, and they are organized into a three-level hierarchy. At the highest level, each kernel creates a single grid that consists of many thread blocks. Besides, each thread block can contain up to 512 threads which can share data through Shared Memory and can perform barrier synchronization by invoking the `--syncthreads` primitive [25]. On the other hand, blocks can not perform synchronization. The synchronization across blocks can only be obtained by terminating the kernel. Finally, the threads within the block are organized into warps of 32 threads.

Each block within the grid have their own identifier[18]. This identifier can be one, two or three dimensions depending on how the programmer has declared the grid. In the same way, 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.

4 A Design of the Simulator for the Class of Recognizing P Systems

4.1 Algorithm Design

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*, we have empirically determined to obtain the overall best performance on the Tesla C1060. 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.

4.2 P System Simulator with Active Membranes

The simulator simulates a recognizer P system with active membranes, i.e $\Pi = (O, H, \mu, \omega_1, \ldots, \omega_m, R)$ according to the notation described in section 2.

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 *selection stage* takes the major part of the simulation time in the sequential code, since this part of the algorithm implies to check all the rules of the system in every membrane. So we have parallelized the *selection stage* on the GPU, and the *execution stage* is still executed on the CPU at this point of the implementation.

The input data for the *selection stage* consists of the description of the membranes with their multisets (strings over O, labels associated with the membrane in H, etc...) and the set of rules R to be selected. The output data of this stage will be the set of selected rules per membrane. Only the *execution stage* changes the information of the configuration.

Besides, we have identified 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 execute, and each individual thread is responsible for identifying if there are some rules associated with the element that it represents, and if so, send it back to the *execution stage*. Finally, the CPU takes the control and executes the rules previously selected.

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 system response is reached.

Our simulator presents two restrictions: it can handle only two levels of membrane hierarchy for simplicity (the skin and the rest of elementary membranes), what is enough for solving lots of **NP**-complete problems; moreover, 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.

5 A Case Study: Implementing a Solution to the N-Queens Problem

In this section, we present a solution to the **N-Queens** problem, given by Miguel A. Gutiérrez–Naranjo et al [8], using our simulator. The **N-Queens** problem is 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 [22] 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. 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.

5.1 Implementation

P-Lingua 1.0[4] 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 2 shows the P-Lingua process to generate the input for our simulator.

P-Linga 2.0[5] 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 recognizer P system for solving the 3-Queens problem. This problem creates 512 membranes and up to 1300 different objects. For the 4-Queens problem, the system would create 65536 membranes and up to 8000

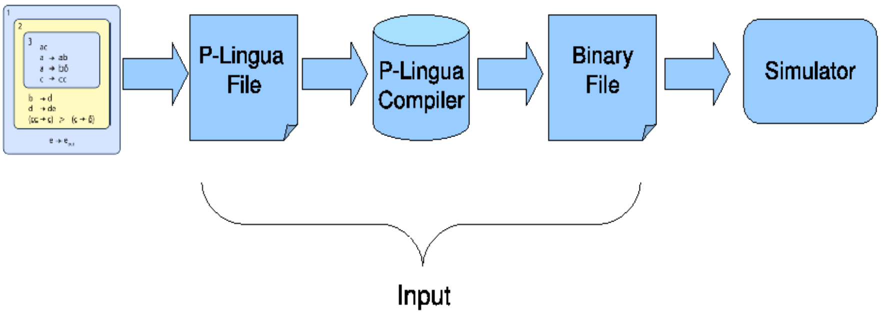

Fig. 2. Generation of the simulator's input

different objects. Currently, our simulator can not handle this example due to memory space limitation (requires up to 8GB in device memory). This problem can be solved with overlays of data and subsequent calls to the GPU. We are working on this solution, and also in other solutions to make possible execute problems with bigger memory size constraints on our simulator. 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 solution to 3-Queen problem previously explained 5.1. Although this problem does not cover all the **NP**-complete problems that we want to simulate in our simulator, it states an example of how a **NP**-complete problem can be solved on the P system with active membranes simulator. We report the *selection stage* time which is executed on the GPU, and compare it with the *selection stage* for the sequential code. We do not include the cost of transferring input data from host CPU memory across the PCI-Express bus to the GPU's on board memory. Selection is one building block of 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.

The *selection stage* on the GPU takes about *195 msec*. This is 12 times faster than the *selection stage* on the CPU which takes *2345 msec*. 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.

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. The membrane computation has 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 generate a solution to 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 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 the traditional machines.

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

On the other hand, we shall adapt our simulator to use the resources available on the GPU at maximum. To develop general purpose programs on the GPU is easier than several years ago with tools such as CUDA. However, extracting the maximum performance on the GPU is still hard, so we need to make a deep analysis to obtain the maximum performance available for our simulator.

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). This limits the size of the instances of **NP**-complete problems whose solutions can be successfully simulated. In the following version of the simulator, we will try to reduce the memory requirements for the simulator in order to be able to simulate bigger instances of **NP**-complete problems. Moreover, it would be interesting to design heuristics to accelerate the computations of our simulator.

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.

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 P08–TIC04200. The last three authors acknowledge the support of the project from the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03", as well as by the EU FP7 NoE HiPEAC IST-217068.

References

1. A. Alhazov, M.J. Pérez–Jiménez: Uniform solution of QSAT using polarizationless active membranes. In J. Durand-Lose, M. Margenstern, eds., *Machines, Computations, and Universality.* LNCS 4664, Springer, 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, Gh. Păun, M.J. Pérez–Jiménez, eds.: *Applications of Membrane Computing.* Springer, 2006.
4. D. Díaz–Pernil, I. Pérez–Hurtado, M.J. Pérez–Jiménez, A. Riscos–Núñez: P–Lingua: A programming language for membrane computing. In D. Díaz–Pernil, M.A. Gutiérrez–Naranjo, C. Graciani–Díaz, Gh. Păun, I. Pérez–Hurtado, A. Riscos–Núñez, eds., *Proceedings of the 6th Brainstorming Week on Membrane Computing*, Sevilla, Fénix Editora, 2008, 135–155.
5. M. García–Quismondo, R. Gutiérrez–Escudero, I. Pérez–Hurtado, M.J. Pérez–Jiménez: P–Lingua 2.0: added features and first applications. In this volume.
6. 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.
7. N.K. Govindaraju, D. Manocha: Cache–efficient numerical algorithms using graphics hardware. *Parallel Comput.*, 33, 10–11 (2007), 663–684.
8. M.A. Gutiérrez–Naranjo, M.A. Martínez–del–Amor, I. Pérez–Hurtado, M.J. Pérez–Jiménez: Solving the $N - Queens$ puzzle with P systems. In this volume.
9. M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez: Available membrane computing software. In G. Ciobanu, Gh. Păun, M.J. Peréz–Jiménez (eds.) *Applications of Membrane Computing*, Springer–Verlag, 2006. Chapter 15 (2006), 411–436.
10. 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.
11. M. Harris, S. Sengupta, J.D. Owens: Parallel prefix sum (Scan) with CUDA. *GPU Gems*, 3 (2007).
12. 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, 15–25.
13. 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, 63–74.
14. E. Lindholm, J. Nickolls, S. Oberman, J. Montrym: NVIDIA Tesla. A unified graphics and computing architecture. *IEEE Micro*, 28, 2 (2008), pp. 39–55.

15. 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, 896–907.
16. J. Michalakes, M. Vachharajani: GPU acceleration of numerical weather prediction. *IPDPS*, 2008, 1–7.
17. J. Nickolls, I. Buck, M. Garland, K. Skadron: Scalable parallel programming with CUDA. *Queue*, 6, 2 (2008), pp. 40–53.
18. J D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips: Gpu computing. *Proceedings of the IEEE,* 96, 5 (2008), 879–899.
19. 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), pp. 80–113.
20. Gh. 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.
21. 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.
22. M. J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrini: A polynomial complexity class in P systems using membrane division. In E. Csuhaj–Varjú, C. Kintala, D. Wotschke, G. Vaszil, eds., *Proceedings of the 5th Workshop on Descriptional Complexity of Formal Systems, DCFS 2003*, Budapest, 2003, 284–294.
23. 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, 1124–1128.
24. 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, 73–82.
25. 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.
26. NVIDIA CUDA Programming Guide 2.0, 2008: `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`
27. GPGPU organization. World Wide Web electronic publication: `www.gpgpu.org`
28. NVIDIA CUDA. World Wide Web electronic publication: `www.nvidia.com/cuda`