A massively parallel framework using P systems and GPUs

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*—Since CUDA programing model appeared on the general purpose computations, the developers can extract all the power contained in GPUs (Graphics Processing Unit) across many computational domains. Among these domains, P systems or membrane systems provide a high level computational modeling framework that allows, in theory, to obtain polynomial time solutions to NP-complete problems by trading time for space, and also to model biological phenomena in the area of computational systems biology. P systems are massively parallel distributed devices and their computation can be divided in two levels of parallelism: membranes, that can be expressed as blocks in CUDA programming model; and objects, that can be expressed as threads in CUDA programming model. In this paper, we present our initial ideas of developing a simulator for the class of recognizer P systems with active membranes by using the CUDA programing model to exploit the massively parallel nature of those systems at maximum. Experimental results of a preliminary version of our simulator on a Tesla C1060 GPU show a 60X of speed-up compared to the sequential code.**

I. Introduction

Conventional computers (electronic devices based on silicon) have increased their performance since the early days of computing, but this trend is limited by physical laws. Although many real-life problems can be solved in reasonable time, other relevant problems need an exponential amount of resources (time or space) to be solved. **NP**-problems belong to the last class, which are hard computational problems such as SAT, K-closure and Network reliability. Therefore, it seems reasonable looking for new computation paradigms that can be implemented in non-electronic devices (*Unconventional Computing*) to go beyond the limits of conventional computing.

Membrane computing (or cellular computing) is an emerging branch within Natural Computing that was introduced by Gh. Păun in [6]. 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*, and they consist of a set of *syntactic* components: a cell-like *membrane structure*, *multisets of objects* associated to each membrane, and *rules* executed in a synchronous non-deterministic maximally parallel manner.

There are different models of P systems that have been investigated in this area [9], and they are theoretically designed to solve diverse problems. In this work, we deal with P systems capable of constructing an exponential workspace (expressed by the number of membranes and objects) in polynomial time. They are based on the model of P systems with active membranes and membrane division, that abstracts the way of obtaining new membranes through the process of *mitosis*. It has been successfully used to design (uniform) solutions to well-known **NP**-complete problems, for example SAT [9] and *Subset Sum* problems, using the massively parallelism among membranes and objects to study all the possible instances of a **NP**-complete problem in parallel.

Up to now, it has not been possible to have implementations neither *in vivo* nor *in vitro* of P systems, so the manipulation and analysis of these devices is performed by simulations on conventional computers. However, sequential implementations like showed in [7][9] have very low performance, so the underlying architecture used to simulate P systems should get profit of their parallel nature to accelerate the simulations. In this sense, several authors have implemented parallel simulators for transition P systems on clusters [10], and many others have designed P systems hardware implementations [12][11]. In this work, we introduce a solution based on the GPU architecture to simulate P systems due to its massively chip-level parallelism.

In this paper, we present a simulator for recognizer P systems with active membranes using CUDA. CUDA programing model presents two levels of parallelism. The first one appears at block-level whereas the second one is presented at the thread-level. We identify those levels of parallelism intrinsic on CUDA with the two levels of parallelism that P Systems naturally have by its definition: among membranes and among objects within membranes. We test this simulator using a family of P systems that solve the N-Queens problem [13], obtaining up to 60X of speed-up compared to the sequential version of this simulator.

The rest of the paper is structured as follows. In Section 2 we outline P systems with active membranes. Section 3 introduces key points of CUDA used in our implementations and the GPU features used for this work. Section 4 explains the design of the simulator, and finally, the paper ends with some conclusions and ideas about future work.

II. P Systems with Active Membranes

P systems consist of a set of *syntactic* components: a cell-like *membrane structure* (it is a hierarchical rooted tree of compartments that delimit *regions*, where the root is called *skin*), *multiset of objects* (corresponding to chemical substances present inside the compartments (membranes) of a cell), and *rules* (corresponding to chemical reactions that can take place inside the cell) executed in a synchronous non-deterministic maximally parallel manner.

The *semantics* of P systems are defined through a non-deterministic synchronous model. A computation of a P system is made by steps called *configurations* (which identifies an instantaneous state of the system: the family of multisets and the membrane structure), assuming a global clock that synchronizes the execution.

The computation starts always with an *initial configuration* of the system, where the input data of a problem is encoded. The *transition* from one configuration to the next is performed by applying rules to the objects placed inside the regions. A computation of the system is a tree of configurations, which is made by transitions until reaching a *halting configuration*, where no more rules can be applied. The result of a halting computation is usually defined through the multiset associated with a specific output membrane, or the *environment*, which is the space out of the skin.

In the P systems with active membranes model, every elementary membrane is able to divide itself by reproducing its content into a new membrane. Specifically, this model is a construct of the form $\Pi = (O, H, \mu, \omega_1, \dots, \omega_m, R)$, 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, $\omega_1, \dots, \omega_m$ 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 one of the following types: *Evolution rules*, *Division rules*, *Dissolution rules* and *Communication rules* (*Send-out* or *Send-in*). See [8] for details.

P systems can be used for addressing the efficient resolution of decision problems. This kind of problems require either a *yes* or *no* answer. In this sense, we consider *recognizer P systems* as P systems with external output (the results of halting computations are encoded in the environment) such that the *yes* or *no* answer is sent in a halting configuration.

III. Parallel Computing on GPUs

In this section, we briefly introduce the NVIDIA Tesla C1060 GPU that we use in this work, and also the CUDA parallel programing model presented in [2][1]. The Tesla C1060 GPU which is based on scalable processor array that has 240 streaming-processor (SP) cores organized as 30 streaming multiprocessor (SMs). Applications start at host side (CPU side) which communicates with device side (GPU side) through PCI Express x16 bus. The SM is the processing unit and it acts both as unified graphics and computing multiprocessor. Every SM contains eight SPs arithmetic cores, a set of 16384 32-bit registers, 16-Kbyte read/write on-chip shared memory that has very low access latency, and also access to device or global memory.

A SM is a hardware device specifically designed with multithreaded capabilities. It manages and executes up to 1024 threads in hardware with zero scheduling overhead. Each thread has its own thread execution state and can execute an independent code path. The SMs execute threads in a Single-Instruction Multiple-Thread (SIMT) fashion (see [2][1]).

To develop parallel applications on the GPU we use C and C++ programing language along with CUDA extensions (Compute Unified Device Architecture)[15]. In the CUDA parallel programing [14], 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.

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

The programmer declares the number of threads block per grid and also the number of threads per thread block. On one hand, blocks in a Grid are declared in one or two dimensions, and all of them have their own and unique identifier. Similarly, the threads in a block can be declared in one, two or three dimensions having their own and unique identifier too. On the other hand, 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.

IV. Simulating P Systems with active membranes

In the design of the simulator, we have identified each membrane as a thread block where each thread represents an object of the alphabet O. It presents a strong restriction because a membrane would have only 512 objects which is not enough to solve many problems. Therefore, if the number of objects is bigger than 512, the objects are distributed equally among 256 threads that is our best empirically block size.

The simulator, which is based on the presented in [7], is executed into two stages: *selection stage* and *execution stage*. The selection stage consists of the search for the rules to be executed in each membrane. The *execution stage* consists of the execution of the rules previously selected. Therefore, it presents a need of global synchronization among each phase. The input data for the selection stage consists of the description of the membranes with their multisets (strings over the working alphabet O, labels associated with the membrane in H, etc...) and the set of rules R to be selected. The output data of this stage is the set of selected rules per membrane.

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

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

The selection stage is implemented as a CUDA *kernel*. Each thread block runs in parallel looking for the set of rules that has to select, and each individual thread is responsible for identifying if there are some rules associated with the object(s) that it represents. Moreover, it also includes the execution of the evolution rules. It is due to two main reasons: the evolution rules do not implies communication among membranes, and they are executed in a maximal manner. Finally, the selection kernel has to finish to obtain global synchronization between the selection stage and the execution stage.

However, the rest of the rules (not evolution) only can be executed one per membrane, and they entail communication among them. Therefore, we design the execution of those rules as different CUDA kernels, one kernel per each kind of rule (send-in communication, send-out communication, dissolution and division), giving a result of the *execution stage*. We use different kernels for the execution stage because, otherwise, we should implement a bigger kernel with branches to identify each kind of rule to be applied, and this model decreases the performance of our application.

V. Conclusion

In this paper, we present a massively parallel framework using GPUs and a class of recognizer P systems with active membranes. We show that GPUs are well suited to simulate membrane systems due to the double parallel nature that these systems present by their definition.

On one hand, using the power that provides GPUs to simulate P systems with active membranes is a new concept in the development of applications for membrane computing. On the other hand, P systems are an alternative approach to extract all performance available on GPUs due to its parallel nature.

This simulator is limited by the available resources on the GPU as well as the CPU. 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 to handle bigger instances of **NP**-complete problems. Besides, we will study the solution of specific problems, and also other kind of P systems.

Although the massively parallel environment that provides GPUs are good enough for the simulator, we need to go beyond. The newest clusters of GPUs provide 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 under grants “Consolider Ingenio-2010 CSD2006-00046“ and “TIN2006-15516-C04-03“. The last three authors acknowledge the support of the project TIN2006–13425 of the Ministerio de Educación y Ciencia of Spain, cofinanced by FEDER funds, and the support of the “Proyecto de Excelencia con Investigador de Reconocida Valía” of the Junta de Andalucía under grant P08–TIC04200.

References

[1] 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.
[2] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 28, 2 (2008), pp. 39–55.
[3] N. Satish, M. Harris, M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, May 2009.
[4] 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.
[5] 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), pp. 1389–1401.
[6] G. Păun. Computing with membranes. Journal of Computer and System Sciences, 61, 1 (2000), pp. 108–143, and Turku Center for Computer Science-TUCS Report No 208.
[7] M. García-Quismondo, R. Gutiérrez-Escudero, M.A. Martínez-del-Amor, E. Orejuela-Pinedo, I. Pérez-Hurtado. P-Lingua 2.0: A software framework for cell–like P systems. Int. J. of Computers, Communications and Control, Vol. IV (2009), 3, pp. 234-243.
[8] G. Păun. Membrane Computing: An Introduction. Springer, (2002).
[9] G. Ciobanu, M.J. Pérez–Jiménez, G. Paun, editors. Applications of membrane computing. Natural Computing Series, Springer, (2006).
[10] G. Ciobanu, G. Wenyuan. P systems running on a cluster of computers. In C. Martin-Vide, G. Mauri, G. Pãun, G. Rozenberg, A. Salomaa (eds.) Workshop on Membrane Computing, vol. 2933 (2004), pp. 123–139.
[11] 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), 415.
[12] V. Nguyen, D. Kearney, G. Gioiosa. An algorithm for non-deterministic object distribution in P systems and its implementation in hardware, Membrane Computing, 9th International Workshop (2009), 325 - 354.
[13] 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), 199 - 210, in press.
[14] NVIDIA CUDA Programming Guide 2.0, (2008): http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf
[15] NVIDIA CUDA. World Wide Web electronic publication: www.nvidia.com/cuda