Accelerated Simulation of P Systems on the GPU: A Survey

Miguel A. Martínez-del-Amor, Luis F. Macías-Ramos, Luis Valencia-Cabrera, Agustín Riscos-Núñez, and Mario J. Pérez-Jiménez

Research Group on Natural Computing, Dept. Computer Science and Artificial Intelligence, University of Seville Avda. Reina Mercedes S/N, 41012, Sevilla, Spain
{mdelamor,lfmaciasr,lvalencia,ariscosn,marper}@us.es

Abstract. The acceleration of P system simulations is required increasingly, since they are at the core of model verification and validation processes. For this purpose, GPU computing is an alternative to more classic approaches in Parallel Computing. It provides a manycore platform with a level of high parallelism at a low cost. In this paper, we survey the developments of P systems simulators using the GPU, and analyze some performance considerations.

Keywords: Membrane Computing, Parallel Computing, GPGPU.

1 Introduction

Membrane Computing [13,14] defines a set of bio-inspired computing devices called *P systems*. They have several *syntactic* ingredients: a *membrane structure* which delimits *regions* where multisets of *objects* and sets of evolution *rules* are placed. P systems have also two main *semantic* ingredients: their inherent *parallelism* and *non-determinism*. The objects inside the membranes evolve according to given rules in a synchronous (in the sense that a global clock is assumed), parallel, and non-deterministic way.

In order to *experimentally validate* P systems based models [4], it is necessary to develop simulators that can help researchers to compute, analyze and extract results from them [14]. These simulators have to be as *efficient* as possible to handle large-size instances, which is one of the major challenges facing today's P systems simulators. In this concern, this parallel computation model leads to looking for a massively-parallel technology by which a parallel simulator can run efficiently. For example, current *Graphics Processor Units (GPUs)* [6] are *massively parallel processors*, featuring thousands of cores per chip [7].

In this paper, we survey the development of P systems simulators on the GPU, specially those available at the *PMCGPU* project [16].

The paper is structured as follows: Section 2 introduces the parallel simulation of P systems, and surveys GPU computing; Section 3 depicts the developed simulators for P systems on the GPU; and finally, Section 4 provides some conclusions and future research lines.

2 Solutions in High Performance Computing

According to [5], a good computing platform for simulating P systems should provide a balance between *performance*, *flexibility*, *scalability*, *parallelism* and *cost*. However, it is difficult for a computing platform to conform with all of them. A factor promoting one attribute might demote another. In fact, flexibility usually comes at expenses of both performance and scalability [8].

Concerning the types of computing platforms for simulating P systems, there are mainly three to mention [8]: *Sequential* (e.g. Java, C++, etc.), *Software-based parallel* (e.g. OpenMP, MPI, etc.), *Hardware-based parallel* (e.g. FPGAs, etc.). When designing parallel simulators for P systems, a developer has to deeply analyze the platform, and adopt different approaches to implement real parallelism of P systems. Today, a well-known parallel platform in *HPC (High Performance Computing)* is the *GPU* (Graphics Processor Unit), that contains thousands of computing processors. In this sense, NVIDIA GPUs can be programmed by using *CUDA (Compute Unified Device Architecture)* [15,7], which offers a programming model that abstracts the GPU architecture.

According to the CUDA programming model, the GPU (or *device*) run thousands of *threads* executing the same code (called *kernel*) in parallel. The threads are arranged within a grid in a two-level hierarchy: a grid contains a set of *thread blocks*, and each block can contain hundreds of threads. All blocks in a grid have the same number and organization of threads. The execution of threads inside a block can be synchronized by *barrier* operations, and threads of different blocks can be synchronized only by finishing the execution of the kernel.

The memory hierarchy in CUDA must be manually managed. There are mainly two to mention: *global* and *shared* memories. Global memory is the largest but the slowest memory in the system, whereas shared memory is the smallest but fastest memory. The latter is key to speedup a code. Moreover, the memory is static, so enough memory space has to be allocated before running a kernel. This is often accomplished by considering the worst case (in space) of a problem.

The real GPU architecture consists of a processor array, organized in *Streaming Multiprocessors (SMs)* of *Streaming Processors (SPs, or cores)*. SMs execute threads in groups of 32, called a *warp*. Threads of the same warp must start together at the same program address. However, they are free to branch and execute independently, but at cost of performance. If a warp is broken (because of branching or memory stall), the real parallelism in CUDA is not achieved.

Although CUDA programming model is flexible enough, the achieved performance depends on the design and implementation. There are several strategies to help improving performance: *emphasizing parallelism* (warps must be maximized with active threads, but minimizing branch divergence), and *exploiting memory bandwidth* (by coalesced access to contiguous memory positions).

Therefore, GPUs entail a software-based parallel computing platform, but with hardware flavor: only by optimizing the code for the GPU architecture can it achieve best performance. The GPU is considered as a relatively low cost technology, leveraging: *good performance*, *an efficiently synchronized platform* with *a medium scalability degree*, and *low-medium flexibility*.

3 Parallel Simulators on the GPU

Next, we briefly survey the parallel simulators available in the PMCGPU *(Parallel simulators for Membrane Computing on the GPU)* project [9,16].

PCUDA [1,9]: This simulator is designed for *recognizer (confluent) P systems with active membranes*. The parallel design takes advantage of the double-parallelism in GPUs to speedup the simulation of the double-parallelism in P systems: each thread block is assigned to each elementary membrane, and each thread is assigned to a portion of the objects defined in the alphabet (rules in this model does not have cooperation). The simulator allocates memory for all the defined objects within each membrane. Although this is the worst case to deal with, it does not take place in the most P systems. Thus, the achieved performance depends on the simulated P system. Two case studies has been simulated [9]: a simple test P system designed to stress the simulator (A), and a solution to SAT problem with active membranes [2] (B). The experiments report up to 7x of speedup for case study A, and 1.67x for B. Therefore, PCUDA simulators are highly flexible, but have low performance and low scalability.

PCUDASAT [2,3,9]: It is a set of simulators for a *family of P systems with active membranes solving SAT in linear time* [2]. The simulation algorithm is based on the stages identified in the solution, and so, in the computation of any P system in the family. The code is tailored to this family, saving space in the representation of objects, and avoiding storing the rules (they are implicit in the code). In the design, each thread block is assigned to each elementary membrane, and each thread to each object of the input multiset. The CUDA simulator achieves up to 63x of speedup. Although PCUDASAT simulators are less flexible, the performance and scalability have been increased, compared to PCUDA, for this special case study.

TSPCUDASAT [11,9]: It simulates a *family of tissue-like P systems with cell division solving SAT in linear time*. The simulation algorithm is based, as in PCUDASAT, in the 5 stages of the computation of the P systems. In the design, each thread block is assigned to each cell, but threads are used differently in each stage. Experiments show that the CUDA simulator achieves up to 10x. Therefore, simulating in CUDA two solutions to the same problem (SAT) under different P system variants leads to different speedups. Indeed, the usage of charges can help to save space devoted to objects.

ABCD-GPU [12,10,9]: These simulators are for *Population Dynamics P systems*, both on OpenMP and CUDA. They follow the DCBA algorithm [12], which is based on four phases. The multicore version is parallelized in three ways [10]: 1) simulations, 2) environments and 3) hybrid approach. Runtime gains of up to 2.5x, using 1), were achieved with a 4-core CPU. The design of the CUDA simulator is as follows [12]: environments and simulations are distributed along thread blocks, and rule blocks among threads. Phases 1, 3 and 4 of DCBA were efficiently executed on the GPU, however Phase 2 was poorly accelerated (it is inherently sequential), becoming the bottleneck. The GPU simulator were benchmarked by a set of randomly generated PDP systems, achieving speedups of up to 7x. Experiments indicate the simulations are memory-bandwidth bound.

4 Conclusions and Future Work

The idea of using the CUDA to develop P systems simulators is to take advantage of the highly parallel architecture, the low cost and the easy synchronization of a GPU.

Flexible CUDA simulators for P systems (PCUDA and ABCD-GPU) statically allocate memory to represent objects using as worst case the whole alphabet. Although it is a naive solution, the access to data is made very efficiently. However, the performance of the simulator totally depends on the simulated P system, since the representation of multisets can be very sparse. There are two measures for P systems associated with performance [9]: *density of objects per membrane* (the ratio of number of objects actually appearing in the regions to the alphabet size), and *rule intensity* (the ratio of rules being applied to the total amount of rules). They should be maximized for better speedups.

Moreover, it has been shown that P systems simulations are both memory and memory-bandwidth bound: the performance of the simulation is relatively low compared with the computing resources available in the GPU. The main reasons are that simulating P systems requires a high synchronization degree (e.g. the global clock of the models, handling rule cooperation and rules competition, etc.), and the number of instructions to execute per memory portion is small. They restrict the design of parallel simulators, since a bad step taken on GPU programming can dramatically demote performance.

Therefore, further improvements of the designs have to be done, so that next generation of simulators on the GPU can take advantage of the full hardware and efficiently encode the representation of P systems.

Acknowledgements. The authors acknowledge the support of the Project TIN2012-37434 of the "Ministerio de Economía y Competitividad" of Spain, cofinanced by FEDER funds. M.A. Martínez-del-Amor also acknowledges the support of NVIDIA for the CUDA Research Center program at the University of Seville, and of the 3rd postdoctoral phase of the "PIF" program associated with the Project of Excellence under grant P08-TIC04200 of the "Junta de Andalucía".

References

1. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P systems with Active Membranes on CUDA. Briefings in Bioinformatics 11(3), 313–322 (2010)
2. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulating a P system based efficient solution to SAT by using GPUs. Journal of Logic and Algebraic Programming 79(6), 317–325 (2010)
3. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Jiménez, M.J., Ujaldón, M.: The GPU on the simulation of cellular computing models. Soft Computing 16(2), 231–246 (2012)

4. Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J.: Applications of Membrane Computing in Systems and Synthetic Biology. Series: Emergence, Complexity and Computation, vol. 7. Springer (2014)
5. Gutiérrez, A., Alonso, S.: P systems: from theory to implementation, ch. 17, pp. 205–226 (2010)
6. Harris, M.: Mapping computational concepts to GPUs. In: ACM SIGGRAPH 2005 Courses, NY, USA (2005)
7. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands on Approach, MA, USA (2010)
8. Nguyen, V., Kearney, D.A., Gioiosa, G.: Balancing performance, flexibility, and scalability in a parallel computing platform for membrane computing applications. In: Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2007. LNCS, vol. 4860, pp. 385–413. Springer, Heidelberg (2007)
9. Martínez-del-Amor, M.A.: Accelerating Membrane Systems Simulators using High Performance Computing with GPU. Ph.D. thesis, University of Seville (2013)
10. Martínez-del-Amor, M.A., Karlin, I., Jensen, R.E., Pérez-Jiménez, M.J., Elster, A.C.: Parallel simulation of probabilistic P systems on multicore platforms. In: Proceedings of the Tenth Brainstorming Week on Membrane Computing, vol. II, pp. 17–26 (2012)
11. Martínez-del-Amor, M.A., Pérez-Carrasco, J., Pérez-Jiménez, M.J.: Characterizing the parallel simulation of P systems on the GPU. International Journal of Unconventional Computing 9(5-6), 405–424 (2013)
12. Martínez-del-Amor, M.A., Pérez-Hurtado, I., Gastalver-Rubio, A., Elster, A.C., Pérez-Jiménez, M.J.: Population Dynamics P Systems on CUDA. In: Gilbert, D., Heiner, M. (eds.) CMSB 2012. LNCS, vol. 7605, pp. 247–266. Springer, Heidelberg (2012)
13. Păun, G.: Computing with Membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000) and Turku Center for CS-TUCS Report No. 208 (1998)
14. Păun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, USA (2010)
15. NVIDIA CUDA website (2014), `https://developer.nvidia.com/cuda-zone`
16. The PMCGPU project (2013), `http://sourceforge.net/p/pmcgpu`