Enhancing the simulation of P systems for the SAT problem on GPUs

J.M. Cecilia, J.M. García, G.D. Guerrero
Computer Engineering and Technology Dept.
University of Murcia, Spain

M.A. Martínez-Amor, M.J. Pérez-Jiménez
Computer Science and Artif. Intel. Dept.
University of Seville, Spain

M. Ujaldón
Computer Architecture Dept.
University of Malaga, Spain

***Abstract*—GPUs constitute nowadays a solid alternative for high performance computing, and the advent of CUDA/OpenCL allow programmers a friendly model to accelerate a broad range of applications. The way GPUs exploit parallelism differ from multi-core CPUs, which raises new challenges to take advantage of its tremendous computing power. In this respect, P systems or Membrane Systems provide a high-level computational modeling framework that combines the structure and dynamic aspects of biological systems while being inherently parallel and non-deterministic. In this work, we implement on GPUs the simulation for a solution provided by Membrane Computing to solve the Satisfiability (SAT) problem. The overall speed up reaches 100x versus a sequential CPU, with an additional 16x due to CUDA optimizations. A promising scalability is also proven on more sophisticated GPU clusters and/or demanding problem sizes.**

I. Introduction

Membrane Computing is a paradigm introduced by Gh. Paun [6] and inspired on living cells where biochemical processes are executed. Devices of this model are called **P systems**. Figure 1 outlines their main components, which consist of two syntactic elements: a membrane structure, composed of a hierarchical arrangement of membranes embedded in a skin membrane, and delimiting regions or compartments holding multisets of objects, and sets of evolution rules which are associated with the membranes. From a computational perspective, P systems are mainly characterized by inherent parallelism and non-determinism.

Solutions to NP-complete problems in Membrane Computing at polynomial (often linear) time are achieved by trading time for space. This is inspired by the capability of cells to produce an exponential number of new membranes by methods like mitosis (cell division) or autopoiesis (membrane creation). Based on them, different models of P systems have emerged, and some of them were proven to be computationally universal.

A milestone for researchers within this area is to find a feasible biological implementation, either in vivo or in vitro. In the meantime, studies on P systems are driven by simulators [2], [3] programmed using Java, C or Prolog to end up running on sequential architectures which undermine its potential parallelism. In this work, we introduce the simulation of a class of recognizer P systems with active membranes on GPUs solving the Satisfiability (SAT) problem in linear time. We summarize the behavior of the solution provided in [1] through a simulator and propose different parallel alternatives:

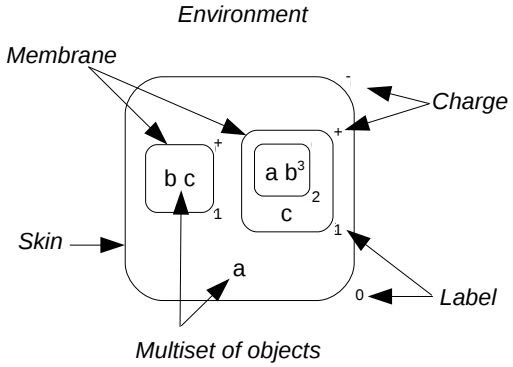


Fig. 1. Internal structure of a recognizer P system.

1) **Baseline.** Our first solution simulates a complete theoretical computation of this particular P system, overcoming previous generalities. Through a hybrid implementation on CPUs and GPUs, we drastically reduce the memory usage to take a step forward in the overall simulation performance for a 93x speed-up factor versus the CPU alternative.
2) **Enhanced:** We use some heuristics to adapt the computation of the P system to the GPU architecture, enhancing parallelism while reducing the synchronization and communication bottlenecks.
3) **MultiGPU:** The GPU memory size is identified as a major constraint due to the exponential workspace created along the computation, which can be overcome with a data partition on a cluster of GPUs.

II. Recognizer P systems with active membranes

Recognizer P system with active membranes [5] is a tuple of the form $\Pi = (O, H, \mu, \omega_1, \ldots, \omega_m, R)$, where $m \geq 1$ is the initial degree of the system; O is the alphabet of *objects*, H is a finite set of *labels* for membranes; μ is a membrane structure (a rooted tree), consisting of m membranes injectively 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 μ; and R is a finite set of *rules* in 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 associated with membranes and depending on the label and the charge of the membranes (*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 an 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, which 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.
- Rules from (a) to (e) are used in a maximal parallel way.
- Rules (b) to (e) cannot be applied simultaneously in a membrane in one computation step.
- 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.

Data representing an instance of the problem is provided to the P system to compute the required answer. This is done by codifying each instance as a multiset placed in an $inputmembrane$. The output of the computation, yes or no, is sent to the environment in the last step of the computation.

III. Solving the SAT problem in linear time

The propositional Satisfiability problem (SAT) was the first known **NP**-complete problem. Given a formula of the propositional calculus, φ, the goal is to find if there is an assignment of truth values to its variables for which such formula evaluates to true. Assuming φ in Conjunctive Normal Form (CNF) with n variables and m clauses, the time to solve the problem using all known deterministic brute force algorithms grows exponentially with n and m.

In [4], it is described a family of recognizer P systems with active membranes solving SAT in linear time, but at the expense of creating an exponential workspace.

IV. Tuning P systems for solving SAT on GPUs

The model of recognizer P systems with active membranes is defined to solve decision problems. Likewise, for each of these problems, the theoretical design of the P system can vary by using a small subset of the computational tools provided by the model. The simulation of a given P system can thus be accelerated by removing general constraints and by adapting it to the architecture to be simulated. Our work here applies this methodology to the particular case of the SAT problem and the GPU architecture.

Our departure point is a generic simulator of a recognizer P system with active membranes previously developed in [1]. The tool is designed to simulate a wide range of recognizer P systems with active membranes, though this generality sometimes penalizes performance. Our baseline simulator was enhanced with a removal of general conditions and a reduction of memory requirements to pursue higher performance.

To increase the memory available for the simulation while improving performance, we have developed a third implementation aimed to multiple GPU systems. Figure 2 shows the data partition for the case of four GPUs.

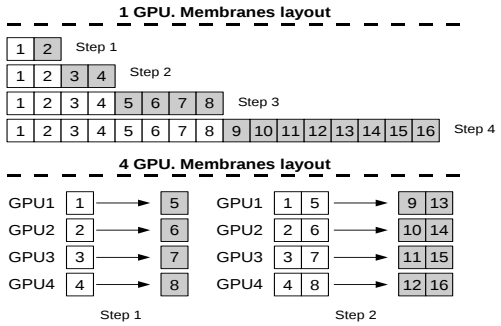


Fig. 2. Data partition on a four GPU system.

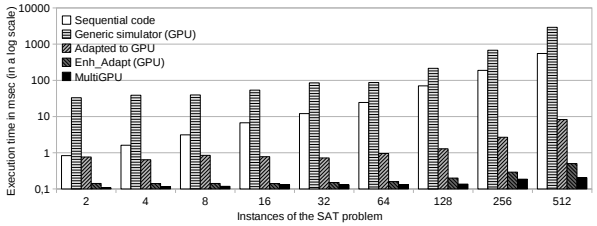
V. Experimental Results

Our hardware platform is a four-socket 2.40 GHz quad-core Intel Xeon E5530 CPU and four Nvidia Tesla C1060 GPUs. We use the gcc v. 4.32 C++ compiler with the `-O4` flag on the CPU, and the CUDA 2.3 SDK on the GPU side.

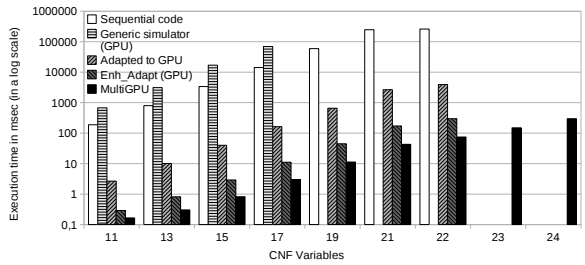
We analyze the performance of our simulators when increasing: (1) the number of threads per thread block, and (2) the number of thread blocks per grid. Benchmarks are generated through the WinSAT program, which generates random SAT problems in DIMACS CNF format file from a given number of variables *(n)*, clauses *(m)* and literals per clause *(k)*.

P systems exhibit two levels of parallelism suitable for GPUs: Among objects, which can be mapped to GPU threads, and among membranes, which may define CUDA blocks. Figure 3 shows the GPU execution times (in msecs., log scale) for the baseline simulator described in Section IV, where a thread per object (or set of objects) is defined in the alphabet O for each block. This generality causes a performance penalty which may lead the code to run even slower on GPUs than the counterpart C++ version we have implemented on CPUs. In addition, up to 17 CNF formula variables may be used in this version for not to exceed memory capacity.

Figure 3(a) illustrates the behavior of our simulator along objects, where performance goes up with a higher degree of thread level parallelism. We raise the number of variable instances ($k = \alpha$) per clause in the SAT formula to increase the number objects, and so the threads per block in the CUDA codes, and keep constant ($n = 11$) the number of membranes in the P system to maintain the number of CUDA thread blocks as $numberMembranes = 2^n$. The highest speed up (70%) is



(a) Varying the number of instances (objects) for 11 variables in the CNF.



(b) Varying the number of variables (membranes) in the CNF.

Fig. 3. Exec. times (msecs., log scale) for our simulator on CPUs and GPUs.

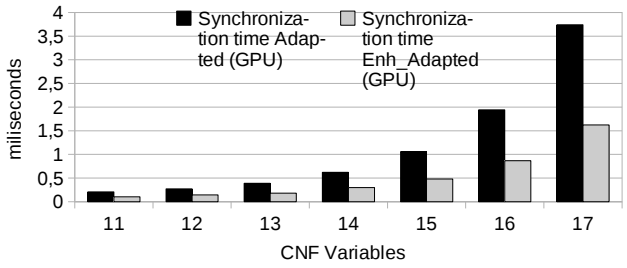


Fig. 4. Sync. time for our initial and enhanced GPU versions of the simulator.

obtained for a configuration of 256 instances per clause, which represents 256 threads per CUDA block. This guarantees a full occupancy of GPU resources.

Similarly, Figure 3(b) analyzes coarse-grained parallelism. Now, we vary the number of variables in the SAT formula while keeping constant the number of 256 instances per clause. In this case, a higher speed-up factor of up to 93x is attained.

The enhanced version of our simulator reaches a 15-16x speed up factor versus our reference implementation. Figure 4 shows the synchronization times spent by these two codes, which varies depending on the problem size, but the reduction in the number of kernels used for the second version saves 50% of the synchronization time on average. We may also point out that times in Figure 5 include the CUDA runtime overhead. The impact of the data movement through PCI-Express bus is completely hidden by the computation, but at the same time, the CUDA runtime API initialization results expensive.

Finally, Figure 3 shows an outstanding scalability when using more demanding problem sizes and/or migrating the code towards a GPU cluster, reaching almost an optimal 4x speed up factor when running on four GPUs, despite of the start-up overhead, partition the data, creating the CPU threads to transfer data to GPUs, and so on.

The largest SAT problem which we have successfully simu-

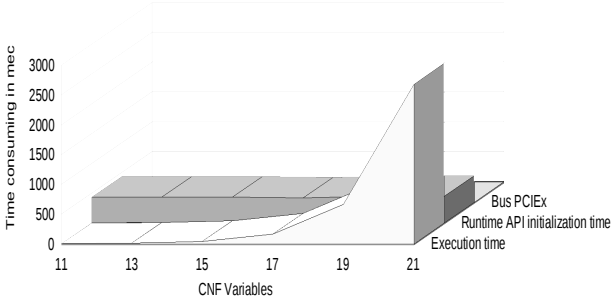


Fig. 5. Cumulative impact of various GPU runtimes for the initial version of our simulator.

lated on our single GPU simulators involves 22 variables with 47 clauses and 235 variable instances. This set up reaches the limit of 4 GB of device memory available on the Tesla C1060. For our multi GPU simulator (16 GB overall), we may use up to 24 variables, 20 clauses and 200 variable instances.

VI. Conclusions

We develop a GPU implementation for a recognizer P systems with active membranes. The Satisfiability problem (SAT) was taken as benchmark, and a generic simulator [1] as a departure point to adapt it to the GPU architecture.

Our implementation reaches up to 93x of speed up compared to the sequential version of the simulator, and defeats by a wide margin the generic simulator. Our experimental results also demonstrate that P system simulation can be accelerated on GPUs as much as 15-16x as long as their design is adapted to the particular idiosyncrasies of GPUs.

Downsides come from the nature of P systems with active membranes, where an exponential workspace can be constructed to achieve polynomial time solutions for NP-complete problems, limiting the size of the NP-complete problem instances whose solutions can be successfully simulated due to memory constraints. In order to partially overcome this issue while improving performance, we implement the P system simulation on a cluster of GPUs where NP-problems of larger sizes can now be simulated without sacrifying performance.

Overall, Membrane Computing is an emerging bio-inspired computational field where the exponential workspace created by the P system simulation poses challenges in memory space, but where GPUs may offer a high performance solution with an extraordinary scalability.

References

[1] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. M. del Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulation of p systems with active membranes on cuda. *Briefings in Bioinformatics*, Parallel and Ubiquitous methods and tools in Systems Biology, 2010.

[2] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of P-lingua 2.0. *Membrane Computing: 10th Intl. Workshop, LNCS*, 5957:264–288, 2010.

[3] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez. Available membrane computing software. In *Applications of Membrane Computing*, pages 411–436. 2006.

[4] M. J. P. Jiménez, álvaro Romero Jiménez, and F. S. Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing: an international journal*, 2(3):265–285, 2003.

[5] G. Paun. Membrane computing. an introduction. pages XI+419, 2002.

[6] G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61 (1):108–143, 2000.