The Reduction Problem in CUDA and Its Simulation with P Systems

Rodica Ceterchi[1], Miguel Ángel Martínez-del-Amor[2], Mario J. Pérez–Jiménez[2]

[1] Faculty of Mathematics and Computer Science
University of Bucharest
14 Academiei st. 010014 Bucharest, Romania
E-mail: `rc@fmi.unibuc.ro, rceterchi@gmail.com`

[2] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: `mdelamor@us.es, marper@us.es`

Summary. We introduce P systems with dynamic communication graphs which simulate the functioning of the CUDA architecture when solving the parallel reduction problem.

1 Introduction

Introduced in [13], P systems are powerful computational devices, with a high degree of parallelism, whose functioning is inspired by biological processes at the level of the cells, and of their membranes ([13],[14]). Among these processes, rewriting and communication play an important role.

It is of interest to compare P systems with other, classical, parallel paradigms. We have begun such a study in the form of simulating parallel classical architectures with P systems, in particular, the perfect shuffle architecture [4], [5] and the mesh architecture [6]. The reduction problem, being one of the most simple, primary ones to be solved in different contexts, was used as an illustration.

In [7] some general guidelines were developed along which a wide class of parallel architectures can be simulated with P systems. P systems with dynamic communication graphs were introduced. In their functioning, rewriting steps and communication steps are separated and made more visible. They differ from the systems introduced in [3] in that the communication graphs are inspired by the particular parallel network architecture being simulated.

CUDA stands for *Compute Unified Device Architecture* [15, 10], and is a technology proprietary of NVIDIA Corp. Since its introduction in 2007, CUDA has allowed programmers to take advantage of the inherent parallel architecture of GPUs, which ranges from 240 cores (Tesla C1060, released on 2008) to 2880 cores (Tesla K40, released on 2013). This is performed by using a threaded, shared-memory, abstracted model of the GPU, which is implemented by C/C++ extensions. CUDA has helped to establish GPU computing [9] as a sub-framework of High Performance Computing. In fact, it has been successfully applied to a broad spectrum of research areas, including Systems Biology and Population Dynamics [12], and Membrane Computing [1, 2, 11], among others.

In the present paper we propose to simulate with P systems the reduction problem as solved in CUDA. Section 2 is devoted to the presentation of several improved versions of solving the reduction problem in CUDA. Section 3 is devoted to the presentation of its simulation with P systems with dynamic communication graphs.

2 Solving the Reduction Problem in CUDA

The *GPU* (Graphics Processor Unit) is the core of graphics cards. A GPU today contains thousands of computing processors devoted for graphics. However, novel techniques enable programmers to take advantage of this highly parallel architecture for scientific computing. These are called *GPGPU* (General Purpose computing on the GPU) [9].

A new era of GPGPU started with the introduction of *CUDA* (Compute Unified Device Architecture) [15, 10] by NVIDIA. It offers a programming model that abstracts the GPU architecture to programmers, so it is enough to learn some extensions to C/C++ language (CUDA extensions), whereas the CUDA driver will execute the code on the GPU. In the following sections we will introduce some concepts and terminology of CUDA which are necessary to understand the work presented in this paper.

2.1 CUDA programming model

The CUDA programming model assumes that the CPU (or *host*) takes control of the execution flow, and permit the GPU (or *device*) to run many instances of the same code in parallel. This code is called *kernel*, and it is executed by a *grid* of *threads*. Typically, a grid is composed of thousands of threads, since the creation of a sufficient number of threads to use all hardware resources requires a large amount of data parallelism. The threads are arranged within the grid in a two-level hierarchy, as seen in Figure 1. At the higher level, each grid consists of one or more *thread blocks*. At the lower level, each block is organized as a three dimensional array of threads. All blocks in a grid have the same number and organization of threads. Each block is identified by a two dimensional identifier, and each thread within its block by a three dimensional identifier (ID). Therefore, any thread can be unequivocally identified by the union of both thread and thread block identifiers. The execution of threads inside a block can be synchronized by *barrier* operations (`__syncthreads()`), and threads of different blocks can be synchronized only by finishing the execution of the kernel.

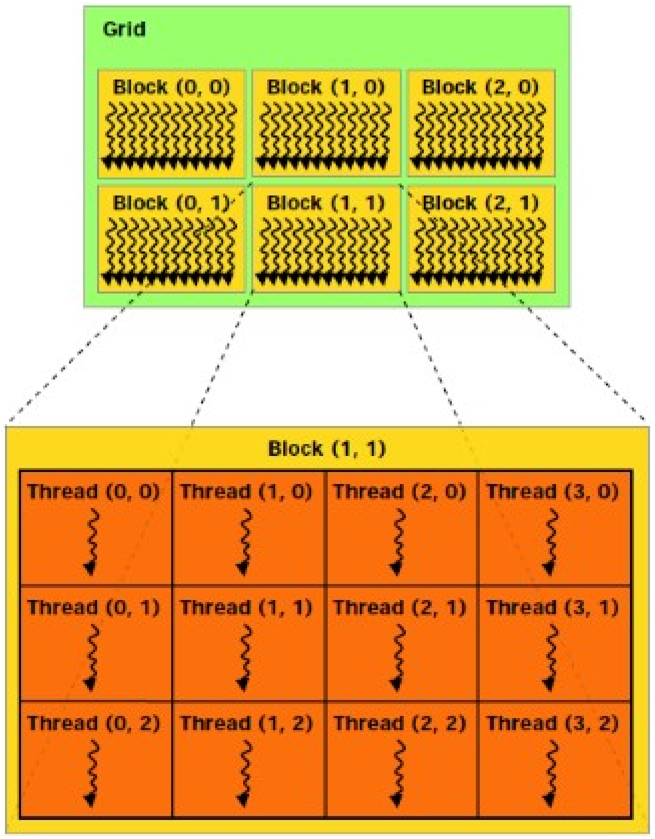

Fig. 1. Threading model in CUDA. Threads are executed in a grid, and they are organized in blocks.

The memory hierarchy is explicitly and manually managed in CUDA. This memory model is composed in several levels, each one offering different speeds and storage properties. We highlight the two most important ones: *global* memory and *shared* memory. Global memory is the largest but the slowest memory in the system. It is accessed by the host (where the input and output data are allocated) and by any thread in execution. Shared memory is the smallest but fastest memory. It is accessed by threads belonging to the same block. Normally, performance of CUDA applications depends on how much shared memory is exploited. Thus, an efficient way to structure an algorithm is as follows:

1. The threads of each block read its corresponding data portion from global memory to shared memory (which is inevitable because the host only can put the data in global memory).
2. Threads work with the data directly on the shared memory.
3. Threads copy these data back to global memory (so the host can retrieve the result).

2.2 Modern GPU architecture

The GPU architecture has evolved in the last years, offering even more compute capabilities. In general terms, it consists of a scalable processor array, organized in *Streaming Multiprocessors (SMs)* of *Streaming Processors (SPs, or cores)*. The number of them depends on the GPU. SMs are based on the *SIMT (Single-Instruction Multiple-Thread)* model. Basically, in the SIMT model all the threads execute the same instruction on different piece of data. SMs create, manage, schedule and execute threads in groups of 32 threads (which is the branching granularity of NVIDIA GPUs). This set of 32 threads is called *warp*, and each SM can handle many of them. Individual 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 serialization and performance (in fact, SIMT is really applied to the warp). If a warp is broken (because of branching or memory stall), the real parallelism in CUDA is not achieved.

2.3 Performance considerations

Although CUDA programming model is flexible enough to run any kind of algorithm, the achieved performance depends on how the programmer had designed the code, and on the target GPU running the program. A CUDA programmer has to perfectly know the CUDA programming model, but also the idea of the GPU architecture, since it provides the restrictions to be considered in order to achieve peak performance. There are several strategies to accomplish it. Next, we stand out two of them:

- *Emphasize parallelism*: the warp is the branching granularity on CUDA; that is, the parallelism unit. Thus, warps must be maximized with active threads, but minimizing branch divergence between thread: they must be executing the same instruction simultaneously to reach peak performance.
- *Exploit memory bandwidth*: the peak bandwidth of using both global and shared memories is achieved mainly by an access pattern: coalesced access to contiguous (aligned) memory positions. Data is transferred from memory to the GPU hardware in blocks, which is formed by contiguous bytes in memory. Thus, we must maximize these blocks with the access to contiguous memory addresses by contiguous threads within a warp.

2.4 Parallel Reduction in CUDA

The reduction problem consists in applying an operator to a set of elements. Let us assume a set of n elements $\{a_1, \ldots, a_n\}$, and the binary and associative, reduction operator $\oplus$. The result of applying reduction to the set of elements is another element $a = a_1 \oplus a_2 \oplus \ldots a_n$. Reduction is a well-known primitive in Parallel Computing, since it resides inside many important algorithms. For example, it can be used to compute the sum or the maximum of an array of numbers. Nowadays, reduce is part of the most used algorithm in Big Data and No-SQL data bases, which is Map-Reduce.

A common way to solve this problem in parallel is by using a tree, in which partial solutions are computed to reach the final one. The time complexity of this solution is $O(logn)$. The process is summarized in figure 2.

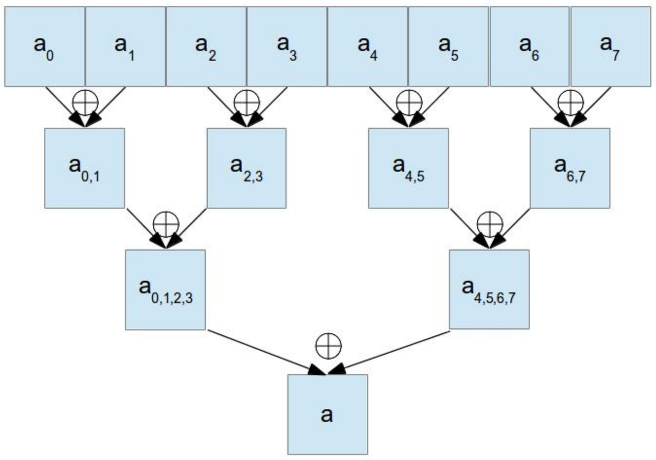

Fig. 2. Scheme of a parallel reduction solution

The most popular CUDA implementations for the reduction primitive can be found on the speech given by Harris [8] in 2007. This document introduces seven kernels from a didactic perspective, in a performance-increasing order. Next, we discuss the first four ones.

Interleaved addressing

Let us assume that the input array of elements is of size n, and that the maximum amount of threads per block is nt. That means that we will need to use $nb = \frac{n}{nt}$ thread blocks to process the whole array. However, if each block compute reduce for nt elements, what would we do with the nb partial results? The answer is to have a second kernel, with a single block having nb threads, which will compute again reduce. It is straightforward to add more kernels in this way until having $nb < nt$.

In what follows, we will assume that the size of the input array of elements is less than nt (maximum number of threads per block). This will mean that just one thread block is enough to compute reduce. We will disregard the second kernel for partial results.

A naive implementation of the tree solution in CUDA is to launch n threads and correspond each one with an element. First, each thread with even ID (called active threads) will compute the partial result with the next element, and the process will continue by halving the number of active threads. This process, called *reduce0* (interleaved addressing), is shown in Figure 3 (white-arrowed threads are inactive).

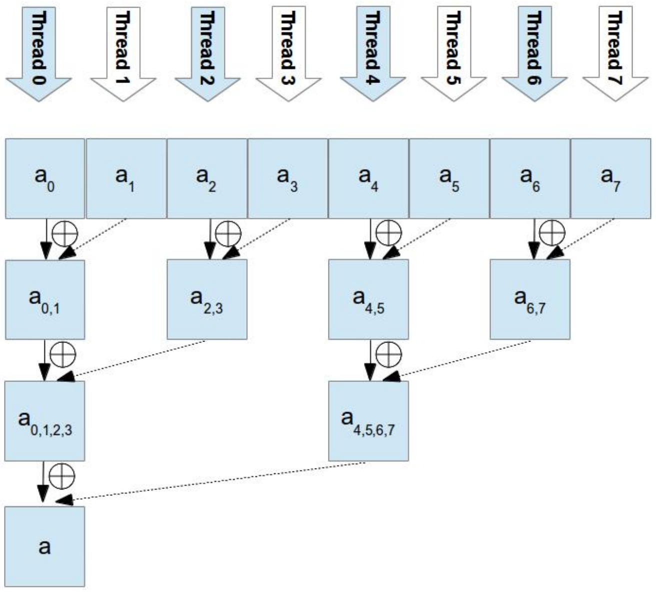

Fig. 3. Reduce0: interleaved addressing.

The main drawbacks of this solution are:

- Warps are not fulfilled. Since the addressing is interleaved, warps can be filled by active threads up to the half. Therefore, we are not maximizing memory.
- Access to memory is also interleaved, so the peak bandwidth cannot be reached.

Sequential addressing

A solution to the drawbacks found in interleaved addressing is to compact the memory accesses to contiguous threads. In order to implement this approach, it will be necessary to change the tree of the reduce primitive solution. Now, the first half of threads will access to the first half of the array to compute the partial solutions with the second half. It can be seen that the access is coalesced in this way, as well as warps are also fulfilled. Figure 4 shows the scheme of this approach, called *reduce 3*, sequential addressing. Again, white-arrowed threads are inactive in the beginning of the process.

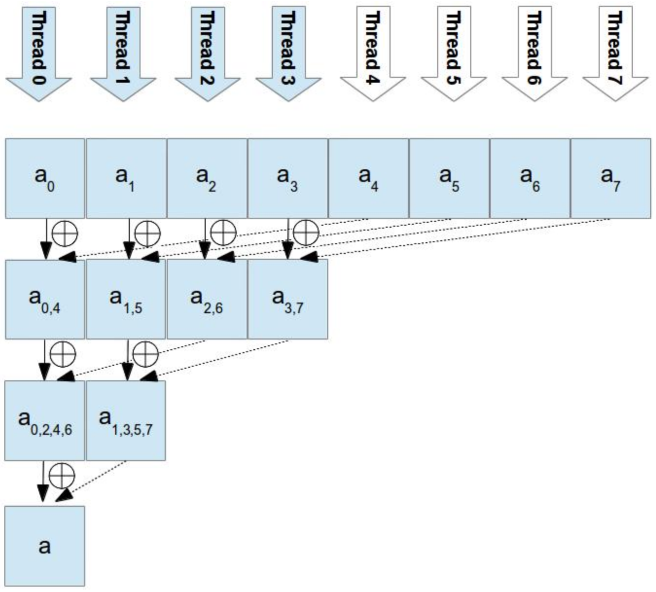

Fig. 4. Reduce3: sequential addressing.

As it can be noticed, the main drawback of this solution is that half of the threads are idle on first loop iteration, what is a waste of resources.

First add during load

The third approach, called *reduce4* first add during load, is based on taking advantage of the threads which are inactive at the beginning of reduce3. The idea is to halve the number of blocks, and to use all the threads at the beginning to apply the reduction operation to the corresponding element with the element that would have corresponded to the avoided extra block. That is, reduce4 will compute first the array $\{a_{0,8}, a_{1,9}, a_{2,10}, a_{3,11}, a_{4,12}, a_{5,13}, a_{6,14}, a_{7,15}\}$, and proceed as in reduce3.

3 The Simulation with P Systems

In this section we use the formal tools developed in [7] to produce a straightforward simulation with P systems of the reduction problem solved in CUDA as presented in section 2. In [7] only **SIMD** machines were considered, and algorithms for which communication took place only via a network of communication and not via a shared memory. The present case is different, we have communication via shared memory.

As a first step we construct for the **CUDA** model a P system $\Pi(\mathbf{C})$ in the spirit of Theorem 5 of [7]. The system must reflect in its membrane structure the particular CUDA architecture. As a second step, we follow Theorem 7 of [7], and construct $\Pi(\mathbf{C}, Y)$ for Y a reduction algorithm. We must specify for each algorithm the specific sequence of pairs $(graph, rules)$ which compose $R_\mu(Y)$.

Let *Graphs* denote the set of all possible graphs having n vertices labeled $P_1, \cdots, P_n$. Having fixed the vertices, each element of *Graphs* will be uniquely identified by the specific set of edges.

A distinguished element of *Graphs* is the *identity* graph, denoted in the sequel Id: (the set of vertices is fixed as mentioned above) the set of edges is defined as

$$Id = \{(i,i) \mid 1 \le i \le n\}.$$

Another distinguished element of *Graphs* is the *total graph*, denoted G_{total}: the set of edges is defined as

$$G_{total} = \{(i,j) \mid 1 \le i,j \le n\}.$$

One can also consider the *strict total graph*, denoted G^+_{total}: with set of edges defined as

$$G^+_{total} = \{(i,j) \mid 1 \le i,j \le n, i \ne j\} = G_{total} \setminus Id.$$

We recall from [7] the following two definitions.

Definition 3.1 *A P system* with dynamic communication graphs *is a construct*

$$\Pi = < V, P_1, \cdots, P_n, R_\mu >,$$

where $P_1, \cdots, P_n$ are elementary membranes, and V is an alphabet of symbols used to codify the contents of the membranes.

R_μ is a set of pairs $[graph, rules]$, with $graph \in Graphs$ and such that:

(i) if $graph \subseteq Id$ then its associated rules are rewriting rules;
(ii) if $graph \subseteq G^+_{total}$ then its associated rules are communication rules.

Definition 3.2 *A P system with dynamic communication graphs will be called* with finite sequential support *iff the set R_μ is both finite and totally ordered, i.e., if it is a finite sequence.*

Let V be an alphabet of symbols with which we will codify the contents of the membranes. An integer n will be codified as a^n (n apparitions of the symbol a, with $a \in V$). We assume we solve the reduction problem for a binary commutative and associative operation $*$, and we assume we can compute $n * m$ inside a membrane by rewriting: i.e. we have symbols $a, b \in V$ and a rewriting rule $r_*(a, b)$ such that $r_*(a, b)(a^n b^m) = a^{n*m}$.

We associate a hierarchical membrane structure to the CUDA components in the following manner: (1) each thread is an elementary membrane; (2) each block is a membrane containing the elementary ones associated to its threads; (3) the global memory is a separate elementary membrane. The presentation of this membrane structure is

$$\mu = (B_0(P_{00}, P_{01}, \cdots P_{0n}), \cdots, B_k(P_{k0}, P_{k1}, \cdots P_{kn}), M_g),$$

where $n = 2^t$ is the number of threads per block, M_g is the global memory membrane, P_{ij} is the membrane corresponding to thread j of block i, and when we reason inside a block the subscript corresponding to the block may be omitted.

If we have a set R_μ of pairs $(graph, rules)$ to obey the conditions of the definition, then the construct

$$\begin{aligned}\mathit{\Pi}(\mathbf{C}) = (V, (B_0(P_{00}, P_{01}, \cdots P_{0n}), \cdots, B_k(P_{k0}, P_{k1}, \cdots P_{kn}), M_g), R_\mu)\\ = (V, \mu, R_\mu)\end{aligned}$$

is a P system with dynamic communication graphs.

Here a discussion may start, comparing the P systems devised in [7] for **SIMD-X** machines, and a potential similar candidate for the CUDA paradigm. Such a candidate depends on a good definition for R_μ, or, at least, the formulation of criteria for 'admissible' candidates.

We open the way for this discussion, which is also a reflection on the power and the limitations of the formalism introduced in [7], by simulating the solving of the reduction problem in CUDA. More precisely, in the following we construct sets $R_\mu(Y)$ with the property that $\mathit{\Pi}(\mathbf{C}, Y) = (V, \mu, R_\mu(Y))$ is a P system with dynamic communications graph, with finite sequential support, which simulates Y, a reduction algorithm among the ones presented in Section 2.

The admissible communication graphs will have to reflect the communication properties of CUDA. We will have communication edges between M_g and each thread membrane P_{ji} to simulate the reading from and the writing to global memory. Between the individual elementary membranes which simulate the threads we can have communication only inside the same block, so the communication graph will have separate connected components for each block. Rewriting rules inside membranes will be associated to subgraphs of Id and communication rules to subgraphs of G^+_{total}.

We use the shorthand notation (A, B, x) for symbol x traveling on an oriented edge (A, B) from A to B, (G, x) for symbol x travelling on all oriented edges of G, and $(\{G_j\}_j, x)$ for symbol x traveling on all oriented edges of the family of graphs $\{G_j\}_j$.

The membrane M_g contains integers n_{ij} each codified with a symbol a_{ij}

$$M_g = \{a_{ij}^{n_{ij}} \mid i = 0, \cdots, k, j = 0, \cdots, n\}$$

The graph for loading from global memory will be $\{(M_g, P_{ij}) \mid i, j\}$, and writing from thread P_{ij} to global memory will use the edge (P_{ij}, M_g).

Loading the integers from global memory into the membranes corresponding to threads will be simulated by the sequence

$$(\{(M_g, P_{ij}, a_{ij})\}_{ij}, \{(P_{ij}, a_{ij} \to a)\}_{ij}),$$

where the first step is a communication step, and the second a rewriting step.

Writing to global memory from block k will be simulated by the sequence

$$((P_{k0}, a \to a_{k0}), (P_{k0}, M_g, a_{k0}))$$

where the first step is a rewriting step, and the second a communication step.

We now construct the graph for *interleaved addressing* and the sequence of rules which simulate the procedure reduce0 of section 2. We assume we are inside a block and we omit the block index. For a fixed stride s the graph of interleaved addressing will be

$$G_s = \{(P_{i+s}, P_i) \mid i \bmod (2s) = 0\}.$$

On one edge of this graph the sequence of rules to be applied is

$$((P_{i+s}, a \to ab), (P_{i+s}, P_i, b), (P_i, r_*(a, b))).$$

We first rewrite a to ab in P_{i+s}, then the b symbol travels to P_i, and finally in P_i the application of the rewriting rule r_* produces the desired result.

For the entire block, the sequence of rules for stride s will be

$$R_s = ((\{P_{i+s}\}_i, a \to ab), (G_s, b), (\{P_i\}_i, r_*(a, b))).$$

To finish the simulation of reduce0 we have to iterate R_s corresponding to the sequence of strides for this case, i.e. we consider the sequence

$$(R_s \mid s = 1, s <= n, s = 2 * s).$$

For *sequential addressing* the communication graph inside a block is

$$G'_s = \{(P_{i+s}, P_i) \mid i = 0, 1, \cdots, s - 1\}.$$

For the entire block, the sequence of rules for stride s will be

$$R'_s = ((\{P_{i+s}\}_i, a \to ab), (G'_s, b), (\{P_i\}_i, r_*(a, b))).$$

To finish the simulation we iterate R'_s corresponding to the sequence of strides for this case, i.e. we consider the sequence

$$(R'_s \mid s = n = 2^t, s > 0, s = s \text{ div } 2).$$

We can analogously simulate the remaining versions of the procedure reduce of section 2. We illustrate with the improvement *first add during load*.

In this case we halve the number of blocks, and thus of threads. Equivalently, we can consider that M_g contains a double number of integers, codified with a number of symbols doubled compared to the number of threads. We denote a_{ij} and b_{ij} the symbols which will correspond to the thread P_{ij}. The previous loading sequence will be replaced by the load-and-add sequence

$$(\{(M_g, P_{ij}, a_{ij}, b_{ij})\}_{ij}, \{(P_{ij}, r_*(a_{ij}, b_{ij}), a_{ij} \to a)\}_{ij}).$$

References

1. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Simulation of P systems with Active Membranes on CUDA, *Briefings in Bioinformatics*, **11**, 3 (2010), 313–322.
2. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Simulating a P system based efficient solution to SAT by using GPUs, *Journal of Logic and Algebraic Programming*, **79**, 6 (2010), 317–325.
3. R. Ceterchi, C. Martín–Vide: Dynamic P Systems. In *"Membrane Computing", International Workshop, WMC-CdeA 2002, Curtea de Argeş, Romania, August 2002, Revised Papers*, (G. Păun, G. Rozenberg, A. Salomaa, C. Zandron eds.), LNCS 2597, Springer 2003, p. 146-186
4. R. Ceterchi, M.J. Pérez Jiménez: Simulating Shuffle–Exchange Networks with P Systems. In *Proceedings of the Second Brainstorming Week on Membrane Computing* (Gh. Păun, A. Riscos, F. Sancho and A. Romero, eds.), Report RGNC 01/04, University of Seville 2004, 117-129.
5. R. Ceterchi, M.J. Pérez Jiménez: A Perfect Shuffle Algorithm for Reduction Processes and its Simulation with P Systems. In *Proceedings of the International Conference on Computers and Communications ICCC 2004* (I. Dzitac, T. Maghiar, C. Popescu, eds.), Editura Univ. Oradea, 2004, 92-97
6. R. Ceterchi, M.J. Pérez Jiménez: On Two-Dimensional Mesh Networks and Their Simulation with P Systems. In *Membrane Computing, 5th International Workshop, WMC 2004, Revised Selected and Invited Papers* (G. Mauri, Gh. Păun, M. J. Pérez–Jiménez, G. Rozenberg, A. Salomaa, eds.), LNCS 3365 (2005), 259-277.
7. R. Ceterchi, M.J. Pérez Jiménez: On Simulating a Class of Parallel Architectures, *International Journal of Foundations of Computer Science, Vol. 17, No. 1 (2006)* 91–110
8. M. Harris: Optimizing Parallel Reduction in CUDA, NVIDIA Developer Technology (2007).
9. M. Harris. Mapping computational concepts to GPUs, *ACM SIGGRAPH 2005 Courses*, NY (USA), 2005.

10. D. Kirk, W. Hwu. *Programming Massively Parallel Processors: A Hands On Approach*, MA (USA), 2010.
11. M.A. Martínez-del-Amor, J. Pérez-Carrasco, M.J. Pérez-Jiménez. Characterizing the parallel simulation of P systems on the GPU. *International Journal of Unconventional Computing*, **9**, 5-6 (2013), 405-424.
12. M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez. Population Dynamics P systems on CUDA. In *10th Conference on Computational Methods in Systems Biology, CMSB2012*, (D. Gilbert, M. Heiner, eds.), LNBI 7605 (2012), 247-266.
13. Gh. Păun: Computing with Membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143, and *Turku Center for CS-TUCS Report* No. 208, 1998
14. Gh. Păun: *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, 2002.
15. *NVIDIA CUDA website*, 2014. `https://developer.nvidia.com/cuda-zone`