GPU powered ROSA Analyzer

Raúl Pardo*†, Fernando L. Pelayo† and Pedro Valero Lara‡
*Department of Computer Science and Engineering
Chalmers University of Technology, Gothenburg, Sweden
Email: pardo@chalmers.se
†Dept. de Sistemas Informáticos
E. Superior de Ingenieria Informática de Albacete
Universidad de Castilla - La Mancha
Campus Universitario. 02071-Albacete, Spain
Email: RaulPardo@dsi.uclm.es, FernandoL.Pelayo@uclm.es
‡CIEMAT, Unidad de Modelización
y Simulación Numérica, Madrid, Spain.
Departamento de Arquitectura de Computadores,
Facultad de Informática, Universidad Complutense
Email: Pedro.Valero@ciemat.es

***Abstract*—In this work we present the first version of ROSAA, Rosa Analyzer, using a GPU architecture. ROSA is a Markovian Process Algebra [11] able to capture pure non-determinism, probabilities and timed actions; Over it, a tool has been developed for getting closer to a fully automatic process of analyzing the behaviour of a system specified as a process of ROSA, so that, ROSAA [10] is able to automatically generate the part of the Labeled Transition System (occasionally the whole one), LTS in the sequel, in which we could be interested, but, since this is a very computationally expensive task, a GPU powered version of ROSAA which includes parallel processing capabilities, has been created to better deal with such generating process.**

As the conventional GPU processing loads are mainly focused on data parallelization over quite similar types of data, this work means a quite novel use of these kind of architectures, moreover the authors do not know any other formal model tool running over GPUs.

ROSAA running starts with the Syntactic analysis so generating a layered structure suitable to, afterwards, apply the Operational Semantics transition rules in the easiest way. Since from each specification/state more than one rule could be applied, this is the key point at which GPU should provide its benefits, i.e., allowing to generate all the new states reachable in a single-semantics-step from a given one, at the same time through a simultaneous launching of a set of threads over the GPU platform.

Although this establishes a step forward to the practical usefulness of such type of tools, the state-explosion problem arises indeed, so we are aware that reducing the size of the LTS will be sooner or later required, in this line the authors are working on an heuristics to properly prune an enough number of branches of the LTS, so making the task of generating it, more tractable.

I. Introduction

Formal methods are more used as Computer Science becomes a more mature science; this happens due to the fact that formal methods provide software designers with a way to guarantee high security and reliability levels for their products, and what is more, formal methods would allow to find software errors in the earliest stage of the software development life cycle so making it significatively cheaper. The main problem is that for systems having a size or a complexity level quite big, the analysis could be difficult to be developed, mainly because of two reasons, the first comes from the size of the graphical model generated, the second is because the most times these analysis are made by hand; we begin with the latter so presenting a tool able to apply automatically the operational semantics of the Process Algebra, ROSA. This evolution is required by the size of the LTS which makes the problem of generating it, very computationally expensive, so that we have assumed the need of a parallel code of the tool; for the sake of giving the most high expectations to its performance, we have decided to implement a parallel GPU powered version of ROSAA that is here being presented.

The existence of formal models tools is more than twenty years old, e.g., UPPAAL tool [1] gives to designers a good environment to use timed automata for such task; it also provides a framework having some analyzing skills. TINA is a frequently used tool [2] in order to use Petri Nets for such sort of work. Several tools have been developed for the sake of applying Process Algebras, the most known could be PEPA Workbench [6] which was developed over PEPA process algebra [7]. ROSA is a Markovian Process Algebra [11] which also has a tool [10]. However, the main problem of process algebra based tools is to deal with the *state explosion* arisen in the labeled Transition System (LTS), because of this, some of the process algebra research lines are searching the way to reduce the size of the LTS to be generated, and so to avoid this problem as much as possible [12], anyway the problem of the computatinal cost of generating LTSs is still there, and we have decided to go for it mainly by designing a parallel running version of the tool of ROSA and, what is completely new, implementing it over a GPU platform.

Due to increased needs to reach a more computational capacity and the incursion of environments GRID, CLOUD, virtualization, ..., along with the limitations of current CMOS technology and the excessive power consumption reached by current platforms, it is necessary a global rethinking of

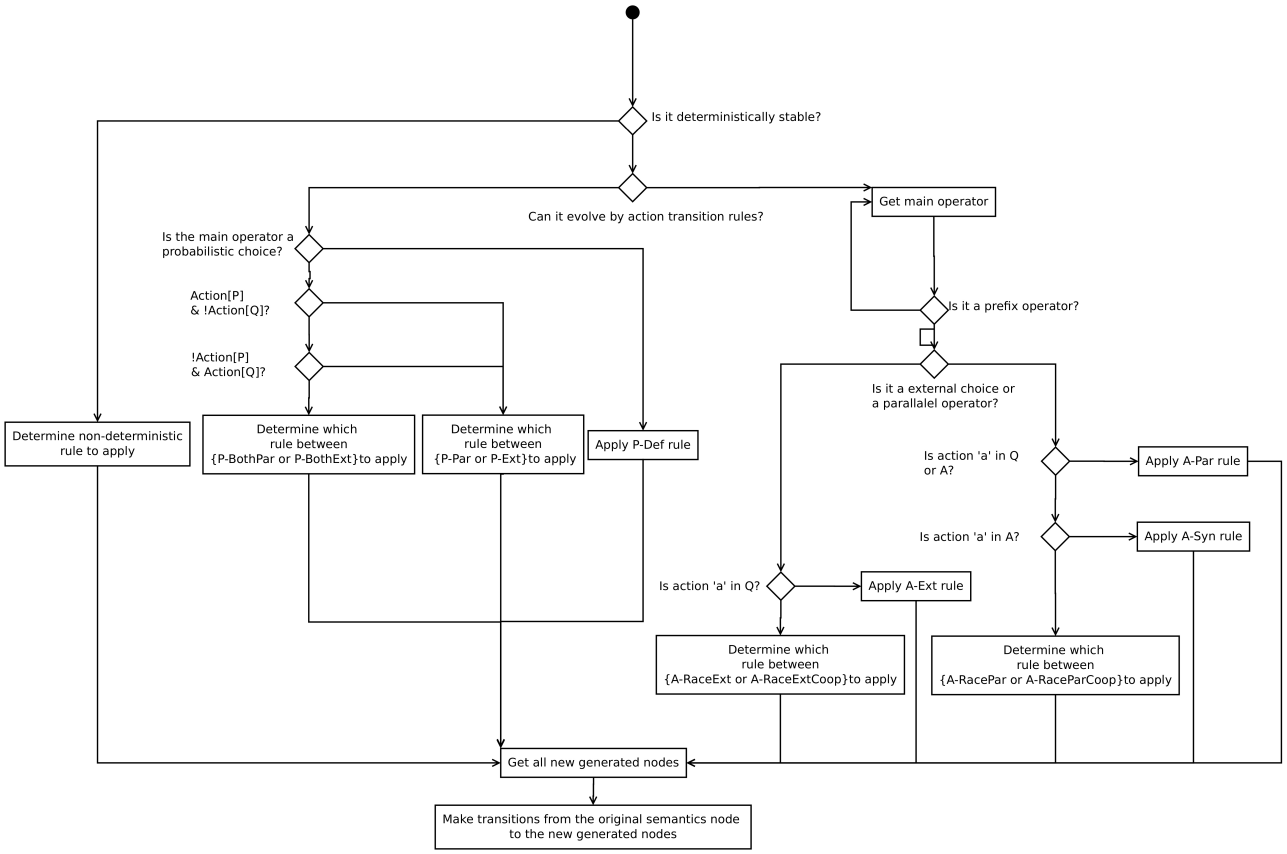


Fig. 1. Activity diagram for Semantic nodes analysis process

software and hardware design. In this context the current GPUs (many-core architectures) are one of the new massively parallel platforms. The main feature of these devices is a large number of processing elements integrated into a single chip, which reduces significantly the cache memory. These processing elements can access to a local high-speed external DRAM memory, connected to the computer through a high-speed I/O interface (PCI-Express). Overall, these devices can offer a higher main memory bandwidth and can use data parallelism to achieve higher floating point throughput than CPUs [4].

Fig. 2 describes the architecture of modern NVIDIA's GPUs. It consists of a number of multiprocessors and each multiprocessor has a set of simple cores. All multiprocessors share the same main memory, called "global memory". In addition, all cores of one multiprocessor can access the same "shared memory". Moreover, they offer the highest ratio performance/cost and an efficient design in terms of power-performance [5].

In this paper we present the GPU powered ROSAA, a tool able to build the LTS of a given ROSA process, by means of applying those operational Semantic rules which must be used for that input process; In particular, at any level of the LTS, the generating procedure of the next (level below it) is computed at the very same time in a single GPU platform, so showing

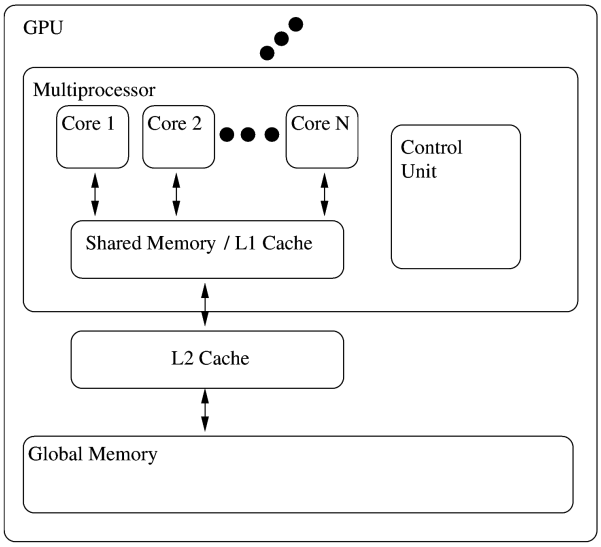


Fig. 2. GPU architecture

an unusual field for these architectures, since, first it is not the classical data parallelization, and second, the structure of the code to be launched simultaneously can perform quite different traces according to the type of the "node" to be computed.

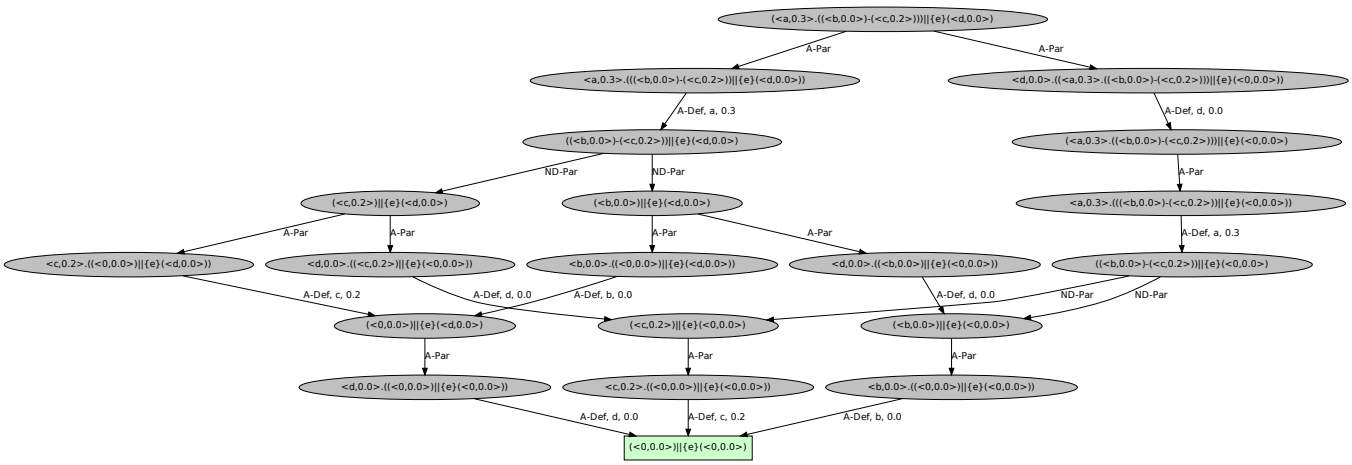


Fig. 3. Semantic Tree Example

This paper is structured as follows, section 2 provides a brief description of ROSA analyzer way of working; Section 3 describes the GPU powered scenario supporting this implementation and the key issues of its running over it. In order to show its skills, section 4 provides a case study, and to finish with conclusions and future work are stated in the last section.

II. ROSA ANALYZER TOOL

ROSA analyzer, ROSAA, was formerly developed in JAVA, this choice was taken looking for optimizing the handling with some complex data structures, as both the syntactic and semantics trees supporting the layered syntactic structure of ROSA specifications, the first; and the LTS, the latter.

A. Syntactic analysis

ROSA analyzer takes as INPUT a ROSA process, i.e., a system specified according to ROSA syntax, but this is quite far to be the optimum description in order to check the conditions to apply the rules of the operational semantics of ROSA.

Let us consider as example the ROSA process
$< a, 0.3 > .(< b, \infty > \oplus < c, 2 >)||_{\{e\}} < d, \infty >$.

According to the procedure captured in fig. 1 it is required first to check whether the process is deterministic, but the presence of $\oplus$ does not give right away a FALSE to the function *DeterministicStability*, instead, it is needed to check if such a $\oplus$ is available immediately or not, which leads us to analyze "when" this $\oplus$ will be enabled, which operators will be available then, and so on.

All the considerations to be followed, lead us to consider the operators priorities map that is easily and thoroughly captured in its syntactic tree, see fig. 4, where the higher syntactical priority the elements/operators have, the closer to the top they are located. Moreover this structure is not only optimized to check the general conditions of the group of rules to be applied, but also both to check the upper side of the operational semantics rules, and, to do the necessary modifications to the process which is being analyzed in order to generate (tentative) new nodes.

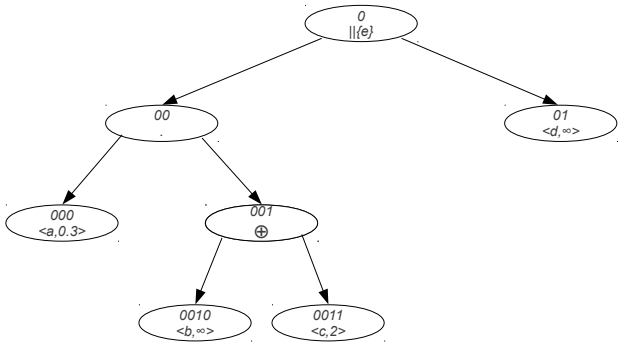


Fig. 4. Syntactic Tree Example

B. Semantic analysis

Once we have defined a proper structure for the process' syntax, it is time to apply the rules of the operational semantics of ROSA so as to generate its corresponding LTS. As usual this LTS is formed of arcs (representing transitions), and, nodes (representing states) which are themselves ROSA processes represented as stated in previous subsection, therefore in each Semantic tree node will appear a Syntactic tree.

Fig. 3 shows the LTS that ROSA associates to the process
$< a, 0.3 > .(< b, \infty > \oplus < c, 2 >)||_{\{e\}} < d, \infty >$

When the root node has to be analyzed in order to apply the proper semantic rule, the first question to answer is whether this process is deterministically stable, the answer is YES (*DeterministicStability* function outputs TRUE), then it is needed to check if the process can only evolve by the execution of any action, the answer is YES (*Action* function outputs TRUE), which means that the tentative rules to be applied belong to the set of Action Transition Rules, table I.

(A-Def)	$\frac{}{a.P \xrightarrow{a,\infty} P}$	$\frac{}{\langle a,\lambda\rangle.P \xrightarrow{a,\lambda} P}$
(A-Ext)	$\frac{P \xrightarrow{a,\lambda} P' \wedge a \notin \mathbf{Type}[\mathbf{Available}[Q]]}{P + Q \xrightarrow{a,\lambda} P'}$	$\frac{Q \xrightarrow{a,\lambda} Q' \wedge a \notin \mathbf{Type}[\mathbf{Available}[P]]}{P + Q \xrightarrow{a,\lambda} Q'}$
(A-Par)	$\frac{P \xrightarrow{a,\lambda} P' \wedge a \notin \mathbf{Type}[\mathbf{Available}[Q]] \cup A}{P\|\|_A Q \xrightarrow{a,\lambda} P'\|\|_A Q}$	$\frac{Q \xrightarrow{a,\lambda} Q' \wedge a \notin \mathbf{Type}[\mathbf{Available}[P]] \cup A}{P\|\|_A Q \xrightarrow{a,\lambda} P\|\|_A Q'}$
(A-RaceExt)	$\frac{P \xrightarrow{a,\infty} P' \wedge Q \xrightarrow{a,\lambda} Q' \wedge \lambda \neq \infty}{P + Q \xrightarrow{a,\infty} P'}$	$\frac{P \xrightarrow{a,\lambda} P' \wedge Q \xrightarrow{a,\infty} Q' \wedge \lambda \neq \infty}{P + Q \xrightarrow{a,\infty} Q'}$
(A-RaceExtCoop)	$\frac{P \xrightarrow{a,\lambda_1} P' \wedge Q \xrightarrow{a,\lambda_2} Q' \wedge (\lambda_1 = \infty = \lambda_2 \vee \lambda_1 \neq \infty \neq \lambda_2)}{P + Q \xrightarrow{a,\lambda_1+\lambda_2} P' \oplus Q'}$	
(A-RacePar)	$\frac{P \xrightarrow{a,\infty} P' \wedge Q \xrightarrow{a,\lambda} Q' \wedge \lambda \neq \infty \wedge a \notin A}{P\|\|_A Q \xrightarrow{a,\infty} P'\|\|_A Q}$	$\frac{P \xrightarrow{a,\lambda} P' \wedge Q \xrightarrow{a,\infty} Q' \wedge \lambda \neq \infty \wedge a \notin A}{P\|\|_A Q \xrightarrow{a,\infty} P\|\|_A Q'}$
(A-RaceParCoop)	$\frac{P \xrightarrow{a,\lambda_1} P' \wedge Q \xrightarrow{a,\lambda_2} Q' \wedge (\lambda_1 = \infty = \lambda_2 \vee \lambda_1 \neq \infty \neq \lambda_2) \wedge a \notin A}{P\|\|_A Q \xrightarrow{a,\lambda_1+\lambda_2} P'\|\|_A Q \oplus P\|\|_A Q'}$	
(A-Syn)	$\frac{P \xrightarrow{a,\lambda_1} P' \wedge Q \xrightarrow{a,\lambda_2} Q' \wedge a \in A}{P\|\|_A Q \xrightarrow{a,\mathbf{min}[\{\lambda_1,\lambda_2\}]} P'\|\|_A Q'}$	

TABLE I
ACTION TRANSITION RULES

As the root node is a parallel operator, the set of, so far tentative rules is formed of A-Par, A-RacePar, A-RaceParCoop and A-Syn; Once checked the conditions it is immediate that the only rule to be applied is A-Par, so generating these two new nodes:

- $(< b, \infty > \oplus < c, 2 >)||_{\{e\}} < d, \infty >$ after executing $< a, 0.3 >$
- $< a, 0.3 > .(< b, \infty > \oplus < c, 2 >)||_{\{e\}} 0$ after executing $< d, \infty >$

This procedure has to be repeated over these two new nodes, so, easily there could appear 4 nodes in the next (upside-down) layer of the LTS and so on.

As it is well known, the number of nodes to be processed could be exponential, therefore, even when dealing with not very big examples the need for more computational power for the sake of making these formal models tools as much usable as possible, is out of doubt.

III. GPU POWERED ROSAA

Our test bed is a "FERMI NVIDIA GPU" architecture.

Table II shows its main characteristics in terms of on-chip memory hierarchy and architecture.

This architecture has been controlled by means of the high level programming language CUDA [9] introduced by NVIDIA. Calculations in CUDA are distributed into a mesh or grid of CUDA blocks of the same size (number of threads). These threads run the GPU code, known as kernel; note that although this kernel is originally called by the CPU, it is executed in the GPU, as seen in fig. 5. Each CUDA block is executed in a single multiprocessor.

GPU Architecture	FERMI (C2070)
Multiprocessors	14
Cores	448
Memory	6 GB
Memory Bandwidth	177.4 GB/s
Max. threads blocks	65535

TABLE II
FERMI CHARACTERISTICS

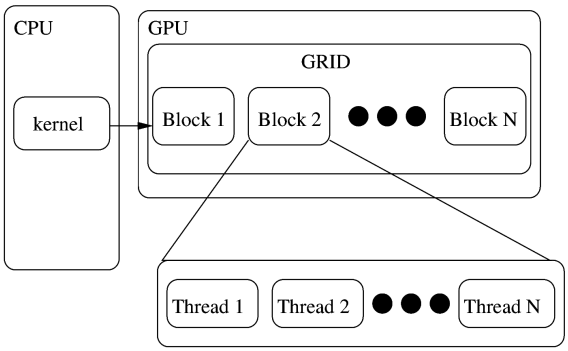


Fig. 5. Grid of CUDA blocks

Every CUDA code is divided into two different parts, CPU code and GPU code. The CPU code, provides the instructions to be performed by the CPU, e.g. allocating data on CPU and GPU, transferring data between GPU and CPU and launching kernels. On the other hand, the GPU code (kernel) provides the instructions to be executed in the GPU.

A. Multi-Kernel

Multi-kernel is defined as the capability to execute more than one kernel on the same GPU at the same time. Recently, this new feature has been included into CUDA, and, is available over FERMI or more recent architectures of NVDIA. Nevertheless, we have used "our" alternative approach [14], compatible with all NVIDIA GPUs, which lacks of these classical constraints of CUDA approaches.

Algorithm 1 CUDA Examples.

```
    Function kernel1(A,B)                                   ▷ GPU Code
 1: i = index of thread
 2: B[i] = A[i] + 100
    Function kernel2(C,D)                                   ▷ GPU Code
 3: i = index of thread
 4: D[i] = C[i] × D[i]
    Function OneByOne                                       ▷ CPU Code
 5: CPUMemAllocate(A_CPU,B_CPU)
 6: GPUMemAllocate(A_GPU,B_GPU)
 7: CPUMemAllocate(C_CPU,D_CPU)
 8: GPUMemAllocate(C_GPU,D_GPU)
 9: CPU->GPUMemTransfer(A_CPU,A_GPU)
10: CPU->GPUMemTransfer(C_CPU,C_GPU)
11: CPU->GPUMemTransfer(D_CPU,D_GPU)
12: Kernel1<NUMTHREADS1>(A_GPU,B_GPU)
13: Kernel2<NUMTHREADS2>(C_GPU,D_GPU)
14: GPU->CPUMemTransfer(D_GPU,D_CPU)
15: GPU->CPUMemTransfer(B_GPU,B_CPU)
```

Fig. 6. CUDA Example

First of all, in order to show the changes required to carry out the execution of this multi-kernel fashion, the example of fig. 6 describes the normal use of current many-core architecture. The pseudocodes are divided into CPU code (code executed by CPU), and GPU code (code executed by GPU).

Algorithm 2 CUDA method.

```
    Function CUDA Method                                    ▷ CPU Code
 1: CUDAStream Stream[2]
 2: CPUMemAllocate(A_CPU,B_CPU)
 3: CPUMemAllocate(C_CPU,D_CPU)
 4: GPUMemAllocate(A_GPU,B_GPU
 5: GPUMemAllocate(C_GPU,D_GPU)
 6: CPU->GPUMemTransfer(A_CPU,A_GPU)
 7: CPU->GPUMemTransfer(C_CPU,C_GPU)
 8: CPU->GPUMemTransfer(D_CPU,D_GPU)
 9: for i = 1 → 2 do
10:     StreamCreate(Stream[i])
11: end for
12: Kernel1<NUMTHREADS1>(A_GPU,B_GPU,Stream[1])
13: Kernel2<NUMTHREADS2>(C_GPU,D_GPU,Stream[2])
14: for i = 1 → 2 do
15:     StreamDestroy(Stream[i])
16: end for
17: GPU->CPUMemTransfer(B_GPU,B_CPU)
18: GPU->CPUMemTransfer(D_GPU,D_CPU)
```

Fig. 7. CUDA Method

CUDA approach consists in using a special data-type called "stream". Each kernel needs its own "stream", which has to be included as parameter in the kernels. The kernels with such stream associated are executed at the same time. This approach allows to execute up to 16 different kernels. A pseudocode of this approach is shown in fig 7, in which both aformentioned kernels are used.

In the alternative approach a different way to manage multi-kernel's execution is carried out. The kernels with such associated stream are executed at the same time. This approach allows to execute up to 16 different kernels. A piece of pseudocode is illustrated in fig. 7, in which both aforedmentioned kernels are used. In this approach, it is not necessary to use a special data-type and there is no limit on the maximum number of kernels. The kernels are mapped on one or on a set of threads blocks. Thus, different kernels are executed at the very same time and independently. In this case, the number of launched threads equals the addition of all threads, and, all variables have to be included as parameters in the same call. The pseudocode shown in fig. 8 shows the way of using this alternative method.

Algorithm 3 Alternative method.

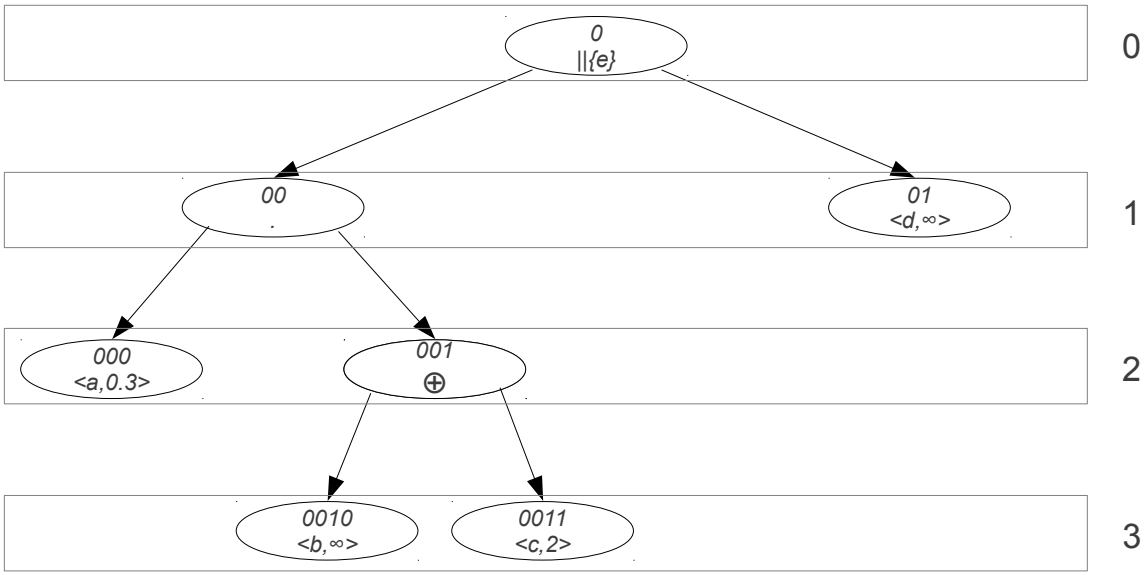
```
    Function Alternative Method                             ▷ CPU Code
 1: CPUMemAllocate(A_CPU,B_CPU)
 2: CPUMemAllocate(C_CPU,D_CPU)
 3: GPUMemAllocate(A_GPU,B_GPU)
 4: GPUMemAllocate(C_GPU,D_GPU)
 5: CPU->GPUMemTransfer(A_CPU,A_GPU)
 6: CPU->GPUMemTransfer(C_CPU,C_GPU)
 7: CPU->GPUMemTransfer(D_CPU,D_GPU)
 8: Kernel<NUMTHREADS1+NUMTHREADS2>
 9: (A_GPU,B_GPU,C_GPU,D_GPU)
10: GPU->CPUMemTransfer(B_GPU,B_CPU)
11: GPU->CPUMemTransfer(D_GPU,D_CPU)
    Function kernel(A,B,C,D)                                ▷ GPU Code
12: i = index of thread
13: j = index of block
14: if j = 0 then                                           ▷ kernel1
15:     B[i] = A[i] + 100
16: else if j = 1 then                                      ▷ kernel2
17:     D[i] = C[i] × D[i]
18: end if
```

Fig. 8. Alternative Method

IV. New data structures and new software design

Although theoretically the GPU implementation might seem easy, in practice it is particularly hard since GPUs have a huge amount of restrictions. One of the most demanding is that, GPUs do not support any dynamical memory handling. It prevents handling data structures by means of its classical algorithms, which consists in using pointers to allocate and to access nodes dynamically [13]. Despite the fact dynamical handling of data structures is a good way to deal with arborescent structures, our aim to achieve a GPU implementation has forced us to develop a new way to deal with binary trees and with the n-ary trees responsible for storing the generated LTS.

The aforementioned restriction requires to set, beforehand, a static memory size for each data structure. Taking into account dynamical memory handling comes from syntactic trees, we have mapped syntactic trees into arrays. Arrays provide us a static memory handling, and what is more, they are a very appropriate structure for GPU computing. In order to be able to map the syntactic tree into an array, we defined an indexation function which provides us the position in which



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	00	01	000	001	010	011	0000	0001	0010	0011	0100	0101	0110	0111
||{e}	.	<d,∞>	<a,0.3>	⊕					<b,∞>	<c,2>				

Fig. 9. Syntactic tree mapped into array

nodes of the syntactic tree should be stored depending on its binary identifier. This indexation function reserves consecutive positions for nodes allocated at the same level within the syntactic tree.

Any node has to be stored in the following position of the array (taking n as the tree level in which the node is located -root node has level 0-):

$Pos = 2^n + 1 + VALUE(Node.Key)$

Fig. 9 illustrates how the previous process

$< a, 0.3 > .(< b, \infty > \oplus < c, 2 >)||_{\{e\}} < d, \infty >$

is mapped into an array.

We would like to point out that (assuming that several LTS nodes can be executed simultaneously over this implementation) since the memory requirements of one LTS node is 2 MB, it would be possible to carry out up to 3000 LTS nodes at the same time in only one GPU call (kernel) by using our alternative method described in Section III-A.

In this first GPU approach, it is exploited a coarse grain parallelism, in which the solver of each LTS is mapped on one block of threads. Although, it is possible to use several threads per block, we use one thread per block (per LTS), fig. 10, since the generation of each LTS node is mainly sequential. In this way, it is exploited the parallelism inter-LTS, therefore the potential parallelism of the GPU is efficiently used so allowing to generate several LTSs on different multiprocessors, since each one can execute its own instructions flow (LTS) independently, on the same GPU device.

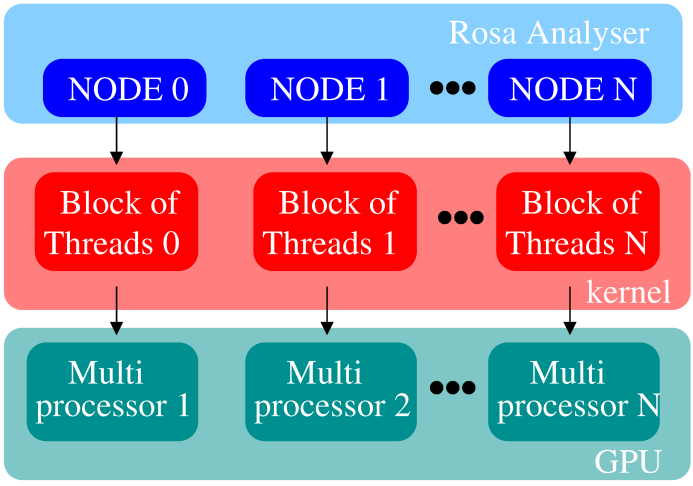


Fig. 10. Architecture scheduling

V. Case study: Simple Multi-Server Multi-Queue System

Communication systems have been extensively used in order to show how formal methods tools work [3], [6]. In particular, we have modelled a Multi-Server Multi-Queue (MSMQ) system which consists of a set of servers which circulates among different nodes providing services [6], [8].

Concretely, in our model we have considered two servers and two nodes. These nodes have a customers petitions queue, which will be able to store just a single petition. On the other hand, servers must search into nodes queues, in order to find requests to be served. It follows a *release-by-resource* mechanism, which means that servers will be continuously checking whether are new petitions in nodes queues or not; and, while they are working on a petition from a node, this node can not be accessed by any other server.

In previous models of these kind of systems, nodes queues checking was carried out in a sequential way, however, we assume that we do not have any order to check nodes. This characteristic has been used, in order to show how ROSA process algebra (and consequently ROSAA) is also able to deal with non-deterministic behaviours.

A. ROSA Specification

ROSA process algebra specification for each component of MSMQ system has been defined as follows:

$$Node_j =< in, \lambda > .available. < serv, \mu > .Node_j + \\ + empty.Node_j$$

$$Server_j = available. < serv, \mu > .Server_j + \\ + empty.Server_j$$

$$MSMQ = (Node_1||Node_2)||_{\{available,serv,empty\}} \\ ||_{\{available,serv,empty\}}(Server_1||Server_2)$$

As expected, $Server$ process represents servers behaviours. There are two possibilities by which a $Server$ process can behaves: either it engages a node which has a petition to process (action $available$), or the server checks a node which has an empty queue (action $empty$). In the former, once both node and server are engaged, they complete their work (action $serv$) taking a time modelled by a negative exponential random variable of parameter μ. The latter represents that the node queue is empty, then they do not perform any action. Anyway, once described actions have been finished, the server restarts its initial behaviour.

$Node$ process represents nodes behaviours, where a node can either get a customer petition which is defined with action in, or being empty. If the node is empty, as server behaves, it does not execute any action, which allows the server to search into another nodes. Again, when there is a customer petition into the node queue, it synchronizes by means of $available$ action, and finally serves the required petition, which is represented by $serv$ action.

It is important to point out that, since this version of ROSAA is not able to deal with that performance analysis defined in ROSA; some parameters as λ and μ have not been taken into account, because we are focused on functional behaviours and from the computational expensive cost point of view it does not matter. On the other hand, the restrictions demand by input syntax of ROSAA (which was defined for ROSA analyzer [10]), force us to rewrite MSMQ systems as follows:

$$S_j = a.s.S_j + n.S_j$$

$$N_j = i.a.s.N_j + n.N_j$$

$$MSMQ = (N_1||\{\}N_2)||\{a, s, n\}(S_1||\{\}S_2)$$

Due to the large size of the generated LTS, which is composed by 736 nodes and 1238 transitions, it is not appropriate to be shown in this format, so a detailed version of it is free available at:

http://raulpardo.files.wordpress.com/2012/12/msmq_lts.pdf

From the size of this LTS and the width of each level, it is easy to see how this parallel design could help us to improve ROSA analyzer performance during the LTS building process.

VI. Conclusions and future work

To sum up we can say that aimed for the wish of making formal models as usable as possible, and dealing with the states explosion problem when generating the LTS associated to a process, we have implemented a GPU powered tool able to apply a formalism as ROSA, a Markovian Process Algebra that captures non-determinism, probabilities and timed actions. This represents an unconventional and very novel use of GPUs platforms that deserves to be analyzed from this perspective rather than for the pure performances (speed-up) one, since it is our first prototype in this, hopefully, long way.

An almost classical case study has been presented, for the sake of illustrating both the real need of this improvements over Formal Models Tools (quite width LTS associated) and the capabilities of this GPU powered ROSAA.

Although we have reached a more practical and computationally fast use of ROSA process algebra, the *state explosion* problem has not been solved yet; as it is an exponential hard problem, the need of making it cheaper tractable still exists, so we keep looking for an heuristic that leads the searching for a given node through as few states as possible.

Acknowledgements

This work has been partially supported by projects CGL2010-20787-C02-02 & SyeC-CSD2007-00050

References

[1] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas Henzinger, and Eduardo Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0020949.

[2] B. Berthomieu *, P.-O. Ribet, and F. Vernadat. The tool tina - construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[3] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, January 1993.

[4] Wu-chun Feng and Dinesh Manocha. High-performance computing using accelerators. *Parallel Comput.*, 33(10-11):645–647, nov 2007.

[5] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, may 2005.

[6] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.

[7] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[8] M. Ajmone Marsan, S. Donatelli, and F. Neri. Gspn models of markovian multiserver multiqueue systems. *Performance Evaluation*, 11(4):227 – 240, 1990.

[9] NVIDIA. Nvidia cuda compute unified device architecture-programming guide, 2012. Version 5, http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[10] Raúl Pardo and F. L. Pelayo. Rosa analyser: An automatized approach to analyse processes of rosa. In César Andrés and Luis Llana, editors, *Proceedings 2nd Workshop on Formal Methods in the Development of Software*, volume 86 of *Electronic Proceedings in Theoretical Computer Science*, pages 25–32. Open Publishing Association, August 2012.

[11] F. L. Pelayo. *Application of formal methods to performance evaluation*. PhD thesis, Universidad de Castilla - La Mancha, 2004.

[12] A. Stefanek, R.A. Hayden, and J.T. Bradley. Gpa - a tool for fluid scalability analysis of massively parallel systems. In *Quantitative Evaluation of Systems (QEST), 2011 Eighth International Conference on*, pages 147–148, sept. 2011.

[13] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford. *Introduction to algorithms*. MIT press, third edition, September 2009.

[14] Pedro Valero and Fernando L. Pelayo. Towards a more efficient use of gpus. In *ICCSA Workshops*, pages 3–9, 2011.