Simulating Large-Scale ENPS Models by Means of GPU

Manuel García–Quismondo[1], Ana Brânduşa Pavel[2], and Mario J. Pérez–Jiménez[1]

[1] Research Group on Natural Computing
Dpt. of Computer Science and Artificial Intelligence, University of Sevilla
Avda. Reina Mercedes s/n. 41012 Sevilla, Spain
E-mail:{mgarciaquismondo,marper}@us.es

[2] Department of Automatic Control and Systems Engineering, Politehnica University of Bucharest
Splaiul Independenţei, Nr. 313, sector 6, 090042, Bucharest, Romania
E-mail: apavel@ics.pub.ro

Summary. Enzymatic Numerical P Systems (ENPS), an extension of Numerical P Systems, have been successfully applied to model robot controllers. GPGPU is an innovative technological paradigm which applies the parallel architecture of graphic cards to solve parallel, general–purpose problems. In previous work, a GPU simulator for ENPS was introduced. In this paper, a performance analysis on the simulator is performed in order to experimentally measure the speed-up factors resulting from the simulations.

Keywords: Enzymatic Numerical P Systems, GPU, simulation

1 Enzymatic Numerical P Systems

Membrane computing is an interdisciplinary field which studies computational models inspired by the compartmental structure of biological cells. There exist many types of membrane systems [12], also known as P systems after mathematician Gh. Păun, who introduced them. Numerical P systems (NPS) are a type of P systems in which numerical variables evolve inside the compartments by means of programs; a program (or rule) is composed of a production function and a repartition protocol [11]. The variables have a given initial value and the production function is a multivariate polynomial. The value of the production function for the current values of the variables is distributed among variables in certain compartments according to a repartition protocol. A formal definition of NPS can be found in [11], where this type of P system is introduced with possible applications in economics.

Enzymatic numerical P systems (ENPS) represent an extension of NPS, proposed and used in the context of modeling robot controllers [13], [14]. ENPS is a more powerful modelling tool than NPS, as it is proven in several articles [13], [3], [17]. ENPS models allow the existence of more than one rule per membrane than NPS while keeping the deterministic behavior. By using a special type of variables referred as enzymes, ENPS models provide a selection mechanism of the active rules during the computational

process. Therefore, ENPS are a flexible and efficient modelling framework that can be successfully used for modeling robot behaviors like obstacle avoidance, localization, follower, etc. [13], [4]. An ENPS model of degree $m, m \geq 1$ is formally defined as follows:

$$\Pi = (H, \mu, (Var_1, E_1, Pr_1, Var_1(0)), \ldots, (Var_m, E_m, Pr_m, Var_m(0))) \tag{1}$$

where:

- H is an alphabet that contains m symbols (the labels of the membranes);
- μ is a membrane structure;
- Var_i is the set of variables from compartment i, and the initial values for these variables are $Var_i(0)$;
- E_i is a set of enzyme variables from compartment i, $E_i \subseteq Var_i$
- Pr_i is the set of programs (rules) from compartment i. Programs process variables and have two components; a production function and a repartition protocol. In ENPS models, programs can have one of the two following forms:

  1. Non-enzymatic form, which is exactly like the one from standard NPS:

  $$Pr_{j,i} = (F_{j,i}(x_{1,i}, \ldots, x_{k_i,i}), c_{j,1}|v_1 + ... + c_{j,n_i}|v_{n_i}) \tag{2}$$

  2. Enzymatic form

  $$Pr_{j,i} = (F_{j,i}(x_{1,i}, \ldots, x_{k_i,i}), e_{j,i}, c_{j,1}|v_1 + ... + c_{j,n_i}|v_{n_i}) \tag{3}$$

  where:
  - $F_{j,i}(x_{1,i}, ..., x_{k_i,i})$ is the production function;
  - $e_{j,i} \in E_i$, is the enzyme–like variable associated to $e_{j,i}$;
  - k_i represents the number of variables in membrane i;
  - $c_{j,1}|v_1 + \ldots + c_{j,n_i}|v_{n_i}$ is the repartition protocol;
  - n_i represents the number of variables contained in membrane i, plus the number of variables contained in the parent membrane of i, plus the number of variables contained in the children membranes of i.

In ENPS models, all active rules are executed in parallel on each computational step. A rule is always active if it is in the non-enzymatic form. Otherwise, if it is in the enzymatic form, a rule is active only if the associated enzyme–like variable has a greater value than the minimum of the absolute values of the variables involved in the production function. The repartition protocol works like in classical NPS [11]
Both NPS and ENPS models have been successfully used for modelling robot controllers [4], [13], [14]. The advantages of using ENPS for this kind of applications are pointed out in [3]. For testing the robot controllers, a Java implementation of a numerical P systems simulator was implemented and used. This Java simulator, *SimP*, simulates ENPS models. Since ENPS are an extension of NPS, SimP can be used to simulate classical NPS as well. SimP was proposed in [15] and it is available as a free executable version (for a free executable version of SimP, please contact `anabrandusa@gmail.com`). SimP allows the computation of rational production functions and not only polynomials. This is useful for implementing more complex models required for robotics applications. An example of model of a rational production function is presented in subsection 4.3.
In this paper, the performance of a parallel and distributed simulator which computes ENPS structures is analysed. The simulator was first proposed in [7]. In order to test the

parallel simulator and analyse its performance, a replication mechanism of membranes was used. For instance, in swarm robotics applications, the robots in a group may have similar behaviors which need to run in parallel. For example, each robot needs to run an obstacle avoidance behavior. Therefore, the ENPS structure for obstacle avoidance [13] must be executed in parallel for all the robots in the swarm (figure 1). Those applications in which each robot needs to run one or even more membrane controllers in parallel and all robots must work in parallel as well take real advantage of the parallel and distibuted ENPS simulator, which will be further discussed.

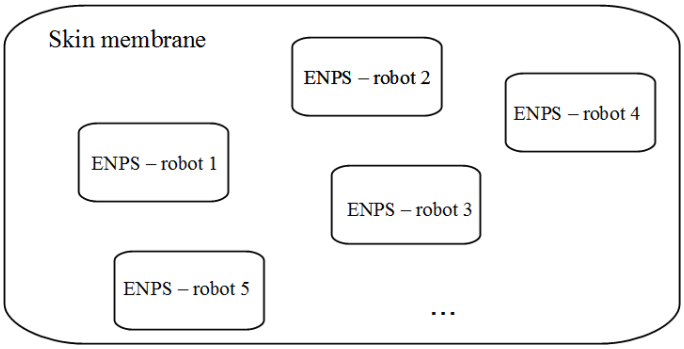

Fig. 1. Replication of ENPS models can be used for swarm robotics applications

2 An introduction to GPU Computing

Graphic cards, also known as Graphic Processing Units (*GPUs*) are devices whose main task is to solve image rendering problems. These problems are usually massive parallel problems, as they can commonly be reduced to render pixels and vertices in a parallel fashion. As a result, the industry has turned these devices into powerful highly–parallel computers with a large number of processors. However, they have been of limited use to scientist for quite a long time. The reason is that GPUs were only suitable for image–related problems, with little application in the scientific world out of image processing itself. Therefore, the amount of effort required to translate general–purpose problems into their graphical interpretations made scientists use other parallel architectures such as computer clusters and FPGAs [1] [9] [16].

However, NVIDIA changed this landscape by providing a toolkit for general–purpose GPU computing named Compute Unified Device Architecture (*CUDA*) [19]. This API permitted developers to solve scientific, non–graphical problems on GPUs. From then on, the adoption of GPU computing by the scientific community has gone widespread. As a result numerous scientific papers have shown moderate to impressive performance improvements on a GPU over a CPU [2].

2.1 The CUDA programming model

Nowadays, the number of processors in a GPU can reach up to 448 processor cores and 1.536 processing units per core, thus resulting in a total number of $448 \times 1.536 = 688.128$ processing units [19]. Thus, GPUs are structured as a grid of multiprocessor cores or *streaming multiprocessors*. Each one of these multiprocessors (figure 2) is composed of several simpler processing units known as *streaming processors*. This hierarchy allows programmers to structure their code in a distributed way, so they can couple processes with a high communication bandwidth with each other, thus clustering and allocating them within the same multiprocessor in the GPU. All processing units execute the same code at the same time, hence applying a computational paradigm known as Single Instruction Multiple Program (*SIMP*) [16]. CUDA provides an abstract model of GPU

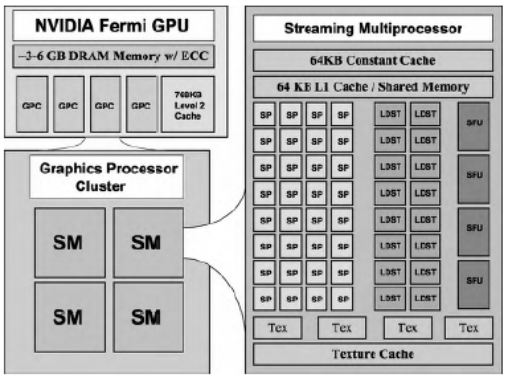

Fig. 2. Simplified hardware block diagram of an Nvidia Graphic Card

architecture. This model is known as the *CUDA programming model*. The idea is to make use of an abstraction of the specific graphic card on which the code runs, so as not to force the code depend on particular devices. This model is composed of a grid of elements known as *blocks*, which are abstractions of the streaming multiprocessors mentioned above. Each one of these blocks is composed of computing elements known as *threads* [5]. Thus, the CUDA programming model is a multidimensional three–levelled architecture , as the dimension of grids and blocks can be 1, 2 or 3, depending on the configuration options set by the developer. Figure 3 describes graphically this programming model.

2.2 Programming on CUDA–C

The firsts steps towards obtaining languages for general–purpose GPU computing involved wrapping currently existing languages for graphics processing with mainstream general–purpose languages, such as C and Fortran. However, these approaches were still hindered by the limitations of graphic cards on treating general–purpose data structures and data flows as pixels and vertices. However, the CUDA programming

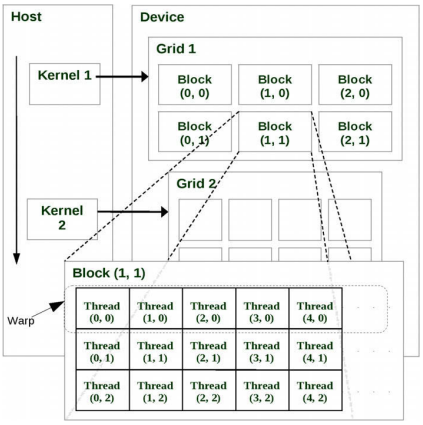

Fig. 3. The CUDA programming model

model, combined with more advanced graphic hardware with support for general–purpose parallel programming allowed to extend mainstream languages with specific primitives for GPU computing [16]. In this direction, Nvidia issued the first version of the *CUDA–C* language in 2006 [19]. This language was an extension of the C language with primitives and operations for general–purpose parallel problems [16].

A CUDA–C program is composed of two different parts: the *host* code and the *device* code. The host code is the part of the code executed on the CPU. This code contains calls to pieces of the device code, which is executed on the GPU. The device code is composed of CUDA–C functions which are executed on the GPU, known as *kernels*. The SIMP paradigm defines that each kernel is executed on all threads at the same time. When it comes to execution, threads are bundled in packages known as *warps*. Threads in the same warp communicate via a fast on–chip memory. However, the communication with threads in different warps is performed by using a slow off–chip memory. That is the reason why threads in the same warp should have a high communication bandwith with each other, as well as a low communication bandwith with threads out of the warp [5].

In practice, the CUDA programming model claims that each block is assigned a sequence of warps. Therefore, each block takes a warp from its sequence and executes it, assigning each warp thread to a different block thread [16]. The order of this sequence, as well as the block to which each warp is associated, is not controlled by the developer. This entails that the problem should not depend on the order in which the warps are executed, as this order cannot be guaranteed [5]. A more thorough description of the CUDA programming model can be found in [10].

3 A GPU Simulator for Enzymatic Numerical P Systems

3.1 A brief description of the simulator

In a previous work [7], a GPU simulator for Enzymatic Numerical P Systems was introduced. This simulator takes as input an XML file defining a description of an ENPS model and simulates it for a number of cycles defined. This number of cycles can be defined in the XML file or as an optional argument in the call sentence. The simulator does not need to check for errors in the input file. The reason is that the XML format accepted by the simulator is also accepted by SimP, which is a Java simulator for ENPS proposed in [15]. This way, errors in the specification of the ENPS model can be checked on SimP. Therefore, if a specification is regarded as error–free by SimP, then it can be given as input to the GPU simulator. This simulator will be published under GNU GPL version 3 license [8], and it is currently available by contacting the authors. In [7], some open problems were proposed. One of this problem had to do with the comprehensive evaluation of the performance of the simulator on large–scale models composed of a considerably high number of programs. In these cases, the overheads related to the workload distribution and the setup operations needed to run the simulations were supposed to be minimal, in comparison to the speed–up factor obtained from the parallel application of the programs in the model. This is not the case of models with a small numbers of programs. In these cases, the performance gain obtained as a result of the parallel application of programs is suffocated by the considerably time–consuming task of setting up the CUDA programming model elements [7].

3.2 Functioning of the GPU simulator

The functioning of the GPU simulator described in [7] consists of two stages. The first stage initializes the model to simulate. The second stage simulates a computational step of the model. This stage is repeated for each computational step simulated. These stages are described below:

Initialization stage: First, all operations concerning the setup of the GPU device itself are carried out. These operations include allocations in the GPU memory and transactions between the CPU memory and the GPU memory. Then, this stage normalizes the coefficients found in the repartition protocols in the input model. In practical terms, this normalization substitutes each coefficient $c_{l,s}$ by $\frac{c_{l,s}}{\sum_{j=0}^{k_{l,i}} c_{l,j}}$. This normalization can be performed on each simulation step, but performing it only once at the beginning of the simulation spares computational time.

Computation stage: This stage simulates checks the programs in the model and applicates the checked programs. It consists of four sub–steps, which are:

1. For each program, set its activation. That is, check whether this program is active, that is, is going to be applied on the current computational step.
2. Calculate the production function of the active programs.
3. Set to 0 the values consumed by the active programs, that is, the values of variables such that there is any active program which depends on its value.

4. Multiply the results of the active production functions by the normalized coefficients calculated on the initialization stage.
5. Add these results to the variables contributed by the active programs, according to their repartition protocols.

4 Performance analysis of the simulator

In order to carry out an analysis of the performance and runtimes obtained from the GPU simulator, we have developed a sequential simulator for Enzymatic Numerical P Systems in C language. This sequential simulator has been developed to compare simulation times obtained from a sequential, low–level simulator to those obtained from the GPU simulator. Nevertheless, it can also be used for the efficient simulation of Enzymatic Numerical P Systems in those environments in which no Nvidia card is available. Besides, this simulator takes as input a file which describes an ENPS in the same format that the GPU simulator and SimP. Therefore, the same files can be used for all three simulators, hence sparing time on translations between formats.
For a fair comparison between execution times, no memory allocation is performed after the setup stage. This feature is compulsory on the GPU simulator, because all computation steps are implemented by means of kernel calls and all memory in the GPU can only be allocated from the host code [19]. The importance of this feature rises from the fact that memory allocation in C is a time–consuming instruction. Therefore, if there were a significant number of memory allocations on each computational step the performance of the C sequential simulator would be severely hindered. This would result on an even higher speed–up factor due to a bad design of the sequential simulator, instead of a good design of its GPU counterpart.

4.1 Performance comparison with SimP

Apart from comparing the CUDA–C simulator with a C–based one, we have also compared the GPU simulation times with the ones obtained from SimP [15]. SimP is an ENPS simulator in Java language. Java programs are executed on a virtual machine which does as a middleware between the actual device and the software. This virtual machine is known as Java Virtual Machine (JVM) [18]. JVM ensures that Java programs can be executed on any device in which JVM is installed, thus guaranteeing complete compatiblity among different hardware architectures. However, this virtual machine approach comes at a cost. Firstly, the programmer loses control of the way in which the memory is managed. For instance, memory objects cannot be freed directly. Instead, an execution thread named *garbage collector* checks which objects are not referenced anymore in the program and frees the allocated memory. Secondly, the translatiosn from JVM instructions to assembly instructions are performed in runtime. Thus, an overhead in the execution time is produced as a result of these translations. All in all, the programmer cannot control directly the execution flow of Java programs. Therefore, in cases where efficiency is required, Java is not, in most cases, a true rival to low–level languages such as C.

4.2 Generation of input models

In [7], some problems about an extensive analysis of the simulator are discussed. One of them has to do with the fact that the existing ENPS models have too few programs for parallel simulations to pay off. The reason is that the setup operations computed at the beginning of parallel simulations take a long time, in comparison to the whole sequential computation runtime. In order to overcome this difficulty, we have taken some ENPS models as reference and replicated them several times. This way, we can have an on–demand number of programs per model. Therefore, the more times the seed models are replicated, the more programs will compose the resulting models. When the number of replications given as input is high enough, the GPU simulation does pay off in terms of execution time.
The algorithm used for performing the model replication takes the following inputs:

N: The number of copies to perform.
Π: The seed model to replicate:
$\Pi = (H, \mu, (Var_1, Pr_1, E_1, Var_1(0)) \dots (Var_m, Pr_m, E_m, Var_m(0)))$.
For these inputs, the algorithm takes the following steps:

1. Replicate Π a number of times N.
2. Associate an index $o, 1 \leq o \leq N$ to each of the copies of Π. As a result, a set Φ of ENPS models of degree m is obtained. Formally speaking, $\Phi = \{\Pi_o = (H_o, \mu_o, (Var_{1,o}, E_{1,o}, Pr_{1,o}, Var_{1,o}(0)) \dots (Var_{m,o}, E_{m,o}, Pr_{m,o}, Var_{m,o}(0)))\}$, $1 \leq o \leq N$, where:
   - $\forall o | 1 \leq o \leq N, H_o = \{\{1, o\} \dots \{m, o\}\}$
   - $\forall i, o | 1 \leq i \leq m, 1 \leq o \leq N, Var_{i,o} = \{x_{1,i,o} \dots x_{k_i,i,o}\}$
   - $\forall i, o | 1 \leq i \leq m, 1 \leq o \leq N, E_{i,o} \subseteq Var_{i,o}$ is the set of all enzyme–like variables associated to programs in $Pr_{i,o}$.
   - $\forall i, o | 1 \leq i \leq m, 1 \leq o \leq N, Pr_{i,o} = Pr_{1,i,o} \dots Pr_{q_i,i,o}$, where:
     - $Pr_{l,i,o} = (F_{l,i,o}(x_{1,i,o}, \dots, x_{k_i,i,o}) \rightarrow c_{l,1,o}|v_{o,1} + \dots + c_{l,n_i,o}|v_{o,n_i})$ if $Pr_{l,i}$ is in non–enzymatic form.
     - $Pr_{l,i,o} = (F_{l,i,o}(x_{1,i,o}, \dots, x_{k_i,i,o})(e_{l,i,o} \rightarrow) c_{l,1,o}|v_{o,1} + \dots + c_{l,n_i,o}|v_{o,n_i})$ if $Pr_{l,i}$ is in enzymatic form.
   - $\forall i, o | 1 \leq i \leq m, 1 \leq o \leq N, Var_{i,o}(0) = \{\lambda_{1,i,o} \dots \lambda_{k_i,i,o}\}$
3. $\forall i, o, j, l, (1 \leq i \leq m), (1 \leq o \leq N), (1 \leq j \leq k_i), (1 \leq l \leq q_i)$, assign a different random value in $\{1, \dots, 10\}$ to $\lambda_{j,i,o}, c_{l,n_i,o}$ and each one of the numerical constants in $F_{l,i,o}$.
4. Return a new ENPS model $\Pi_r = (H_r, \mu_r, (Var_{1,1}, E_{1,1}, Pr_{1,1}, Var_{1,1}(0)) \dots (Var_{m,N}, E_{m,N} Pr_{m,N}, Var_{m,N}(0)), (Var_{skin}, E_{skin}, Pr_{skin}, Var(0)_{skin}))$, where:
   - $H_r = \cup_{o=1}^{N} H_o \cup \{skin\}$
   - $\mu_r = [\mu_1 \dots \mu_N]_{skin}$
   - $Var_{skin} = \emptyset$
   - $E_{skin} = \emptyset$
   - $Pr_{skin} = \emptyset$
   - $Var(0)_{skin} = \emptyset$

The replication process has been performed by using Java [18]. For the purposes of parsing the seed model and writing the resulting models, an extension of P–Lingua [6]

has been developed. Specifically, a new input and a new output format has been included into the P–Lingua framework. The input format delegates the parsing process to SimP [15]. The output format generates an XML description of the model. This description is encoded on the common format accepted by all ENPS simulators (Java, C and CUDA–C). This extension proves the versatility of P–Lingua as a useful, assisting tool for a wide variety of Membrane Computing–related tasks; in our case; for analysing the performance of a GPU simulator.

4.3 Case study

In our simulator, two seed models have been replicated. After these replications, the resulting expanded models have been simulated. The first seed model is a dummy one with no particular purpose apart from this performance analysis. This model is an ENPS composed of 2 membranes: $\Pi_1 = (H, \mu, (Var_1, E_1, Pr_{1,1}, Var_1(0)), (Var_2, E_2, Pr_{1,2}, Var_2(0)))$, where:

- $H = 1, 2$
- $\mu = [[]_2]_1$
- $Var_1 = \{x_{1,1}, x_{2,1}, x_{3,1}\}$
- $E_1 = \{x_{3,1}\}$
- $Pr_{1,1} = \{3 \cdot x_{1,1}(x_{3,1} \rightarrow)2|x_{1,1} + 1|x_{2,1}\}$
- $Var_1(0) = \{1, 2, 3\}$,
- $Var_2 = \{x_{1,2}\}$
- $E_2 = \emptyset$
- $Pr_{1,2} = \{2 \cdot x_{1,2} \rightarrow 2|x_{1,2}\}$
- $Var_2(0) = \{1\}$

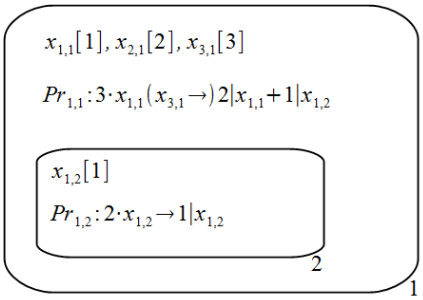

Fig. 4. Dummy ENPS used as seed for replication

On the other hand, the second seed model performs a function approximation. Mathematical functions like trigonometric functions, exponential functions, etc. are often used in control algorithms in robotics. Therefore, in the following example an ENPS model which computes e^x is presented. The proposed GPU simulator also allows computation of rational production functions as well as polynomial functions, which is an

important advantage for the modeling process of complex membrane systems. In order to approximate e^x, the following power series is used:

$$e^x \approx \sum_{n \geq 0} \frac{x^n}{n!} \tag{4}$$

The partial sum of this power series is the next sequence:

$$s_n = \sum_{k \geq 0}^{n} \frac{x^k}{k!} \tag{5}$$

Sequence s_n can be written in a recurrent form, as follows:

$$s_n = s_{n-1} + a_n \tag{6}$$

where a_n is:

$$a_n = \frac{x^n}{n!} \tag{7}$$

Sequence a_n can also be written in a recurrent form, by computing $\frac{a_n}{a_{n-1}}$ as follows:

$$a_n = \frac{x}{n} \cdot a_{n-1} \tag{8}$$

Formula 8 can be implemented as a rational production function, as shown in figure 5 (rule $Pr_{1,2}$).

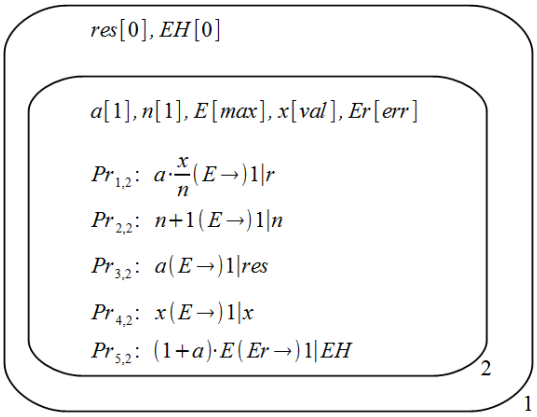

Fig. 5. An ENPS which computes e^x. This model has also been used as seed model for replication

As it is shown in figure 5, two membranes were used in order to approximate the exponential function e^x. The skin membrane (membrane 1) contains a non enzyme–like

variable res which represents the result of the computation and an enzyme–like variable, EH, which is a stop enzyme. Stop enzymes are used in order to test the stop condition of the computation. Therefore, the number of computational steps is different for different arguments of the function. The computation finishes when the value of the term added to the sum is lower than a given value, err.
The child membrane (membrane 2) is responsible for the approximation. It contains the following variables:

- a stores the next term of the a_n sequence and has an initial value equal to 1.
- n is a counter variable and has the initial value equal to 1.
- x represents the argument of the function e^x.
- E is an enzyme–like variable which controls the program flow. it allows the execution of the valid production functions and it is consumed when the computation finishes; the initial value of the enzyme is given as input max; max must be a value grater than the maximum possible value of x.
- Er is an enzyme–like variable used in the stop condition.

Er receives an input value, $err = 10^{-10}$; when the term of the series is lower than err, the computation stops.
Membrane 2 has five rules responsible for the following tasks:

- $Pr_{1,2}$ computes the next term in the series; if the value of E is greater than a, x or n, the rule is active.
- $Pr_{2,2}$ produces the incrementation of n.
- $Pr_{3,2}$ accumulates the terms in variable res, which will be the final result.
- $Pr_{4,2}$ copies the value of x, which was consumed and must be stored.
- $Pr_{5,2}$ stops the computation when $a < Er$.

The value of a is decreasing because the sequence a_n is convergent. When $Pr_{5,2}$ is activated, the stop enzyme EH, receives a positive value and E is consumed, so the other production functions become inactive. A condition outside the membrane system tests if the stop enzyme, EH, is greater than 0 and if that happens, the simulation stops.

4.4 Simulation results

For each seed, a total of 36 models have been simulated. The number of programs range from 1 to 9000. Each model has been run for 100 steps. Figure 11 displays the execution times obtained from the Java simulator included in SimP [15], the C simulator described in section 4 and the CUDA–C simulator from [7]. Figure 13 displays the speed–up factors obtained from the runtimes among the simulators. Figures 14 and 12 plot the same data, with the exception of the Java results. The reason is to display cleaner statistics on the execution times and speed–up factors obtained from the most efficient studied simulators. By examining the dummy model charts, one can observe that there is a great difference between the execution times from the SimP (Java) simulator and the C and CUDA–C simulators. Thus, a maximum speed–up factor of about 85x is reached on the comparison between the CUDA–C and Java simulators. However, the maximum speed–up factor obtained between the CUDA–C and Java simulators is only 6x. The maximum speed–up factor on the simulation of the e^x model is about 49x on the Java vs CUDA–C comparison and about 10x on the C vs CUDA–C comparison. The used graphic card is a domestic, commercial one. Thus, it is not designed for intensive parallel, computations. In contrast,

Tesla models contain more RAM memory and a larger number of processors, as they are specifically engineered for intensive High Performance Computing [19]. For instance, Nvidia Tesla C1060 graphic cards contain 240 streaming multiprocessors and 4GB RAM memory [19]. Thus, the speed–up factors obtained on one of these cards is expected to be higher than in the ones obtained in this study. An extension of this performance analysis on these extensive cards is left for future work.

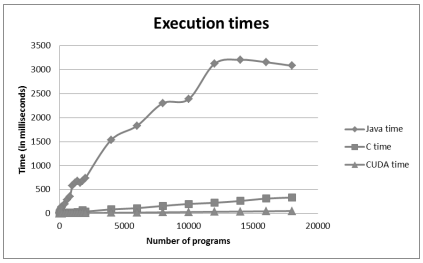

Fig. 6. Execution times for SimP (Java), C and CUDA-C ENPS simulators

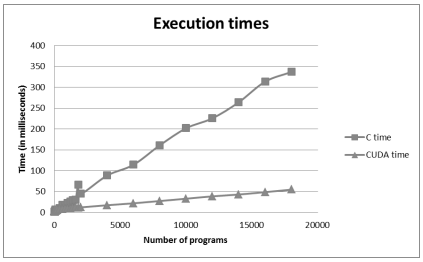

Fig. 7. Execution times for C and CUDA-C ENPS simulators

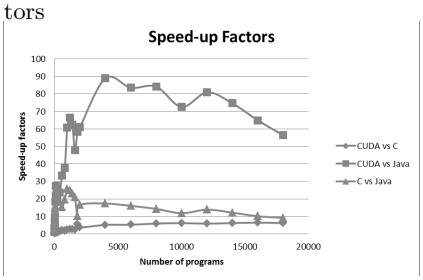

Fig. 8. Speed–up factors for SimP (Java), C and CUDA-C ENPS simulators

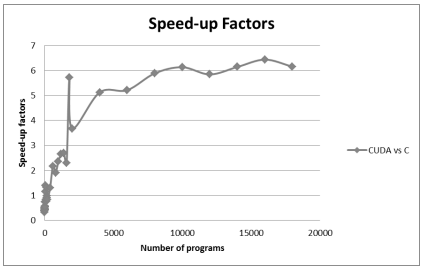

Fig. 9. Speed–up factors for C and CUDA-C ENPS simulators

Fig. 10. Execution times and speed–up factors obtained from the simulation of the dummy model

The models have been simulated on an *Nvidia GeForce GTX 460M* card with 1.5GB of dedicated RAM memory [19]. This model supports the new Fermi technology. Fermi cards allow programmers to make use of new features impossible (or at least very hard) for previous models. Some examples of these features are the computation of recursive functions and atomic operations with float–type numbers [19]. The impact of these features on the simulator code is really important. Thus, by employing these features, the development process is eased and the simulator code is much clearer. For instance, in order to calculate production functions in programs, recursion comes as a straightforward approach. That is because these general mathematical expressions can be easily represented as tree–like structures. Tree–like structures are usually stepped through by using recursive algorithms. Another important feature brand–new on Fermi technology

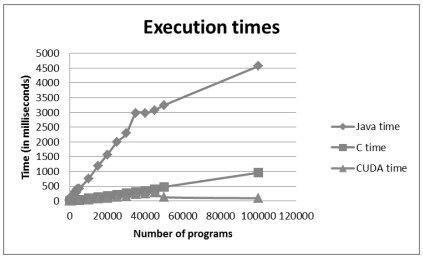

Fig. 11. Execution times for SimP (Java), C and CUDA-C ENPS simulators

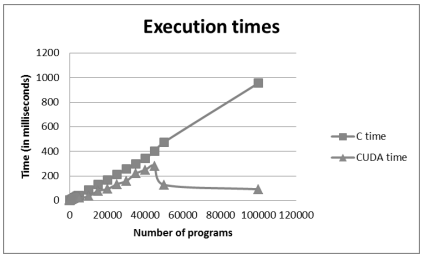

Fig. 12. Execution times for C and CUDA-C ENPS simulators

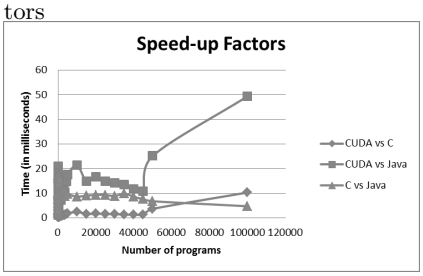

Fig. 13. Speed–up factors for SimP (Java), C and CUDA-C ENPS simulators

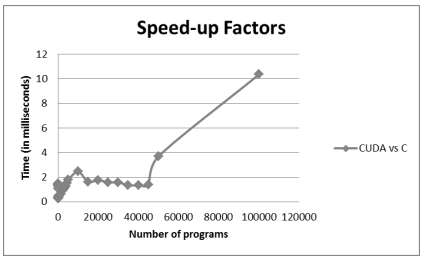

Fig. 14. Speed–up factors for C and CUDA-C ENPS simulators

Fig. 15. Execution times and speed–up factors obtained from the simulation of the e^x model

is the use of atomic operations on floating–point numbers. In order to add up the contributions from repartition protocols, these instructions have been used, as the values of contributed variables can be modified by different CUDA threads. However, the use of these features comes as a cost. Specifically, the GPU simulator can only be run on Fermi Nvidia cards, as previous models do not support these features. An improvement on the code in order to add compatibility for previous graphic cards is thus left as a future work.

5 Conclusions

This paper displays how much the simulation of ENPS models can be accelerated by applying the GPU technology. It shows a noticeable speed–up factor obtained from comparing the CUDA–C and C simulation runtimes and a dramatic acceleration when these execution runtimes are compared with a previously existent Java simulator. As the number of processors in the GPU device and the number of programs per model is augmented, one can expect a greater speed–up factor. The Research Group on Natural Computing owns a High Performance Computing server equipped with an Nvidia Tesla C1060 card [19], [5]. These cards contain 240 streaming multiprocessors and 4GB of RAM memory. Unfortunately, our simulator is unable to run on this server. The reason is that

Nvidia Tesla C1060 cards do not implement the new Fermi technology. Therefore, they cannot execute recursive functions and atomic operations on floating–point numbers. Hence, an adaptation of our code to lay out these constraints is conveniently left as a future extension.

6 Acknowledgements

Manuel García-Quismondo and Mario J. Pérez-Jiménez are supported by project TIN 2009-13192 from "Ministerio de Ciencia e Innovación" of Spain, co-financed by FEDER funds. Manuel García-Quismondo and Mario J. Pérez-Jiménez are also supported by "Proyecto de Excelencia con Investigador de Reconocida Valía P08-TIC-04200" from Junta de Andalucía. Manuel García-Quismondo is also supported by the National FPU Grant Programme from the Spanish Ministry of Education.

References

1. N. Azizi, I. Kuon, A. Egier, A. Darabiha, P. Chow. Reconfigurable molecular dynamics simulator. *Annual IEEE Symposium onf Field–Programmable Custom Computing Machines*, 2004, 197–206.
2. F. Bleichrodt, R.H. Bisseling, H.A. Dijkstra. Accelerating a barotropic ocean model using a GPU. *Ocean Modelling*, 41, (2012), 16–21.
3. C. Buiu, A.B. Pavel, C.I. Vasile, I. Dumitrache. Perspectives of using membraine computing in the control of mobile robots. *Beyond AI. Interdisciplinary Aspects of Artificial Intelligence*, 08/12/2011-09/12/2011, Pilsen, Czech Republic.
4. Buiu, C., Vasile, C., Arsene, O.: Development of membrane controllers for mobile robots. Information Sciences, vol 187, doi:10.1016/j.ins.2011.10.007 (2012), 33-51.
5. 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.
6. 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. *International Journal of Computers, Communications and Control*, 4, 3 (2009), 234-243.
7. M. García-Quismondo, M.J. Pérez–Jiménez, L.F. Macías–Ramos. Implementing ENPS by means of GPUs for AI applications. *Beyond AI. Interdisciplinary Aspects of Artificial Intelligence (BAI 2011)*, 08/12/2011-09/12/2011, Pilsen, Czech Republic.
8. *http://www.gnu.org/copyleft/gpl.html*
9. Y. Gu, T. VanCourt, M. Herbordt. Accelerating molecular dynamics simulations with configurable circuits. *International Conference on Field–Programable Logic and Applications*, 2005, 475–480.
10. J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, *Queue*, 6, 2 (2008), 40-53.
11. Gh. Paun, R. Paun. Membrane Computing and Economics: Numerical P Systems. *Fundamenta Informaticae*, 73, 1-2 (2006), 213-227.
12. Gh. Păun, G. Rozenberg, A. Salomaa (eds.): *The Oxford Handbook of Membrane Computing*. Oxford University Press (2010).

13. A.B. Pavel, C. Buiu. Using enzymatic numerical P systems for modeling mobile robot controllers. *Natural Computing*, DOI: 10.1007/s11047-011-9286-5 (2011).
14. A.B. Pavel, C.I. Vasile, I. Dumitrache: Robot localization implemented with enzymatic numerical P systems. submitted.
15. A.B. Pavel, Membrane controllers for cognitive robots. *Masters thesis*, Department of Automatic Control and System Engineering, Politehnica University of Bucharest, Romania, February 2011.
16. J.E. Stone, D.J. Hardy, I.S. Ufimtsev, K. Schulten. GPU-accelerated molecular coming of age. *Journal of Molecular Graphics and Modelling*, 29, (2010), 116–125.
17. C.I. Vasile, A.B. Pavel, I. Dumitrache, Gh. Păun: On the Power of Enzymatic Numerical P Systems. submitted.
18. *http://www.java.com*
19. *http://www.nvidia.com/object/cuda_home_new.html*