2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

A CUDA implementation of the Spatial TAU-leaping in Crowded Compartments (STAUCC) simulator

Giulia Pasquale [*], Carlo Maj [†] [‡], Andrea Clematis [*], Ettore Mosca [‡],
Luciano Milanesi [‡], Ivan Merelli [‡] and Daniele D'Agostino [*]
[*]*Institute for Applied Mathematics and Information Technologies*
National Research Council of Italy, Genoa, Italy
{pasquale, clematis, dagostino}@ge.imati.cnr.it
[†]*Department of Computer Science, Systems and Communications*
University of Milano-Bicocca, Milan, Italy
carlo.maj@disco.unimib.it
[‡]*Institute for Biomedical Technologies*
National Research Council of Italy, Segrate (Milan), Italy
{mosca, merelli, milanesi, maj}@itb.cnr.it

***Abstract*—The increasing awareness of the pivotal role of noise in biochemical systems has given rise to a strong need for suitable stochastic algorithms for the description and the simulation of biological phenomena. However, the high computational demand that characterizes stochastic simulation approaches coupled with the necessity to simulate the models several times to achieve statistically relevant information on the model behaviors makes the application of such kind of algorithms often unfeasible. So far, different parallelization approaches have been employed to reduce the computational time required for the analysis of biochemical systems modeled using stochastic algorithms. Most of the proposed solutions use an embarrassingly parallel approach to run in parallel several simulations using the cores of a workstation and/or the nodes of a cluster. In this work we present the Spatial TAU-leaping in Crowded Compartments (STAUCC) simulator, a software that relies on an efficient CUDA implementation of the Stau-DPP algorithm, a voxel-based method for the stochastic simulation of Reaction-Diffusion processes. We evaluate its application and performance for the modeling of diffusion processes simultaneously occurring within a space represented considering different levels of granularity.**

***Keywords*-parallel stochastic simulations in CUDA; Reaction-Diffusion processes in CUDA; voxel-based methods; tau-leaping; gene-regulatory networks;**

I. Introduction

The simplified representation of intracellular reactions with deterministic and homogeneous approaches may be inadequate when the observed properties of the real system are also the consequence of stochastic fluctuations and macromolecular crowding. In such situations, a more accurate representation is achievable by using modeling approaches based on stochastic Reaction-Diffusion (RD) systems taking into account the geometry of the space and the effect of crowding elements.

Two broad classes of methods for stochastic RD exist: particle-based and voxel-based. Particle-based methods compute the Brownian motion of individual particles (e.g. MCell [1], Smoldyn [2] and GridCel [3]), while voxel-based methods calculate changes in the number of molecules occurring in small well-stirred compartments in which the modeled space is partitioned (e.g. STEPS [4], MesoRD [5], NeuroRD [6], MURCIA [7] and Sτ-DPP [8]).

The Spatial TAU-leaping in Crowded Compartments (STAUCC) simulator uses the Sτ-DPP algorithm [8]. This algorithm is based on a modified version of τ-leaping method (described in [9]) taking into account the spatial conformation of compartments and the volumes occupied by molecules. Hence, STAUCC enables a proper representation of crowding effects on biochemical reactions. Indeed, this aspect turns out to be very useful when analyzing environments in which space and mobility are important constrains for the stochastic behavior of the system, such as during gene expression in the nucleus - the application example considered in this work -, but also during neuronal transmission of action potentials and a wide variety of other biochemical processes.

So far, different parallelization approaches have been employed to reduce the computational time required for the analysis of biochemical systems modeled using stochastic algorithms. Most of the proposed solutions are based on the use of embarrassingly parallel approaches to run several instances of model simulation at a time, while maintaining the sequentiality in the simulation algorithm. A typical example is given by the exploitation of distributed platforms, such as Grid computing [10]. Other related works concern the use of multicore CPUs [11] and HPC clusters [12].

However, an efficient parallelization of the algorithm itself, improving the execution time of each specific simulation, can more effectively exploit the available computational resources allowing to reach a higher computational efficiency. An MPI implementation of STAUCC, which parallelizes the Sτ-DPP algorithm [8], was presented in [13]. However, the major limit of such a kind of parallelization is that the exploitation of all the cores within a single node of a cluster yields to good performance only if the MPI implementation is smart enough to exploit the shared memory to perform the communications. The exploitation of multiple nodes proved to be not suitable due to the large communication overhead, at least using a Gigabit Ethernet.

To overcome this limit, in literature some works consider the use of Field Programmable Gate Arrays (FPGAs) (see, e.g., [14]). However, we preferred to consider the use of General Purpose-Graphics Processing Units (GPGPUs) because of the larger diffusion among the scientific community due to its parallelization potentiality. In effect, the variety of programming models for heterogeneous computing which are increasingly appearing offer a major flexibility for possible future extensions of our simulator. Thus, we started from the concepts outlined in [15] for the GPU implementation of such a kind of simulation algorithm, and present a CUDA-based parallelization of the Sτ-DPP algorithm, which is the computational kernel of the software that we call STAUCC simulator. With respect to the algorithm considered in [15], in the voxel-based Sτ-DPP algorithm, as said above, the compartments and the molecules inside them have a quantitatively defined volume, this fact complicates the implementation because requires more synchronization operations (e.g., for checking, at each time step, the free space left in the membranes). Moreover, the available Fermi and Kepler architectures (compute capabilities 2.x and 3.x respectively) present features that make non-suitable some of the optimizations proposed in [15] for the Tesla architecture (compute capability 1.x), giving way to other kinds of considerations. By exploiting the shared memory architecture of a common graphics processor we avoid most of the communication overhead achieving higher and more scalable speedups in comparison to the sequential and MPI implementations [13] of the Sτ-DPP algorithm.

In Section II we resume the cardinal points of the algorithm whose CUDA implementation is described in Section III; in Section IV the gene regulatory network to which we applied the STAUCC simulator is presented; in Section V some details about the simulations performed are provided, together with the speedups achieved, and, finally, Section VI draws the conclusions and future work.

II. Spatial TAU-leaping in Crowded Compartments (STAUCC) simulator

In this Section we briefly report the key features of the Sτ-DPP algorithm [8] employed by the STAUCC simulator. A more accurate description is provided in [13].

A system is defined by a set of compartments, or membranes, each containing a multiset of objects, or molecular species. The state of the system at a time frame is given by the number of molecules of each species inside each compartment.

For each compartment there is a set of rules describing the possibilities of interaction among the molecules contained in it and that can occur at each step. Two types of rules are defined: reactions and diffusions. A reaction rule substitutes the molecules specified in its left-hand side (reactants) with the molecules specified in its right-hand side (products). A diffusion rule moves reactants from the current compartment to the compartment specified in the right-hand side.

In case a reactant that is involved in more than one rule is present in little amount, the occurrence of a rule before the other(s) may prevent the other(s) rule(s) to happen and vice versa. In such cases, the system can evolve following different dynamics according to the order in which rules occur. The rules in which are involved potential limiting reactant(s) are defined "critical rules"; these rules are the ones mainly responsible of model behavior variability.

As defined in the original Gillespie's Stochastic Simulation Algorithm (SSA), each rule is associated with its propensity function (stochastic reaction rate), defined as $a = c \cdot h$, i.e., the product of c - the stochastic constant associated to the rule - and h - a combinatorial function depending on the left-hand side of the rule - [16]. However, the novelty of the Sτ-DPP algorithm (see [8]) is that it takes into account also the size of compartment volumes and molecules involved in the system in order to describe the effect of crowding on the rate of cellular processes. For this reason, in STAUCC the propensity functions of reactions are computed by explicitly considering also the amount of free space in the current compartment. The free space of a compartment is defined as the space available in the empty compartment minus the space occupied by the molecules and compartments inside it. Thus, while propensity functions of rules representing diffusive events or reactions of order less than two (e.g., $A \to R$) are not influenced by the free space F in the target compartment, propensity functions of reactions of order greater than one (e.g., $A + B \to R$) are scaled according to such free space: $a = \frac{c \cdot h}{F}$.

At each iteration, a time increment τ is computed independently inside each compartment on the basis of its current state. Then, the smallest time increment is selected among those computed (in a parallel implementation this must be done after synchronizing the computation in all compartments) and shared by all compartments as the common time increment on which to synchronize their evolutions, as specified by the standard τ-leaping algorithm [9].

Indeed, this common time increment is used to evaluate the step evolution of the entire system: first of all, it determines the way to proceed in the current iteration: SSA-like evolution, τ-leaping evolution with non critical reactions only, or τ-leaping evolution with non critical reactions and one critical reaction. Then, the same time increment is used, together with the computed propensity functions to sample the number of reactions to be executed in each compartment, according to the chosen modality.

A second synchronization point among the compartments is then necessary for a consistent application of the extracted rules, which implies the exchange of molecules among the membranes.

Finally, before stepping over the next iteration of the simulation, we need to ensure that the system has not ended up in an unfeasible state, i.e., in some compartments the room filled by molecules may exceeds the available space. To this end, the free space left inside each membrane is checked: if it results negative inside at least one membrane (third synchronization point) the time step is halved, the state of the system before the application of the rules is restored and, accordingly to the halved time step another set of rules is extracted and applied to the system; the verification is repeated on the new state of the system and the time step is repeatedly halved until it reaches a feasible state.

III. CUDA IMPLEMENTATION OF THE STAUCC SIMULATOR

The CUDA implementation of the STAUCC simulator presented in this work exploits a two-level parallelism, which means that it is possible to launch concurrently several simulations of the model on the same GPU, while, in each simulation, the temporal evolution of the state of all membranes is also computed in parallel. Indeed, as already said, in the STAUCC algorithm the space is partitioned in smaller regions or compartments, and most of the computation for each compartment is independent from the others and thus can be performed in parallel.

The possibility of relying on an efficiently accelerated parallelization of STAUCC is essential for doing not only statistical studies of the behaviors of complex systems, but also parametric analysis of the dynamics of their models. Actually, a further level of parallelism could be explicitated, regarding the application of all the rules in parallel in each compartment, but typically the number of rules is of the order of dozens, therefore this level of parallelization has not been considered for the moment.

We developed a CUDA kernel in which the thread blocks carry out independent simulations of the model, given an initial condition of the system and a seed for the generation of the chain of random numbers used by the algorithm throughout the computation of the system's dynamics. Inside each block, the workload is balanced among the threads assigning to each of them the computation of the evolution of the most possible equal number of compartments. In this way, for any given number of membranes in the system, we are free to choose the number of threads per block which best fits the GPU architecture, and then launch an arbitrary number of blocks according to the number of simulations to perform.

The choice of computing a simulation instance inside a single thread block is dictated by the need of synchronizing the compartments during the dynamic evolution of the system and, at the moment, there is no possibility to synchronize different thread blocks on the GPU; if we would split the simulation among multiple thread blocks, the only way to synchronize them would be to synchronize the host and the device, resulting in a prohibitive slow-down of the computation. Moreover, in this way we minimize the communication overhead, avoiding any communication between the host and the device during the simulation.

As anticipated, inside the thread block, at each iteration of the algorithm (or time frame) three communication/synchronization points are required (see Section II), in order to: a) determine the smallest time increment; b) verify the correctness of the overall system evolution with respect to the resulting free space in the compartments; c) transfer objects between the membranes. The synchronization operation on the present CUDA architectures is possible only among the threads of a block, therefore acting on the structure of the algorithm.

A fundamental problem in developing parallel applications is the memory management. In a first version, we placed in the device shared memory those variables which must be visible to all threads, either because involved in constraining the overall system evolution (e.g., the computed time increment τ and the free space for each compartment), or because are changed by the threads of other compartments (e.g., the number of molecules). Instead, in the device constant memory we kept the fixed parameters of the model (e.g., the stochastic constants, the sizes of molecules and compartments, the left-hand and right-hand side of rules, etc.). However, subsequently we decided to place all the variables in the device global memory because of at least two reasons. The first one is the principal reason and is because our main goal is to provide a simulator which can deal with an arbitrary number of membranes, and keeping in the shared or constant memory even a single location (integer or floating point, 4 bytes anyway) per membrane (i.e., an array of size equal to the number of membranes) imposes too strict limits to the size of the systems which is possible to simulate. For example, if the shared memory is of 48 Kbytes (a reasonable value for the NVIDIA GPUs available nowadays on the market), and we want to run concurrently 16 simulation instances (i.e., we launch a grid of 16 blocks), then the maximum number of membranes for the system is $(48 * 1024)/(16 * 4) = 768$, which is quite small for our purposes. The second reason is that, keeping in such kind of memories so few arrays because of what we have said above, the first version turned out to be not faster enough to justify their use. In [15] the smart use of shared and constant memories was of fundamental importance because the main memory of a graphics card based on the Tesla architecture is not cached, while this feature is present on Fermi-based devices.

In a second version of the implementation, to parallelize also the memory allocation/deallocation phases, we allocated/released dynamically the arrays in the device heap memory, doing a per-block allocation/deallocation for the ex shared memory arrays (i.e., the arrays common to all threads in a block), and a per-thread allocation/deallocation for the others. However, in this way the threads access very sparse memory locations, thus causing a very bad use of the L1 cache, this fact globally worsen the execution time despite the speedup of the allocation/deallocation phases.

The best solution achieved for the memory management of this application therefore is to allocate through a unique host-side CUDA API call such as *cudaMalloc* all the one-dimensional arrays used by all threads in all blocks (and inside the computational kernel we declare pointers to the memory regions reserved to the thread blocks to reduce address arithmetics). The arrangement of the variables in memory at this point must be such that the threads access memory locations with as much as possible a unitary stride. For example, if we need to store for each membrane the amount of molecules for each species inside it, and we have to deal with M membranes, S species per membrane, and T threads per block (supposing for simplicity that M is a multiple of T) a good solution is to partition the global one-dimensional array allocated (which will be of size $M \times S$) into S subarrays of M elements each, and to further divide the subarrays into groups of T adjacent elements. In this way we can access the variables in a fully coalesced way with an external loop over the S species and an internal loop over the membranes associated to each thread ($\frac{M}{T}$): at each iteration, the T threads in the block will access a contiguous group of T cells in memory.

We minimized the calls to device functions inside the kernel and limited them only to those quite long parts of code repeatedly executed, to avoid further overheads due to parameters passing.

Regarding the stochasticity of the algorithm, we used the device APIs of the cuRAND library [17] provided within the CUDA Toolkit (Production Release 5.5). Such a choice is justified by the following considerations. Generating random numbers on the CPU and then sending them via PCI-express (PCI-e) bus on the GPU has in general relatively poor performance, both because of the performance gap between GPU and CPU on random number generation itself (considering, e.g., a comparison between cuRAND library and Intel Math Kernel Library (MKL) [17]) and because of the limited bandwidth of the bus. Moreover, in our case this solution would be unfeasible because: 1) we cannot know in advance even the approximated number of random numbers we will need and 2) we enclose all the computation in a kernel and we do not return to the host in the middle of the simulations. Trying to re-implement some of the functionalities of the cuRAND library in a customized random number generator seems to be not a smart solution because both the qualitative and computational performance of the library in the generation of random numbers of various distributions have been deeply tested so far and have assessed it to be among the most flexible and well-tuned libraries for generating random numbers in device code.

In particular, we used the pseudo random number generator based on the XORWOW algorithm [18]. As suggested in the library guide, for the highest quality parallel pseudo-random number generation, each experiment (in our case, each block) has been assigned a unique seed, and within an experiment each membrane has been assigned a unique sequence number (i.e., a random state). Thus during the simulation each thread extracts the chain of random numbers from the random state of the membrane of which it is computing the evolution.

Having enclosed in a kernel the computation of the dynamic evolutions of the system allows to avoid any communication between host and device during the simulation. Yet it is true that, having stored the variables of the system in the global memory, it could be possible to retrieve them if necessary with little effort: we could simply return the kernel, keeping track of the current time frame, and then launch another kernel starting from the step where we stopped; but this would considerably slow down the computation time. Thus, the state of the system (the time frame plus the number of molecules in each compartment) is copied every N_{sample} iterations in a pre-allocated buffer in the device global memory that is read back from the host after the kernel returns.

IV. Simulating a gene regulatory network with STAUCC

It is well known in biology that gene expression can be regulated by a different number of factors, which inhibit or promote gene expression under different conditions [19]. Typically regulatory factors are present in low copy numbers and, in fact, noise plays a relevant role in gene expression [20]. A pivotal element in the interactions between regulatory factors and DNA is given by the diffusion of regulatory factors within the cell nucleus [21], an environment crowded by chromatin (an macromolecular agglomeration composed of DNA and proteins). Taken together, these considerations underline the need for a stochastic RD approach in order to properly represent the dynamics of transcription and thus study the effects of regulatory factors.

We defined a model of a gene regulatory system as follows. The nucleus is represented as a two-dimensional grid composed of a finite number of squared compartments of the same size (Figure 1). Six types of objects are considered: four regulatory factors: F_1, F_2, F_3, F_4 and two genes: G_1 and G_2. While the position of genes is fixed, regulatory factors are free to diffuse in the adjacent compartments. The compartments in which the genes are placed, as well as the adjacent compartments, have lower free space in order to represent the crowding due to the presence of chromatin. Factors that reach the compartments in which the genes are placed can bind the genes and either activate or inhibit gene activities (Figure 2). This process is simulated with the binding of factors to genes and thus each of the two genes can be in one of the following three states: free, a state in which a factor can bind the gene; active, in case an activation factor binds the gene; inhibited, in case an inhibition factor binds the gene.

More specifically, sixteen reaction rules have been defined in order to describe the set of possible interactions occurring between genes and regulatory factors, eight rules represent the complexes formation and eight rules represent the corresponding dissociations (Table I).

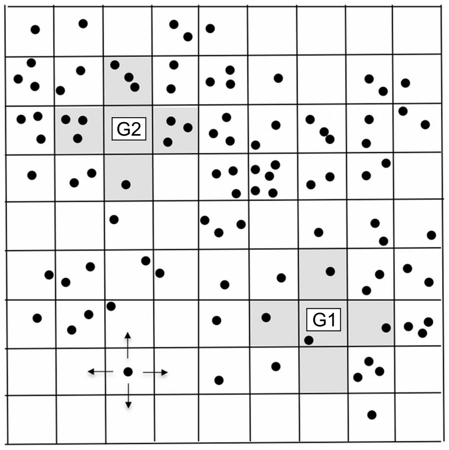

Figure 1. Schematic representation of the space domain. The 2D-grid of compartments represents a section of the nucleus that contains the two genes G_1 and G_2. Regulatory factors (filled circles) diffuse within this environment in all the possible directions (arrows). Compartments filled in gray have a lower free space. The representation is not in scale with actual sizes used in simulations.

The model has been simulated using different stochastic constants and considering an increasing number of compartments (16^2, 32^2, 64^2) for the description of the nucleus space.

The stochastic constants characterizing this kind of systems are: i) the constants for gene-regulatory factor association/dissociation; ii) regulatory factors diffusion coefficients. Association/dissociation constants have been changed in a range between 10^1 and 10^3, while diffusion coefficients have been set at least two orders of magnitude higher than association/dissociation constants. The fact that the time-scale of diffusion processes is much slower than the time scale of reaction processes is crucial for maintaining valid the well-stirred assumption within each compartment [8].

At the beginning of a simulation each regulatory factor F_i is placed in a different corner of the lattice, while the two genes are placed in two inner compartments (Figure 1).

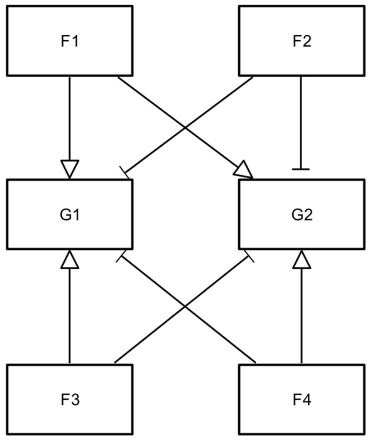

Figure 2. Activity flow in the gene regulatory network. Representation of the gene regulatory network using the SBGN (Systems Biology Graphical Notation).

Table I
LIST OF RULES DEFINED FOR REACTION AND DIFFUSION PROCESSES. REACTIONS OCCUR ONLY IN COMPARTMENTS WHERE G_1 AND G_2 ARE PLACED, WHILE DIFFUSIONS OCCUR IN ALL COMPARTMENTS; (x, y) ARE THE COMPARTMENT'S COORDINATES, $n \in \{-1, 0, 1\}$ AND $z \in \{1, 2, 3, 4\}$.

DESCRIPTION	RULE
Activation of G_1 mediated by F_1	$F_1 + G_1 \rightarrow G_1^+$
Dissociation of F_1 from G_1	$G_1^+ \rightarrow F_1 + G_1$
Activation of G_2 mediated by F_1	$F_1 + G_2 \rightarrow G_2^+$
Dissociation of F_1 from G_1	$G_2^+ \rightarrow F_1 + G_2$
Inhibition of G_1 mediated by F_2	$F_2 + G_1 \rightarrow G_1^-$
Dissociation of F_2 from G_1	$G_1^- \rightarrow F_2 + G_1$
Inhibition of G_2 mediated by F_1	$F_2 + G_2 \rightarrow G_2^-$
Dissociation of F_2 from G_2	$G_2^- \rightarrow F_2 + G_2$
Activation of G_1 mediated by F_3	$F_3 + G_1 \rightarrow G_1^+$
Dissociation of G_1 from F_3	$G_1^+ \rightarrow F_3 + G_1$
Inhibition of G_2 mediated by F_3	$F_3 + G_2 \rightarrow G_2^-$
Dissociation of F_3 from G_2	$G_2^- \rightarrow F_3 + G_2$
Inhibition of G_1 mediated by F_4	$F_4 + G_1 \rightarrow G_1^-$
Dissociation of G_1 from F_4	$G_1^- \rightarrow F_4 + G_1$
Activation of G_2 mediated by F_4	$F_4 + G_2 \rightarrow G_2^+$
Dissociation of F_4 from G_2	$G_2^+ \rightarrow F_4 + G_2$
Diffusion from (x, y) to $(x + nx, y + ny)$	$F_z^{x,y} \rightarrow F_z^{x+nx,y+ny}$

Simulations of the model with STAUCC show that over time the genes fluctuate among all the three possible states. Not surprisingly, the higher the stochastic constants associated to formation and dissociation of $F_i :: G_j$ complexes, the higher the expected number of transitions among gene states in the same interval of time. Moreover, these transitions are also influenced by the number of molecules of each regulatory factor available in the system, i.e., the presence of higher copy numbers lead the genes to be more prone to state variation (Table II).

It is worth noting that the explicit consideration of the spatial dimension enables to study possible relations between regulatory factors localization and gene activation. More specifically, as a consequence of diffusion dynamics,

Table II
DATA RETRIEVED FROM A RUN OF STAUCC FOR EACH MODEL CONFIGURATION CONSIDERING A SIMULATION TIME OF 100 TIME UNIT.

REGULATORY FACTORS	REACTION CONSTANTS	DIFFUSION CONSTANTS	G_1 STATE VARIATIONS
5	10^3	10^5	244
5	10^2	10^5	26
5	10^1	10^5	3
10	10^3	10^5	635
10	10^2	10^5	89
10	10^1	10^5	11
20	10^3	10^5	987
20	10^2	10^5	113
20	10^1	10^5	16

according with the activators and inhibitors localization a gene can go through progressive phases characterized by different state probabilities (Figure 3).

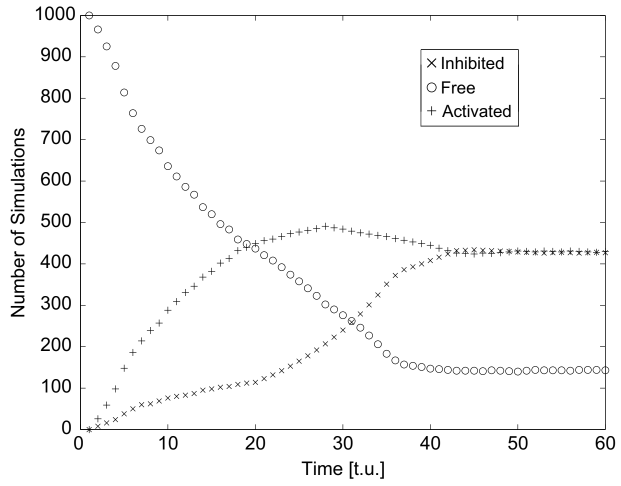

Figure 3. Simulations obtained considering an initial spatial configuration in which an activator is located in the closest corner with respect to the fixed location of $G1$, while an inhibitor is placed in the farthest corner with respect to $G1$. The number of simulations in which $G1$ is in the "activated", "inhibited" or "free" state (vertical axis) is reported in relation to the simulation time (horizontal axis). After a transient phase in which the most frequent state for $G1$ is "activated" the system reaches a plateau in which the two states "activated" and "inhibited" have similar frequency.

V. PERFORMANCE EVALUATION OF THE CUDA IMPLEMENTATION

The STAUCC parallel simulator presented in this paper has ben tested on an NVIDIA GeForce GTX 580 device. The speedups have been computed against a sequential C implementation run on an Intel Xeon E5645 CPU, that are equivalent in terms of the price for their acquisition.

Of course, the actual execution time of an instance of simulation is proportional to the number of iterations done to compute the system evolution for the requested amount of time. However, as anticipated in Section I, the number of steps in turn depends on the system's initial condition, parameter values, and, in case the first two are fixed, at least on the sequence of random numbers generated, which can influence the dynamic routes followed by the system. In general, the faster the dynamics, the higher the number of iterations computed, due to the fact that small time increment values will be required to satisfy the leap condition. As a consequence, the speedup of the STAUCC simulator is computed considering the average time for an iteration of the model simulation (which in Table III is called *timePerStep* and is obtained dividing the total execution time by the total number of iterations performed) and dividing this quantity in the sequential simulation by the same quantity in the parallel simulation (in this case, the total number of iterations performed is the sum, over the total number of simulations launched, of the number of iterations computed in each instance of simulation).

Three time measurements appear in Table III: the first one is the time necessary for memory allocation -in CPU code- and -in case of GPU code- it includes also the uploading of parameters and variables wich define the system on the device memory (*allocTime* in Table III); the second one is the time for memory deallocation -in CPU code- and -again only in GPU code- it includes the downloading of results, which, as explained in Section III, is a buffer containing the state of the system at each N_{sample} iterations (*freeTime* in Figure III); finally, the third one is the *timePerStep* mentioned above obtained dividing the execution time of the simulation by the total number of iterations.

As explained in Section IV, we modeled the nucleus space as a grid of increasing number of compartments (16^2, 32^2, 64^2). For all the considered granularities, we distributed the workload among a block of 32 threads (i.e., one warp per block). In this way each thread computes the evolution of 8, 32, 128 compartments respectively.

For each system size, we launched concurrently on the GPU an increasing number of stochastic simulations, i.e., we launched the simulation kernel with an increasing number of blocks (as explained in Section III, the evolution of a simulation instance is computed entirely by a single block of threads independently from the others). As larger systems occupy more space in the global memory of the device, and we avoid any communication between host and device during the simulation (which is performed running a single kernel as explained), the maximum number of simulation instances that is possible to launch decreases as the number of compartments increases. For this reason, in Figure III the kernel can compute a maximum of 256 simulation instances for the system with 16^2 compartments, 128 for the system with 32^2 compartments and 64 for the system with 64^2 compartments .

The memory allocation/deallocation times (*allocTime*, *freeTime* in Table III) remain quite constant when varying the system size in the considered range both in the sequential and in the parallel implementations, for this reason we do not consider them when calculating the speedups (it is sufficient

Table III
EXECUTION TIMES IN MILLISECONDS OF THE SEQUENTIAL AND CUDA IMPLEMENTATIONS OF THE STAUCC SIMULATOR WITH RELATED SPEEDUP VALUES. *Nmem* IS THE NUMBER OF COMPARTMENTS; *gridim* IS THE NUMBER OF BLOCKS PER GRID IN THE LAUNCH CONFIGURATION OF THE SIMULATION KERNEL (THE NUMBER OF THREADS PER BLOCK IS NOT SHOWN BECAUSE FIXED TO 32); *allocTime*, *freeTime*, *timePerStep* ARE THE TIME MEASUREMENTS EXPLAINED IN THE TEXT.

	Nmem	gridDim	allocTime [ms]	freeTime [ms]	allocTime + freeTime	timePerStep [ms]	speedup
parallel	256	1	163,08	0,67	163,75	6,52	0,3
		32	169,70	1,15	170,85	0,36	5,8
		64	158,19	1,62	159,81	0,17	12,4
		128	184,72	2,00	186,73	0,09	22,6
		256	159,97	2,72	162,69	0,09	24,5
	1024	1	192,64	0,82	193,46	22,10	0,3
		32	177,04	2,10	179,14	1,38	5,5
		64	185,75	2,81	188,56	0,64	12,0
		128	177,98	4,18	182,16	0,38	20,4
	4096	1	174,50	1,10	175,60	80,40	0,4
		32	172,88	4,14	177,02	4,88	6,1
		64	198,01	7,01	205,01	2,43	12,3
sequential	256	x	0,04	81,19	81,23	2,08	
	1024	x	0,04	82,43	82,47	7,64	
	4096	x	0,04	88,82	88,86	29,87	

to keep in mind that the overall time necessary for the memory management on the GPU is about 2 times slower than on the CPU).

The *timePerStep* of course increases with the size of the system, both in the sequential and parallel implementations. However, it becomes smaller when launching more simulation instances in the parallel implementation because the total execution time increases less than the number of iterations computed. This means that the device is via via exploited more efficiently when launching more thread blocks which hide the latencies caused by sequential dependences and memory stores and loads. It is reasonable to think that it should reach a minimum when the device's cores are fully exploited, after which it should begin to grow, but, due to the limited size of the device memory, we cannot observe this trend completely.

The implementation scales well when augmenting the number of compartments: for a fixed number of simulation instances (*gridDim* in Table III) increasing the system dimension does not affect the speedup achievable. Moreover, due to the discussed trend of the time per iteration, the speedup increases with the number of simulation instances launched.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we presented the STAUCC simulator, a software for the efficient stochastic simulation of RD processes in multi-compartment crowded environments. Beyond launching hundreds of simulation instances of a system in an embarassignly parallel way, this simulator exploits the intrinsic parallelism of the the Sτ-DPP algorithm proposed in [8], which is here implemented on a CUDA architecture. The exploitation of GPU allows to accelerate the simulation of the evolution of a multi-compartment system by alternating parallel execution phases for the independent computations in each compartment and synchronization points to coherently update the state of the system allowing the chemical factors to potentially move across the compartments.

To test the performance of the simulator we built an abstract model of the diffusion of regulatory factors within the nucleus cell space and we employed STAUCC to stochastically simulate the evolution of such system. The computational speedups obtained, up to around twenty times faster than the sequential version when fully exploiting the GPU resources (i.e., when launching a sufficiently high number of simulation instances) pave the way to more detailed analysis of such systems and RD processes in general. In effect, through the analysis of our model simulations we verified that it is possible to find relations between the spatial localization of the different biochemical actors and the dynamics of the system.

Using a single GPU, we observed that, keeping fixed the computational resources assigned to a single simulation instance (i.e., for a fixed number of compartments balanced among a fixed number of threads per block), the speedup increases well as the number of stochastic simulation instances launched increases. Because the single simulation instance is fully computed by a single thread block - i.e., on a single Streaming Multiprocessor (SM) - completely independent from the others, we expect that, having more SMs per GPU and launching more blocks of threads would not influence this trend because we have not got any communication overhead.

Instead, the extension of the presented implementation of the simulator for a multi-GPU usage, which would allow to collect statistically significant results in even less time is the next step in which we are working on. Indeed, in this case, the global memory from which the threads read and write is split among different devices, and the role of the communication overhead in such a situation is not obvious.

In particular, it should be verified whether the considerations made in Section III about the memory management and allocation for the single-GPU case would still be valid in a multi-GPU context.

Acknowledgments

The work has been supported by the Italian Ministry of Education and Research through the Flagship (PB05) "InterOmics", "HIRMA" (RBAP11YS7K) projects and the European "MIMOmics" (305280) project.

References

[1] R. A. Kerr, T. M. Bartol, B. Kaminsky, M. Dittrich, J. C. Chang, S. B. Baden, T. J. Sejnowski, and J. R. Stiles, "FAST MONTE CARLO SIMULATION METHODS FOR BIOLOGICAL REACTION-DIFFUSION SYSTEMS IN SOLUTION AND ON SURFACES." *SIAM Journal of Scientific Computing*, vol. 30, no. 6, p. 3126, Oct 2008.

[2] S. S. Andrews, "Spatial and stochastic cellular modeling with the Smoldyn simulator." *Methods Mol Biol*, vol. 804, pp. 519–542, 2012.

[3] B. Laurier, S. A. Assaad, M. Dumontier, and W. J. Gross, "GridCell: a stochastic particle-based biological system simulator." *BMC Systems Biology*, vol. 2, p. 66, 2008.

[4] I. Hepburn, W. Chen, S. Wils, and E. D. Schutter, "STEPS: efficient simulation of stochastic reaction-diffusion models in realistic morphologies." *BMC Systems Biology*, vol. 6, no. 36, 2012.

[5] D. Fange, A. Mahmutovic, and J. Elf, "MesoRD 1.0: Stochastic reaction-diffusion simulations in the microscopic limit." *Bioinformatics*, vol. 28, no. 23, pp. 3155–3157, 2012.

[6] R. F. Oliveira, A. Terrin, G. D. Benedetto, R. C. Cannon, W. Koh, M. S. Kim, M. Zaccolo, and K. T. Blackwell, "The role of type 4 phosphodiesterases in generating microdomains of cAMP: large scale stochastic simulations." *PLoS One*, vol. 5, no. 7, p. e11725, 2010.

[7] Q. Zhang, J. Wang, G. D. Guerrero, J. M. Cecilia, M. García, Y. Li, H. Pérez-Sánchez, and T. Hou, "Accelerated conformational entropy calculations using graphic processing units." *Journal of Chemical Information and Modeling*, vol. 53, no. 8, pp. 2057–2064, 2013.

[8] E. Mosca, P. Cazzaniga, D. Pescini, G. Mauri, and L. Milanesi, "Modelling spatial heterogeneity and macromolecular crowding with membrane systems." in *Membrane Computing*. Springer, 2011, pp. 285–304.

[9] Y. Cao, D. T. Gillespie, and L. R. Petzold, "Efficient step size selection for the tau-leaping simulation method." *The Journal of Chemical Physics*, vol. 124, no. 4, p. 044109, 2006.

[10] I. Merelli, D. Pescini, E. Mosca, P. Cazzaniga, C. Maj, G. Mauri, and L. Milanesi, "Grid Computing for Sensitivity Analysis of Stochastic Biological Models." in *PaCT*, 2011, pp. 62–73.

[11] M. Aldinucci, M. Coppo, F. Damiani, M. Drocco, M. Torquati, and A. Troina, "On Designing Multicore-Aware Simulators for Biological Systems." in *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP 2011. IEEE Computer Society, 2011, pp. 318–325.

[12] H. Li, Y. Cao, L. R. Petzold, and D. T. Gillespie, "Algorithms and Software for Stochastic Simulation of Biochemical Reacting Systems." *Biotechnology Progress*, vol. 24, no. 1, pp. 56–61, 2008.

[13] E. Mosca, I. Merelli, L. Milanesi, A. Clematis, and D. D'Agostino, "A Parallel Implementation of the Stau-DPP Stochastic Simulator for the Modelling of Biological Systems." *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, vol. 0, pp. 427–431, 2013.

[14] L. Salwinski and D. Eisenberg, "In silico simulation of biological network dynamics." *Nature Biotechnology*, vol. 22, pp. 1017–1019, 2004.

[15] H. Li and L. R. Petzold, "Efficient Parallelization of the Stochastic Simulation Algorithm for Chemically Reacting Systems On the Graphics Processing Unit." *The International Journal of High Performance Computing Applications*, vol. 24, no. 2, pp. 107–116, 2010.

[16] D. T. Gillespie, "Stochastic simulation of chemical kinetics." *Annual Review of Physical Chemistry*, vol. 58, pp. 35–55, 2007.

[17] (2013, November) cuRAND library. NVIDIA Corporation. [Online]. Available: https://developer.nvidia.com/cuRAND

[18] G. Marsaglia, "Xorshift RNGs." *Journal of Statistical Software*, vol. 8, no. 14, 2003.

[19] V. Chickarmane, C. Troei, U. A. Nuber, H. M. Sauro, and C. Peterson, "Transcriptional dynamics of the embryonic stem cell switch." *PLoS Computational Biology*, vol. 2, no. 9, p. e123, 2006.

[20] M. Kærn, T. C. Elston, W. J. Blake, and J. J. Collins, "Stochasticity in gene expression: from theories to phenotypes." *Nature Reviews Genetics*, vol. 6, no. 6, pp. 451–464, 2005.

[21] J. S. van Zon, M. J. Morelli, S. Tanase-Nicola, and P. R. ten Wolde, "Diffusion of transcription factors can drastically enhance the noise in gene expression." *Biophysical Journal*, vol. 91, no. 12, pp. 4350–4367, 2006.