Artificial Chemistries on GPU

Lidia Yamamoto, Pierre Collet, and Wolfgang Banzhaf

Abstract An Artificial Chemistry is an abstract model of a chemistry that can be used to model real chemical and biological processes, as well as any natural or artificial phenomena involving interactions among objects and their transformations. It can also be used to perform computations inspired by chemistry, including heuristic optimization algorithms akin to evolutionary algorithms, among other usages.

Artificial chemistries are conceptually parallel computations, and could greatly benefit from parallel computer architectures for their simulation, especially as GPU hardware becomes widespread and affordable. However, in practice it is difficult to parallelize artificial chemistry algorithms efficiently for GPUs, particularly in the case of stochastic simulation algorithms that model individual molecular collisions and take chemical kinetics into account.

This chapter surveys the current state of the art in the techniques for parallelizing artificial chemistries on GPUs, with focus on their stochastic simulation and their applications in the evolutionary computation domain. Since this problem is far from being entirely solved to satisfaction, some suggestions for future research are also outlined.

L. Yamamoto (✉) · P. Collet
ICUBE, University of Strasbourg, Illkirch, France
e-mail: Lidia.Yamamoto@unistra.fr; Pierre.Collet@unistra.fr

W. Banzhaf
Memorial University of Newfoundland, St. John's, Canada
e-mail: banzhaf@cs.mun.ca

1 Introduction

An Artificial Chemistry (AChem) [22] is an abstract model of a chemistry in which molecules of different types (species) interact in chemical reactions, getting converted into new molecular species in the process. Artificial Chemistries can be used to model real chemical and biological systems, to perform computations [19], and as heuristic optimization techniques [7, 37] akin to evolutionary algorithms, among other usages. Moreover AChems can be combined with evolutionary algorithms to evolve chemical reaction networks able to achieve desired goals [9, 40].

In comparison with evolutionary algorithms, the optimization process within an AChem is open to a wider range of dynamics, of which evolutionary dynamics [52] is only a special case. Furthermore, evolutionary behavior may also emerge spontaneously from carefully designed AChems [4, 5, 20, 22, 24]. In this context, AChems have been used to understand the origin of evolution from a pre-evolutionary, random initial state [20, 24], as well as to model evolutionary behavior [34, 49, 64]. The use of AChems for evolutionary optimization follows as a natural step from here: for instance, in [7] machines analogous to enzymes operate on molecules encoding candidate solutions, selecting them according to a fitness criterion, and allowing the selected ones to reproduce. As a result, an evolutionary process guided by enzymes takes place, leading to a parallel optimization algorithm whose amount of parallelism is regulated by the amount of enzymes in the system. Another example of AChems for optimization is [37], in which reaction rules select and modify molecules, driving the system from a higher entropy (more disordered) to a lower entropy (more ordered) state, in which molecules encode increasingly better solutions. A related optimization chemistry has been proposed by some of us in [75], where chemical reactions creating fitter solutions are energetically favored, and thus occur with a higher probability.

Artificial chemistries are naturally parallel: several molecules may move, collide, and react simultaneously. Moreover, algorithms for the simulation of AChems tend to be computationally expensive: a large number of molecules may be present, and many of them may react in a short time interval. Therefore the use of parallel computer architectures for implementing AChem algorithms is reasonable and can be extremely helpful to reduce computation time. In particular, with the advent of affordable Graphics Processing Unit (GPU) hardware, and their general-purpose programming, it has become attractive to implement such algorithms on top of GPUs. However, the parallelization of AChems on GPUs is not straightforward in general, mainly due to the mismatch between the synchronized processing in GPU architecture based on the SIMD (single-instruction, multiple-data) design principle and the typically fluctuating chemical dynamics where myriads of reactions may occur at any time and place.

In this survey we review the current state of the art in techniques for parallelizing AChems on GPUs, with a focus on AChems that can be used for heuristic optimization akin to evolutionary algorithms. We will see that, as the chemistry exhibits more fluctuations, that is, the stronger its stochastic behavior, the more difficult it becomes to parallelize it efficiently on a GPU. However, such stochastic AChems are often necessary in domains where interactions involving a small number of molecules must be modelled (such as gene regulation, cell signalling and other cellular processes), where noise has a qualitative impact on the behavior of the system, and, in our case of heuristic optimization algorithms, where they must rely on some amount of randomness to generate novel solutions and to explore new regions of the search space. So far the challenge of parallelizing stochastic AChems on GPUs remains far from solved to satisfaction though, therefore some suggestions for future research will be outlined at the end.

We start with an introduction to artificial chemistries (Sect. 2) and their (typically sequential) simulation algorithms (Sect. 3). Section 4 provides a brief overview of GPU architecture and programming. Parallel simulation algorithms for AChems are presented in Sect. 5, with focus on GPU algorithms. Finally, Sect. 6 discusses AChems for optimization and their potential parallelization on GPUs, and Sect. 7 concludes the chapter.

2 Artificial Chemistries

An Artificial Chemistry is formally defined in [22] as a tuple (S, R, A), where S is the set of possible molecules in the system, R is the set of reaction rules governing their molecular interactions, and A is an algorithm that determines when and how the reaction rules are applied to the molecules. Numerous algorithms exist for this purpose, as discussed in Sect. 3.

The molecules in an AChem may float in a well-stirred (or well-mixed) tank reactor (spatially homogeneous AChem), may be scattered in different regions of space (spatial AChem), or may be contained within abstract compartments akin to cellular structures. In the first case, since no spatial considerations are taken into account, all molecules share the same opportunities to encounter any other molecules within the tank reactor. In the second case, space is explicitly modelled in the system, for instance, in the form of 2D surfaces or 3D volumes; so the probability of a molecule to collide with other molecules situated in its vicinity is higher than that of bumping into molecules situated far away. In the third case, compartments may contain molecules or other compartments inside, perhaps in a hierarchical manner, but their location may remain abstract, that is, molecules and compartments are not necessarily placed in a spatial structure with an explicit coordinate system. Such differences have a large impact on the dynamics of the system as well as on the algorithms used to simulate the AChem.

When they collide, molecules may react with each other (resulting in an *effective* collision) or not (*elastic* collision). In an artificial chemistry, the reaction rule set R determines which molecular species react with which. When the collision is effective, the chemical reaction rearranges the participating atoms into the products of the reaction. Therefore the quantities of each molecular species change in the process. When the amount of molecules is very large, as happens often in a real chemistry, these quantities are typically measured in terms of concentrations of each species, that is, the amount of (moles of) a given molecular species per unit of volume. Hence, from a macroscopic perspective, a system contained in a fixed volume can be fully described by the concentration dynamics of its molecules in space and time. For a well-stirred vessel containing a sufficiently large number of molecules, such concentration dynamics can be expressed as a system of ordinary differential equations (ODEs), where each equation describes the change in concentration of one particular molecular species. This ODE system can be expressed in matrix notation as follows:

$$\frac{d\mathbf{c}(t)}{dt} = M\mathbf{v}(t), \tag{1}$$

where $d\mathbf{c}(t)/dt$ is the vector of differential equations expressing how the concentration c_i of each of the species C_i changes in time t; M is the *stoichiometric matrix* of the system, which expresses the net changes in number of molecules for each species in each reaction; and $\mathbf{v}(t)$ is a vector of rates for each reaction. The rates typically follow kinetic laws from chemistry, such as the law of mass action, or other laws such as enzyme or Hill kinetics.

In order to remain simple yet close to real chemistry, the law of mass action is often employed. This law states that in a well-stirred tank reactor, the average speed (or rate) of a chemical reaction is proportional to the product of the concentrations of its reactants [3].

For example, consider the following set of chemical reactions, representing the classical Lotka–Volterra model of predator–prey interactions in a simple ecology:

$$A + B \xrightarrow{k_a} 2A \tag{2}$$

$$B \xrightarrow{k_b} 2B \tag{3}$$

$$A \xrightarrow{\mu} \emptyset. \tag{4}$$

These reactions can be interpreted as: "species A (a predator such as a fox) eats prey B (such as a rabbit) and reproduces with speed k_a; prey B reproduces with speed k_b, after eating some nutrient C (such as grass) often assumed to be abundant enough to remain at a constant level; a predator dies with rate μ." When applying the law of mass action, such reactions lead to the following ODE system:

$$\frac{da}{dt} = k_a ab - \mu a \tag{5}$$

$$\frac{db}{dt} = k_b b - k_a ab, \tag{6}$$

where a is the amount or concentration of predator A and B the amount of prey B. Equation (5) states that the population of predators grows when predators reproduce in reaction (2) (and this happens with a rate proportional to the product of concentrations of its reactants A and B) and shrinks when predators die in reaction (4). Conversely, (6) shows that the prey population decreases when they are eaten in reaction (2) and increases when they eat some grass in reaction (3). This example can be generalized to derive the ODEs corresponding to any given set of chemical reactions in an automatic way.

The law of mass action is a simplification that considers molecules as dimensionless particles moving like gas molecules in a bottle. Nevertheless this law is still useful to model the speed of chemical reactions related to many natural phenomena.

In a spatial chemistry, besides chemical reactions, the location and movement of the molecules in space must also be modelled. Sometimes individual molecules are tracked, but more often the quantities of molecules are too large for efficient individual tracking, so the movement of macroscopic amounts of molecules must be simulated instead. Molecules may simply diffuse in space, resulting in a *reaction–diffusion* process [70], they may be dragged by fluid or atmospheric currents (resulting in an advection–reaction–diffusion process [60]), or they may be actively transported by other mechanical, electrical, or chemical forces. For conciseness we focus on reaction–diffusion processes.

In a reaction–diffusion process, molecules not only react but also diffuse in space, and this can be expressed macroscopically by a system of partial differential equations (PDEs) describing the change in concentrations of substances caused by both reaction and diffusion effects combined:

$$\frac{\partial \mathbf{c}(\mathbf{p},t)}{\partial t} = \mathbf{f}(\mathbf{c}(\mathbf{p},t)) + D\nabla^2\mathbf{c}(\mathbf{p},t). \tag{7}$$

The vector $\mathbf{c}(\mathbf{p},t)$ now refers to the concentration level c_i at time t of each chemical C_i at position $\mathbf{p} = (x, y, z)$ in space. The reaction term $\mathbf{f}(\mathbf{c}(\mathbf{p},t))$ describes the reaction kinetics, like in (1), but now expressed for each point in space. The diffusion term $D\nabla^2\mathbf{c}(\mathbf{p},t)$ tells how fast each chemical substance diffuses in space. D is a matrix containing the diffusion coefficients, and ∇^2 is the Laplacian operator.

As an example, consider now that the predators and prey in the Lotka–Volterra model may wander on a two-dimensional surface, with respective speeds D_a and D_b. The corresponding PDEs then become:

$$\frac{\partial a}{\partial t} = k_a ab - \mu a + D_a\nabla^2 a \tag{8}$$

$$\frac{\partial b}{\partial t} = k_b b - k_a ab + D_b\nabla^2 b. \tag{9}$$

The reaction part remains unchanged, while the diffusion part is represented by the last term in (8) and (9). Albeit simple, the Lotka–Volterra model is well known to display a rich set of behaviors, leading sometimes to oscillations in predator and prey concentrations, explosion of prey populations, extinction of both species, waves of predators chasing prey, clustering of species, and so on. It has been studied in a wide variety of settings, including well-mixed and spatial scenarios, as well as deterministic and stochastic settings [2, 50]. Therefore we will use this example to illustrate the various algorithmic aspects discussed throughout this chapter.

In spatial chemistries, the modelled space is often divided into small lattice sites or into larger containers (subvolumes). Lattice sites typically hold one or very few molecules, while subvolumes may potentially contain a larger number of molecules. Each subvolume can be treated as a well-mixed reactor where the law of mass action applies. Diffusion is handled as a flow of molecules between neighboring reactors, with molecules being expelled from one compartment and injected in the neighboring one.

A special case of spatial organization is a multi-compartmental AChem: in such a chemistry, a population of compartments is modelled, each with chemicals and perhaps also other compartments inside, hence allowing hierarchies of compartments to be constructed recursively. A typical example of this case is Membrane Computing or P Systems [54], a formal model of computation inspired by chemistry.

3 Algorithms for Artificial Chemistries

As stated in Sect. 2, an AChem is characterized by the tuple (S, R, A). The algorithm A determines when and how the set of reaction rules R should be applied to a multiset or "soup" of molecules M currently in the system, where each element of M is an instance of an element in S.

A naive way to implement A would be to pick a few random molecules from the soup M (simulating a molecular collision), remove them from M, perform a lookup into the rule table R for a reaction rule $r \in R$ that involves the collected molecules, apply r to these molecules obtaining the reaction products (thus simulating an effective collision), and inject the products into the soup. In case no reaction rule applies, the removed molecules would be reinserted back without change (elastic collision). In case more than one reaction rule applies, the tie could be broken by a priority scheme or simply by random selection. This process would be repeated for the desired number of time steps, or until the system reaches an inert state where none of the reaction rules in R can be applied to the molecules in M. However, when the number of molecule types and/or the number of reaction rules is large, this naive algorithm can be very inefficient, wasting too much computation time on elastic collisions. Moreover, with the naive algorithm, it is difficult to simulate reaction rates accurately in order to follow rate laws such as the law of mass action or others.

For these reasons, various algorithms have been proposed to simulate chemical reactions more efficiently and accurately, by focusing the computation effort on effective reactions. These algorithms can be classified into deterministic and stochastic simulation algorithms. The deterministic algorithms work by numerically integrating the ODEs or PDEs that describe the chemical system. The stochastic algorithms take into account individual molecular collisions and calculate which reaction should occur when scheduling the reactions and updating the molecule counts accordingly.

Our focus is on GPU implementations of AChems, and it turns out that deterministic algorithms tend to be straightforward to parallelize, as will be explained in Sect. 5. The main research challenges for the parallelization of AChems on GPUs, however, lie within the stochastic algorithms. These algorithms can be exact to the level of each individual reaction or approximate in order to trade accuracy for performance. They are reviewed below.

3.1 Stochastic Simulation of Well-Mixed AChems

In order to simulate only the effective collisions, it is useful to observe that the faster a reaction on average, the more likely it is to occur within a given time interval. Moreover, the more molecules there are in the vessel, and the more they can react, the smaller the expected time interval between any two consecutive reactions, that is, the greater the amount of reactions that might be occurring (almost) simultaneously. These two rather intuitive observations are at the heart of the famous Stochastic Simulation Algorithm (SSA) method by Gillespie [29]. This is probably the most well-known algorithm for the stochastic simulation of chemical reactions in well-stirred vessels and the basis for several improved variants that followed.

Gillespie's SSA and its variants are exact methods: they simulate each individual chemical reaction, resulting in a stochastic behavior for the whole system that accurately reflects what would occur at the level of each individual molecule. When the number of molecules is very large, and the stochastic behavior must still be considered, approximate methods are an alternative: they simulate ensembles of reactions, at a granularity that can be controlled as a parameter of the simulation.

We introduce Gillespie's SSA below, together with various related methods that will provide some useful background to discuss parallelization of such algorithms on GPUs in Sect. 5. An overview of the various algorithms in this area can be found in a recent review by Gillespie [31].

3.1.1 Gillespie's SSA

The original SSA by Gillespie as described in [29] is still widely used, since it is at the same time simple, accurate, and sufficiently efficient in many cases. The algorithm is based on the notion of *propensity* (defined formally in [31]): informally, the propensity of a reaction is a value proportional to the probability that the reaction will occur within the next infinitesimal time interval, given the current state of the system.

The pseudo-code for Gillespie's SSA is displayed in Algorithm 1. For each time step iteration, it calculates which reaction should occur and when it should

Algorithm 1 Gillespie SSA (Stochastic Simulation Algorithm) [29]

```
 1: multiset of molecules currently in the system: $M$
 2: set of possible reactions: $R$
 3: number of reactions: $m = |R|$
 4: simulation time: $t = t_0$
 5: while $desired$ do
 6:     for all $r_j \in R$ do
 7:         calculate the propensity $a_j$ of reaction $r_j$ as: $a_j = c_j h_j$
 8:         $c_j$: stochastic reaction constant for $r_j$
 9:         $h_j$: number of possible collision combinations leading to $r_j$
10:     end for
11:     $a_0 = \sum_{j=1}^{m} a_j$
12:     draw a reaction $r_j \in R$ at random (uniformly) with probability $P(r_j) = a_j / a_0$
13:     draw a random number $p$ uniformly within the unit interval (0,1).
14:     draw time interval $\tau$ from an exponential distribution: $\tau = -\frac{ln(p)}{a_0}$
15:     update current simulation time $t$ as: $t \leftarrow t + \tau$
16:     perform reaction $r_j$ by removing its educts and adding its products to multiset $M$
17: end while
```

occur. The next reaction to occur (reaction r_j) is chosen at random from a uniform distribution, with a probability proportional to its propensity a_j. The time interval τ after which the reaction occurs is also drawn at random, but from an exponential distribution with average $1/a_0$, such that the expected interval between reactions is $1/a_0$.

The stochastic constants c_j can be either given or derived from the kinetic rate coefficients using the examples from [29] or the generic formula exposed in [73]. The values h_j count the number of different ways to collide the subset of molecules in M required to perform reaction r_j, and their calculation is also explained in [29] (and generalized in [73]). In a nutshell, when the law of mass action applies, it suffices to take h_j as the product of the numbers of molecules of each type involved in r_j, and adjust c_j by a constant factor that takes into account the number of simultaneous collisions involved in r_j. In practice, collisions involve molecular pairs, and collisions of three or more molecules are very rare. Reactions involving multiple molecules usually combine several reaction steps in a single one for simplification.

Figure 1 shows one run of our implementation of the well-mixed Lotka–Volterra example using SSA with the same parameters as in [29]. As shown in [29], the ODE should remain stable in this case, while the stochastic simulation exhibits oscillations that sometimes amplify themselves.

The runtime for this algorithm scales linearly with the number of possible reactions $|R|$, as can be noticed from lines 6 to 10. Hence, although at each time step the algorithm picks only reactions that effectively occur, when the set R is large, a considerable amount of time can be spent in calculating all their propensities. In order to alleviate this problem, a number of variants of the original SSA and alternative algorithms have been proposed. We summarize them next.

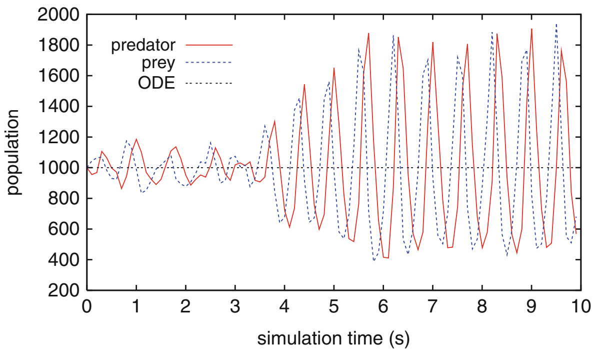

Fig. 1 Lotka–Volterra stochastic simulation using Gillespie's SSA. Reactions (2)–(4) are applied starting from initial concentrations $a_0 = b_0 = 1000$. Stochastic rate constants: $c_a = 0.01$, $c_b = 10$, $\mu = 10$ (parameters from [29])

3.1.2 Other Exact Algorithms

The original Gillespie SSA is sometimes referred to as the direct method (DM). A variant of the DM is Gillespie's First Reaction Method (FRM) [31]: instead of picking a random reaction in a propensity-proportional way, FRM draws one random τ_j interval for each reaction r_j independently, $j = 1, \ldots, m$, and executes the reaction with the smallest τ_j. The value of τ_j is calculated using the formula of line 14 of Algorithm 1 with a_j in the place of a_0. The remaining τ_j values are then discarded (because similarly to the propensity values in DM, they must be updated for the next simulation time step according to the new state of M).

The Next Reaction Method (NRM) [28] goes one step further: it sorts the reactions by increasing τ_j on a waiting list, where they are scheduled to occur at $t + \tau_j$. As the products and educts change the composition of M, only those reactions on the waiting list that were affected by the change need to be rescheduled. The waiting list is kept in the form of a binary tree for efficient lookup and update. If the number of reactions is large, and each reaction changes only a few molecules in M, NRM can significantly outperform DM. However, it is also more difficult to implement.

3.1.3 Approximate Algorithms

Whatever the simulation method chosen, simulating individual molecular reactions does not scale well to a large number of reactions, nor to a large amount of molecules. For these cases, an alternative solution is to rely on approximate algorithms. These algorithms simulate multiple reactions in a single step, trading accuracy for performance.

One of the most well-known algorithms in this category is the τ-leaping method [30]. It assumes that an interval τ can be found such that the propensities of the reactions change by a negligible amount within this interval. This assumption is called the *leap condition*: if it holds, several reactions can be fired within one simulation step (one *leap*), without updating the propensities after each individual

Algorithm 2 τ-leap [30]

```
1: let $M$, $R$, $m$, $t$ as in Algorithm 1
2: while desired do
3:     choose a suitable leap size $\tau$, for instance according to [11]
4:     $t \leftarrow t + \tau$
5:     for all $r_j \in R$ do
6:         calculate propensity $a_j$ as in Algorithm 1
7:         $\lambda = a_j \tau$
8:         draw $k_j$ from a Poisson distribution: $k_j \sim$ Poisson($\lambda$)
9:         fire reaction $r_j$  $k_j$ times, and update $M$ accordingly
10:    end for
11: end while
```

reaction. Therefore the algorithm takes a leap forward by several reactions at each time step, resulting in significant savings in computation time. The price to pay is a loss of accuracy, since the assumption of constant propensities over an interval is only an approximation.

The leap size τ plays a crucial role in the τ-leaping algorithm: it must be small enough such that the propensities do not change significantly within this interval and must also be large enough such that several reactions can be fired in a single leap, in order to save simulation time. Therefore a good choice of τ is essential but not always easy to make. A preliminary procedure for adjusting τ automatically during the simulation was sketched in [30]. Several improvements followed, with the method in [11] advised by [31] as being more accurate and faster than earlier attempts. The goal of all τ adjustment methods is to find the largest possible τ that still satisfies the leap condition. The method in [11] does that indirectly, by choosing a τ that leads to a bounded change in the relative amount of each reactant species in the system (this is faster than to calculate the propensities directly). When the leap size found is too small, the algorithm usually falls back to the baseline SSA.

Once τ has been chosen appropriately, the number of firings k_j for each reaction $r_j \in R$ is drawn from a Poisson distribution with mean and variance $\lambda = a_j\tau$. The state vector is then updated accordingly. The pseudo-code for the basic τ-leap algorithm is sketched in Algorithm 2. Figure 2 shows the prey populations for the Lotka–Volterra example now running over the τ-leaping algorithm with various leap sizes set as a function of the global propensity a_0: for instance, "leap 100" means $\tau = 100/a_0$. One can see that leap 10 still displays a good agreement with the expected SSA behavior, while leap 100 excessively amplifies the oscillations, and leap 500 leads to the premature extinction of the species. The run instances shown have been selected arbitrarily out of multiple runs, as instances that look representative of the stochastic behavior of the system under the chosen parameters. For instance, the majority of leap 500 runs lead to premature extinction, while only a few leap 100 runs lead to extinction, the majority of them just displaying a larger than normal oscillatory behavior. This illustrates the importance of setting the τ leap size properly, in order to satisfy the leap condition.

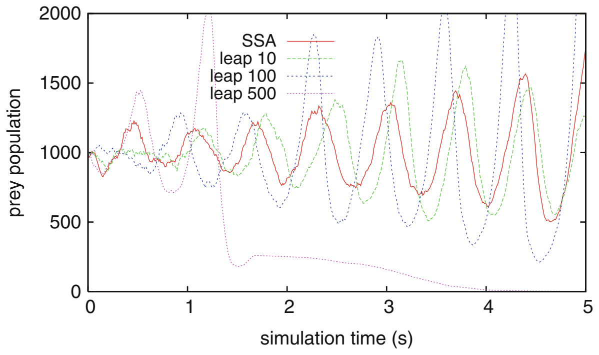

Fig. 2 Prey populations in Lotka–Volterra stochastic simulations using tau-leap with several leap sizes

Apart from setting τ properly, several other extensions of τ-leaping have been proposed. First of all, due to the potentially large leaps, Algorithm 2 may easily produce negative molecule counts if care is not taken. To solve this problem, each k_j may be constrained to a maximum by drawing it from a binomial distribution [14,69], or alternatively, only the number of firings of certain critical reactions (those with reactants approaching extinction) may be constrained [10].

Another important class of τ-leap extensions are those that deal with stiff systems [42,57]. In stiff systems, very fast reactions coexist with very slow reactions, where reaction speeds can differ sometimes by several orders of magnitude. The implicit τ-leaping method [57] deals with stiff systems by extending the implicit Euler method (used to integrate stiff ODEs) to the stochastic simulation domain. More recently, the so-called stochastic projective methods have been proposed [42], also extending upon corresponding ODE methods, with the aim of improving calculation efficiency by including a number of extrapolation steps in between leaps.

Beyond simulation performance, an important feature of τ-leaping and variants is that they can be adjusted to a wide range of behaviors, ranging from one-reaction stochastic exact simulation to a deterministic ODE integration approach: as the number of molecules approaches infinity, the stochastic variations in the Poisson distribution become negligible, and τ-leaping converges to ODE integration.

3.2 Simulating Spatial and Compartmentalized AChems

A well-mixed system is a simple but generally poor representation of a real system and does not scale well to a large number of interacting objects. Since the applications of purely well-mixed systems are limited, we now turn our attention to spatial systems. Here again, there are deterministic and stochastic simulation algorithms, and we focus on the stochastic methods.

Like their ODE counterpart, the simplest PDE integration method consists of simply discretizing δt in (7) into small fixed-sized time steps Δt: for each successive integration time step Δt, calculate the change in concentration $\Delta \mathbf{c}$ (in one time unit) for each molecular species in the system at each point in space using (7) and update the concentration vector $\mathbf{c}$ accordingly: $\mathbf{c}(\mathbf{p}, t + \Delta t) = \mathbf{c}(\mathbf{p}, t) + \Delta\mathbf{c}\Delta t$. This method relies on Δt being sufficiently small such that the coarse concentration changes Δc remain close enough to their ideal δc. It is not always easy to choose an appropriate Δt that strikes a good balance between execution time and accuracy, and this is especially problematic in the case of stiff systems. Therefore more sophisticated algorithms are available, and the interested reader is referred to [6] for an overview and further literature pointers.

As for the stochastic algorithms, since it would be too expensive to keep track of the movement of each individual molecule separately, the space is usually partitioned into equally sized *sites* or *subvolumes*, each holding a number of molecules. Each subvolume is treated as a well-mixed vessel with a given volume and a given coordinate in space. Diffusion is implemented as a unimolecular reaction that expels a molecule out of one vessel and injects it in another nearby vessel. The corresponding stochastic diffusion coefficients can be obtained from the deterministic ones by taking into account both the unimolecular reaction case (recall Sect. 3.1.1) and the volume of the compartment, as explained in [23]. Based on this idea, a number of spatial extensions of Gillespie's SSA for the stochastic simulation of reaction–diffusion systems have been proposed [23, 26, 63].

The Next Subvolume Method (NSM) [23] is one of the most well-known algorithms for the stochastic simulation of reaction–diffusion systems. It is a spatial extension of NRM, in which subvolumes are scheduled by event time as were single reactions in NRM. The event time for a subvolume is computed as a function of the total propensity of the reactions (including diffusion as a unimolecular reaction) within the subvolume, in the same way as the event time was computed for a single reaction in NRM as a function of its propensity. In each iteration, the next subvolume is picked from the top of the waiting list. From inside this subvolume, a random reaction (or diffusion instance) is chosen for firing in a propensity-proportional way as in the basic SSA (direct method). The propensities and event times for the concerned subvolumes (that is, the subvolume where the reaction occurred and the one that received a diffused molecule, if any) are updated accordingly, and the algorithm proceeds to the next iteration. Like in NRM, events are kept in a binary tree, to accelerate the search for the compartment within which a reaction or diffusion event should occur. In this way, the execution time for one iteration of the NSM algorithm scales logarithmically with the number of subvolumes and therefore represents a significant advantage over a linear search for subvolumes as would stem from a direct extension of SSA. NSM is implemented in the software package MesoRD [33], and several other algorithms are based on it.

The Binomial τ-leap Spatial Stochastic Simulation Algorithm (Bτ-SSSA) [46] combines binomial τ-leap and NSM in order to simulate longer time spans. In a nutshell, the algorithm operates as follows: subvolumes are scheduled by event time as in NSM. Whenever the propensities allow it, binomial τ-leap is used within a selected subvolume, in order to take a leap of several reactions in a single iteration. Otherwise, a single reaction is chosen within the subvolume using the basic direct SSA.

Rather than considering diffusion as a special kind of reaction, the Gillespie Multiparticle Method (GMP) [58] takes a different approach by splitting diffusion and reaction events in time, as follows: diffusion events advance synchronously in time across all sites, using a multiparticle lattice gas model. Between two diffusion events, reaction events are executed at each site independently, using Gillespie SSA, until the time for the next reaction reaches the time for the next diffusion event. Another diffusion event is then recomputed, and the procedure is repeated in the next iteration. Time between diffusion events is deterministic, but the particles diffuse to neighbors chosen at random. Due to the synchronous nature of the diffusion events, this algorithm is easier to parallelize [71], as will be seen in Sect. 5.

The Multi-compartmental Gillespie's Algorithm (MGA) was presented in [55], and improved variants thereof followed in [27, 59]. MGA is an extension of SSA to multiple compartments following a nested membrane hierarchy or P System [54]). P Systems are artificial chemistries intended as formal models of parallel computation, in which rules akin to chemical reactions are applied to objects akin to molecules enclosed in a membrane. Membranes can be nested, forming a hierarchical structure. Originally, P System rules would execute in a maximally parallel way, with no account for different reaction rates. When applied to systems biology however, a more realistic reaction timing must be taken into account, and MGA seeks to fill this gap. The algorithm is based on NRM: the events occurring in each membrane are ordered by firing time, and at each iteration, the algorithm picks the events with lowest time for firing, updating the affected variables accordingly.

In [65] Membrane Computing is used to evolve populations of artificial cells displaying growth and division. The volume of the compartments is not explicitly modelled in [65]. An extension of Gillespie for compartments with variable volume is introduced in [43], in order to simulate cellular growth and division. Indeed, the authors show that when the volume changes dynamically, the propensities are affected in a non-straightforward way, and adaptations to the original SSA are needed to accurately reflect this.

All the algorithms above assume that molecules are dimensionless particles moving and colliding randomly. However, in reality, intracellular environments are crowded with big macromolecules such as proteins and nucleic acids that fold in complex shapes. In such an environment, the law of mass action no longer applies. Simulations of reaction kinetics in such crowded spaces need different algorithms, such as presented in [61], which also show that fractal-like kinetics arise in such cases.

Recognizing that there is no perfect "one size fits all" algorithm for all possible applications in systems and cell biology, a meta-algorithm was proposed by [67]. The meta-algorithm is part of the E-Cell simulation environment and is able to run several potentially different sub-algorithms inside (such as ODE integration and NRM), in an integrated way. Time synchronization is achieved by taking the sub-algorithm ("stepper") with minimum scheduled time, in a manner similar to NRM. Such a meta-algorithm could also be interesting for simulation of multi-compartmental systems, where each sub-system may be simulated by a different algorithm.

4 GPU Computing in a Nutshell

In recent years, parallel computing on General-Purpose Graphics Processing Unit (GP-GPU) hardware has become an affordable and attractive alternative to traditional large and expensive computer clusters. Originally designed for high-performance image processing in computer graphics, movies, games, and related applications, the popularity of GPUs has reached domains as diverse as scientific computing for physics, astronomy, biology, chemistry, geology, and other areas; optimization and packet switching in computer networks; and genetic programming and evolutionary computation, among other domains [8, 17, 32, 44].

In this section we summarize the GPU architecture and programming very briefly, just enough for the reader to be able to follow the discussion on the parallelization of the AChem algorithms in Sect. 5. See [45] and chapter 2 of this book for a more comprehensive overview of GPU hardware and [17] for a more comprehensive overview of GPU computing applied to the modelling of biochemical systems.

Initially difficult to program due to its specialized internal architecture, GPU cards are now becoming increasingly easier to program, thanks to high-level programming frameworks such as CUDA (Compute Unified Device Architecture, by the GPU card manufacturer NVIDIA) and OpenCL (Open Computing Language, a framework designed to execute over multiple GPU platforms from different manufacturers). However, in many aspects GPU programming still remains rather low level and architecture dependent: in order to fully exploit the parallelism provided by GPUs, the programmer needs to know the internals of the GPU architecture very well and design an efficient program accordingly. Moreover, not all tasks can fully benefit from the parallelism provided by GPUs: GPUs have a SIMD (single-instruction, multiple-data) architecture, in which each single processor is able to process multiple data items in parallel using the same instruction. Therefore, the tasks that are good for GPUs are those that must handle multiple data items using the same flow of instructions.

In a nutshell, the GPU architecture is organized as follows: each GPU device contains a grid of multiprocessors (typically between 15 and 30). Each multiprocessor is organized as a set of SIMD processors. Each SIMD processor can handle a number of data items in parallel, typically 8 or 32. All the processors share a global memory space. Each multiprocessor has a local memory space that is not visible to other multiprocessors, but that can be shared among its own SIMD processors, and is called the shared memory space.

The CUDA framework tries to hide the internal organization of a GPU device, while exposing the aspects that are necessary for the programmers to optimize their algorithms to run efficiently on the GPU. As such, each card is structured as a grid of blocks. Blocks are scheduled to multiprocessors, and each block may run a certain number of threads in parallel (typically 512 or 1,024). These threads are mapped to SIMD processors in a preemptive way: each SIMD processor can handle up to N threads in parallel (N is called the *warp size* and is typically 32), provided that they all run the same instruction on different data items. If a processor gets a group of threads in which half of them is doing something different than the other half (this is called *thread divergence* and typically occurs during conditional branching), then the processor must run one group of threads first, then the second, in sequence, therefore increasing the overall execution time needed to complete the multithreaded task. Therefore, avoiding thread divergence is one technique to help improving the performance of GPU programs.

Other techniques to improve the performance have to do with memory management: the access time to global memory items is much slower than to items placed in shared memory. Therefore, placing frequently used data items on shared memory can improve performance significantly. On the other hand, these local memory items are deallocated when the GPU call returns to the host machine's CPU; therefore, they have to be reinitialized at each GPU iteration call, typically by copying them from global to local memory. Since the global memory space is much larger and data items stored there persist across GPU invocations, its use is often very convenient. A good technique to improve its access time is to retrieve items in groups of contiguous memory positions (coalescent memory access). Another important aspect to consider is the communication cost between CPU and GPU: passing data items from the CPU to the GPU and vice versa is an expensive operation and therefore should be minimized.

The synchronization among threads running on a GPU also deserves attention: threads within the same block can synchronize together, while threads in different blocks cannot. Blocks may be scheduled and preempted at any order; therefore, inter-block synchronization is problematic. It is usually achieved by returning control back to the host and paying a corresponding performance penalty.

In order to run a given program on the GPU, a so-called kernel function must be specified, in which the code for each thread is written, usually as a function of the data items that each thread should handle. After transferring all the necessary data items from the host CPU to the GPU card's global memory, the kernel is then invoked with the grid and block dimension parameters, together with any function parameters needed for the kernel to run, including the locations of the global data items to be processed. After completion, the processed data items are then transferred to the CPU where the computation results can be extracted for further analysis.

The performance of an algorithm running on GPU is usually measured in terms of the speedup of the GPU implementation with respect to a single conventional CPU (single core). In this context, a speedup of $\times 10$ (or $10\times$) means that the parallel algorithm runs ten times faster on the GPU than the corresponding sequential algorithm on a single CPU. Note that speedup figures must be interpreted with care, since they depend on the actual GPU and CPU models used in the measurements.

5 Parallelizing Artificial Chemistries on GPUs

A survey of the use of GPUs for the simulation of biological systems is presented in [17]. Several algorithms are described, including Molecular Dynamics (MD) simulations, lattice-based methods such as cellular automata (CA), multiparticle diffusion models, reaction–diffusion, and P Systems on GPU. A survey of related algorithms for parallel architectures in general can be found in [6], covering ODE integration, as well as the sequential and parallel stochastic simulation of chemical reactions. Here we focus on the chemistry part, and for this reason we do not cover methods that go down to the physics of the system, simulating molecular shapes and movements, such as MD and particle-based methods. We refer the interested reader to [17] for an overview of these other methods in the GPU context.

Among the AChem-related algorithms, numeric PDE integration, cellular automata, and other spatially oriented algorithms are the easiest to parallelize on GPUs, due to their data structure resemblance to those of the image processing tasks for which GPU hardware was originally conceived. Section 5.1 provides a brief overview of some existing approaches to the parallelization of deterministic AChem algorithms on GPU.

The parallelization difficulties increase as we move away from such highly repeated data structures with identical data handling and approach stochastic algorithms in which multiple different reactions may take place at different times. Section 5.2 provides an overview of the existing approaches to parallelize stochastic algorithms on GPU, and Sect. 5.4 points out the main remaining difficulties and potential improvements.

5.1 *Deterministic Algorithms*

The parallelization of ODE and PDE integration on GPUs is generally straightforward and has been applied to solve problems in systems biology and other domains [51, 60]. This is especially true for the case of numeric PDE integration, where the same differential equations and diffusion rules apply to all the points of the grid, providing a nearly perfect match to the GPU architecture: each GPU thread can take care of one point in space, and the threads in a warp can perform the same computation on different data points.

A GPU parallelization of the numeric integration of reaction–diffusion equations in three-dimensional space is described in [51]. A detailed overview of GPU approaches to parallelize the numeric integration advection–reaction–diffusion equations is presented in [60].

In previous work [77], we used a parallel implementation of reaction–diffusion on a GPU to look at large patterns, and in [76] we complemented such a GPU implementation with an evolutionary algorithm in order to search for reaction–diffusion systems forming desired patterns. In both cases the parallelization of reaction–diffusion and their automatic evolution on GPUs achieved speedups of about two orders of magnitude compared to the single CPU case and was essential to make the experiments run within a feasible duration.

5.2 Stochastic Algorithms

Several efforts to parallelize stochastic algorithms for AChems can be found in the literature, with various degrees of success. This section briefly reviews these efforts, starting with exact methods for well-mixed systems, moving on to approximate methods and spatial and compartmental systems, and finally citing some very recent work.

In [41] multiple instances of Gillespie's SSA are launched in parallel on the GPU, in order to repeat a given experiment several times. A variant of Gillespie's SSA called the logarithmic direct method (LDM) with sparse matrix update is used in order to improve performance. Speedups of up to $\times 200$ are reported. Such good performance is obviously expected given that the SSA algorithm itself is not parallelized, so no global state needs to be maintained. Similar parallelization strategies can be found in software packages such as AESS [35] and CUDA-sim [78].

A parallelization of Gillespie's FRM on GPU is proposed in [18]: the calculation of the smallest τ interval is partitioned among several tasks on the GPU, each of which takes care of a group of chemical reactions and calculates its local minimum τ value accordingly. All the local minima are collected in order to compute the global minimum, which is then used to compute the next state of the algorithm on the CPU. Several GPU-specific technical optimizations are also included in order to improve the performance of the algorithm. Despite the careful design, a weak performance gain of less than $\times 2$ speedup is reported, which can probably be attributed to the excess of synchronization needed between GPU threads and between CPU and GPU in order to compute and maintain the global state of the system.

A parallel implementation of τ-leap on GPUs is presented in [74], extending upon a previous method called parallel Coarse-Grained Monte Carlo (CGMC) for the simulation of spatially distributed phenomena on multiple scales. CGMC partitions the space into cells akin to subvolumes. Only three types of events are considered: diffusion, adsorption and desorption of molecules on cell surfaces. In parallel CGMC, the τ-leap method is extended from a well-mixed to a spatial context. A master-slave configuration is used for this purpose: a master node calculates τ and broadcasts it to the slave (cell) nodes that compute propensities and fire reactions. The locally updated molecular populations are returned to the master who collects all the values and updates the global state for the next iteration. The parallelization of CGMC on GPU [74] also works in the coarse-grained spatial context of CGMC: each GPU thread (slave) takes care of one cell and performs the leaps for the reactions inside the cell, given the interval τ provided by the master node (CPU). Experiments show that simple parallelization strategies, including the use of shared memory for storing local propensities and molecule counts, perform better on large systems than more sophisticated strategies based on a multilayered structure.

An implementation of P Systems on GPU with CUDA is shown in [12]. Membranes are assigned to blocks on the GPU, where threads apply the rules to the objects inside the membrane. Although impressive speedup figures are reported in [12], reaching values in the range of $1,000\times$, the experiments used to obtain these figures included only very simple reaction rules, essentially to refresh and to duplicate objects. These rules were executed in a maximally parallel manner, therefore obviously making full use of the GPU to perform the same operation on multiple data items as fast as possible. A more realistic case study of P Systems on GPUs is presented in [47], where an instance of the N-Queens problem (a well-known NP-hard problem) is solved with the help of a P System running on a GPU card. In [47], only the selection of the rule to be fired is done on the GPU, while the actual rule execution is left for the CPU. In this context, a speedup of about $12\times$ is reported for the selection part on GPU, with respect to the corresponding selection on CPU.

As mentioned in Sect. 3.2, the GMP algorithm is easier to parallelize because diffusion events occur in a synchronous way. Recently, the GMP algorithm was indeed parallelized on a GPU (and GPU cluster), resulting in the GPGMP algorithm [71]. GPGMP consists of a main loop on the CPU, from where three GPU kernels are invoked: first, the Gillespie kernel computes the chemical reactions according to the plain SSA; afterwards, the Diffusion kernel decides which molecules will move in which direction; finally, the Update kernel is invoked to update the molecule counts for each subvolume; and then the loop repeats, with a central update of the simulation time flow on the CPU. Speedups of up to two orders of magnitude are reported but on a GPU cluster and on very simple examples where all processors run the same kind of reaction. Later, the diffusion part of GPGMP was extended to support inhomogeneous diffusion [72], also on a GPU.

At the time of this writing, the most recent algorithm to parallelize stochastic simulations on GPUs is [38]. It parallelizes well-mixed τ-leaping on GPUs with the help of the NVIDIA Thrust library, which provides convenient parallel operations on vectors. The calculation and update of propensities, the random choice of the number of firings k_j for each reaction, and the update of the state vector are done in parallel on the GPU. A speedup of up to $60\times$ is reported with respect to an optimized sequential SSA implementation. A comparison against a sequential implementation of τ-leaping would be useful to assess the actual performance gain of the parallelization procedure more clearly.

The algorithm in [38] also includes a parallel version of SSA (direct method), used when τ-leaping must fall back to SSA. At each time step, this parallel SSA performs the reaction selection procedure, the update of propensities, and the update of the molecule counts in parallel. The computation of τ, followed by the update of the simulation time, is done by the CPU. This method is therefore efficient in systems with large numbers of reactions and molecular species, in cases where

τ-leaping cannot be applied due to a violation of the leap condition. However the performance of the parallel DM part in isolation is not reported in [38].

5.3 A Simple Stochastic AChem on GPU

As an example to illustrate the issues involved in parallelizing AChems on GPUs, we have implemented a simplified version of a spatial stochastic AChem on GPU. The algorithm combines some elements of GPGMP [71] and the GPU-based CGMC [74]. It supports both SSA and τ-leap for the reaction step, while the diffusion step is preferentially stochastic. Diffusion can also be switched off, in order to run multiple independent instances in parallel as in [35, 41, 78].

For simplification, we ignore the most difficult problem with τ-leap so far, namely, the adaptive nature of τ, and adopt a fixed τ interval that is used by all processors during the full duration of the simulation. In multiple-instance mode (without diffusion), the leap size can also be chosen as a multiple of the propensity a_0 for each instance (Sect. 3.1.3). As with the example of Fig. 2, adjusting τ manually allows us to illustrate the trade-off between speed and accuracy in the algorithm.

Algorithm 3 shows how our implementation works. Three kernels are needed: one for the reaction component (lines 5–8) and two for the diffusion component (lines 9–22 and 23–28, respectively). At each iteration, these kernels are invoked sequentially from the CPU, in order to choose the amount of reactions to be fired and of molecules to be diffused during an interval τ. Each kernel launches a number of threads in parallel; each thread takes care of one lattice cell (equal to one subvolume or one compartment). Uniformly distributed random numbers are generated using the CURAND library under CUDA, and Poisson-distributed numbers are derived from these upon demand, using a logarithmic variant of the basic Knuth's method for small lambda ($0 < \lambda \leq 10$) and a Gaussian approximation with continuity correction for $\lambda > 10$. For simplification, no special memory access optimizations are implemented, and the molecule counts for each compartment are stored in global memory.

The first kernel takes care of the reaction part. Each thread simply invokes SSA or τ-leap within its compartment. In the case of SSA, multiple iterations are involved until an interval τ is simulated. In the case of τ-leap, a single iteration with step τ is invoked; negative molecule counts are avoided by simply ignoring reactions that produce them.

The diffusion component is divided into two kernels in order to solve the critical region problem arising from the need to transport molecules from one compartment to another (controlled by another thread). The first diffusion kernel decides how many molecules go to which neighbor positions. This is done by treating diffusion as a unimolecular reaction (as in NSM) and then drawing the number of firings for this pseudo-reaction from a Poisson distribution (as in τ-leap). In this way, each thread computes the amount of molecules to be transported, subtracts this amount

Algorithm 3 Stochastic Reaction–Diffusion on GPU

```
 1: $\tau$: time step interval
 2: $t = t_0$
 3: while desired do
 4:     $t \leftarrow t + \tau$
 5:     for all threads on GPU in parallel do
 6:         run $\tau$-leap (Algorithm 2) for compartment
 7:         or alternatively, run SSA (Algorithm 1) until $t$ is reached
 8:     end for
 9:     for all threads on GPU in parallel do
10:         $N$: set of neighbors of this cell in the lattice
11:         $M$: multiset of molecules within this cell
12:         for all $s_i \in M$ do
13:             $B_{i,j} = 0 \; \forall j \in N$
14:             $d_i$: stochastic diffusion coefficient of species $s_i$
15:             if $d_i > 0$ then
16:                 $n_i$: number of molecules of type $s_i$ in subvolume
17:                 $\lambda = d_i n_i \tau$
18:                 draw $k \sim$ Poisson($\lambda$)
19:                 $B_{i,j} = k$, where $j$ is a random neighbor of this cell
20:             end if
21:         end for
22:     end for
23:     for all threads on GPU in parallel do
24:         $N$ = set of neighbors of this cell
25:         for all $B_{i,j}$ from $N$ where $j$ is the index of this cell do
26:             $s_i \leftarrow s_i + B_{i,j}$
27:         end for
28:     end for
29: end while
```

from its local count, and writes it to a transport buffer B, indexed by molecular species and destination compartment. After all the threads have completed the first kernel, the second kernel is then launched. During this kernel, each thread inspects the neighboring transport buffers, and increases the molecule counts for each species destined to its position, by the amount given in the corresponding buffer position.

When the population of diffused molecules is large enough, the diffusion step can be easily made deterministic by taking $k = \lambda$ on line 18. A deterministic diffusion step combined with SSA within each compartment would turn Algorithm 3 essentially into GPGMP. The combination of stochastic diffusion with τ-leap leads to a variation of CGMC on GPU where any type of chemical reaction can be supported. In multiple-instance mode, the two diffusion kernels are simply not invoked.

Figure 3 shows some snapshots of the spatially extended stochastic Lotka–Volterra example run on a lattice of 128×120 cells (this is the minimum lattice size for which the GPU card is fully loaded with one cell per thread). The τ-leaping algorithm was used for the reaction step in Algorithm 3, with a leap interval of $\tau = 0.01$ s. For better visibility, the color intensities have been normalized relative

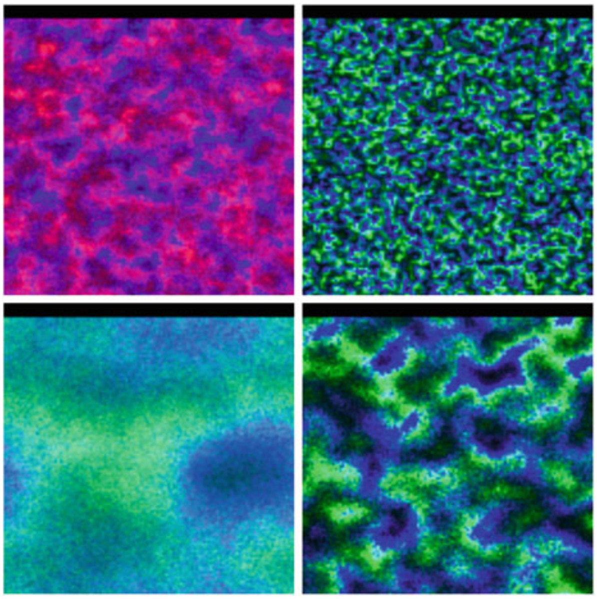

Fig. 3 Snapshots of predator (*red* or *green*) and prey (*blue*) populations in spatial stochastic Lotka–Volterra simulations with varying diffusion coefficients (D parameter) for the predator and prey species. *Top*: $D = 1.0$. *Bottom*: $D = 10.0$. Snapshots taken at the end of the simulation (at $t = 100$ s of simulated time)

to the cell with the highest amount of chemicals (depicting therefore the relative rather than absolute population values). When there is very little or no diffusion (not shown), either the predator or prey populations quickly go extinct, and no patterns are observed. For diffusion coefficients smaller than $D = 1.0$, the prey and predator populations are too scattered (due to extinct regions) or too mixed, so no pattern is formed. For the case of $D = 1.0$ (upper part of Fig. 3), some loose segregation of predator and prey populations in small clusters starts to become apparent. The clustering phenomenon seems to increase with increased diffusion: for instance, for $D = 10.0$ shown in the lower part of Fig. 3, irregular but clearly visible patterns seem to form, including dark areas with scarce populations. The position and shapes of the clustered areas change very quickly in time, but the overall qualitative behavior tends to persist throughout the simulation. However, beyond a certain diffusion rate (for instance, for $D = 100$, not shown), the system reverts back to the high extinction rates observed in the original case without diffusion, indicating that we approach the well-mixed case for the whole lattice.

Figure 4 shows that extinction can be significantly delayed by adding a small amount of diffusion to the system. Extinction here means that the population of either predator or prey gets depleted in a given cell. The fraction of extinct cells is then the fraction of cells that have either population extinct in its local compartment. The diffusion coefficient was set to the same value for both predator

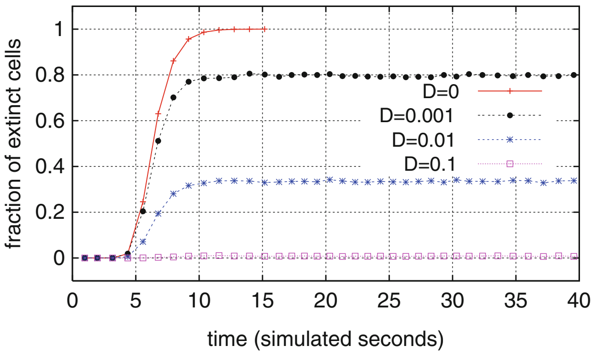

Fig. 4 Fraction of extinct cells in a spatial stochastic Lotka–Volterra simulation, for varying diffusion coefficients

and prey species. When no diffusion is present ($D = 0$), the system is reduced to the multiple-instance mode, without interaction between compartments. In this case, a quick extinction rate is observed. Moreover, the simulation stops at around $t = 15$ s due to the extinction of all cells on the grid. In contrast, by adding an amount of diffusion for predator and prey as low as $D = 0.001$, the total collapse of the simulation is avoided: the global population remains stable and is able to survive till the end of simulation (at $t = 100$ s, although the plot is truncated at $t = 40$ s for better visibility), in spite of a large number of extinct cells. Further increasing diffusion also causes a further drop in extinction rate, until it reaches zero at $D = 1.0$. The extinction rate is kept at zero for $D = 1.0$ until about $D = 10$ (not shown). A further increase in diffusion speed leads again to faster extinction, until a scenario similar to $D = 0$ is achieved (not shown). This return of the danger of extinction with increased diffusion can be delayed to higher diffusion coefficients by decreasing the global time step τ (thus increasing the accuracy of the simulation). Hence these results must be interpreted with care and in a qualitative rather than quantitative way.

Since the focus of this chapter is on the GPU parallelization of stochastic AChems, we will not delve further into the details of this specific Lotka–Volterra example. See [2, 50] for more information about stochastic predator–prey systems and [62] for an early parallel implementation thereof. The results shown in this section seem in line with the known literature in the area; however, the fixed τ step size has been carefully chosen to capture the relevant qualitative aspects of this specific example. For a more general usage, in order to avoid the computational load of adjusting τ globally, an alternative could be for each thread to choose independently whether to apply SSA or τ-leap within the given τ interval, for instance, by falling back to SSA when the leap condition cannot be satisfied in its local compartment. However this could lead to thread divergence when neighboring compartments (executed by the same SIMD processor) choose different algorithms. Once more, such options express the trade-off between simulation accuracy and computational efficiency.

Concerning computational efficiency, Table 1 shows the average speedups achieved for our GPU implementation of SSA and τ-leap, for a leap size of

Table 1 Speedups obtained for our implementations of Gillespie SSA and τ-leap on GPU (for $\tau = 10/a_0$), averaged over 100 runs

	GPU	
CPU↓	SSA	τ-Leap
SSA on CPU	60.7 ± 2.1	120.0 ± 4.0
τ-Leap on CPU	23.1 ± 0.4	45.7 ± 0.8

$\tau = 10/a_0$ in multiple-instance mode. The speedup is calculated over a runtime of 10 simulated seconds of the Lotka–Volterra example under the same initial conditions and parameters as in Fig. 1 but with different seeds for random number generation. We can see that the GPU always outperforms the CPU (speedups are always greater than one), even in the less obvious case of the GPU running SSA against the CPU running τ-leap. Obviously, the best speedup is achieved under the most favorable conditions: when the GPU runs τ-leap and the CPU runs SSA (this was also the comparison used by [38]). Note that in spite of a leap size ten times larger than the average SSA time step interval, τ-leap only reaches twice the speed of SSA. This is mainly due to the higher cost of generating Poisson-distributed random numbers for each reaction, when compared to the two uniform numbers needed at each SSA iteration. Higher speedups can be achieved with τ-leap by increasing the leap size, however, this also makes the algorithm less accurate as discussed in Sect. 3.1.3.

We have also measured the share of the diffusion part of Algorithm 3 over the total simulation time. This was done for the GPU implementation only (due to the slowness of running the spatial algorithm on a single CPU), for the same spatial Lotka–Volterra example and τ-leap with $\tau = 0.01$. Figure 5 shows the runtimes together with the fraction of time spent on the diffusion process alone (consisting of the two diffusion kernels in Algorithm 3). Diffusion takes a considerable share of the total runtime, up to about two thirds, with a peak coinciding with the parameter regions where cells tend to survive until the end of the simulation (at $t = 100$ s) (since cells with extinct populations have nothing to diffuse thus consume no computation time). Part of the responsibility for such a high computation load might be attributed to the required return to the CPU in between the two diffusion kernel invocations, plus the necessity for each cell to inspect neighbor information for local state updates (which increases memory access delay since some neighbors might not be located in nearby global memory positions allowing coalescent memory access). More efficient memory access schemes could be used to optimize the diffusion part, such as those suggested in [38, 74].

5.4 Summary and Discussion

The original Gillespie SSA is known to be hard to parallelize [17]. Indeed, it can be seen in Algorithm 1 that the global state of the system is refreshed at every time step: the propensities of all reactions are summed up to obtain the value a_0 which is then used both to calculate the time τ until the next reaction and to pick a random

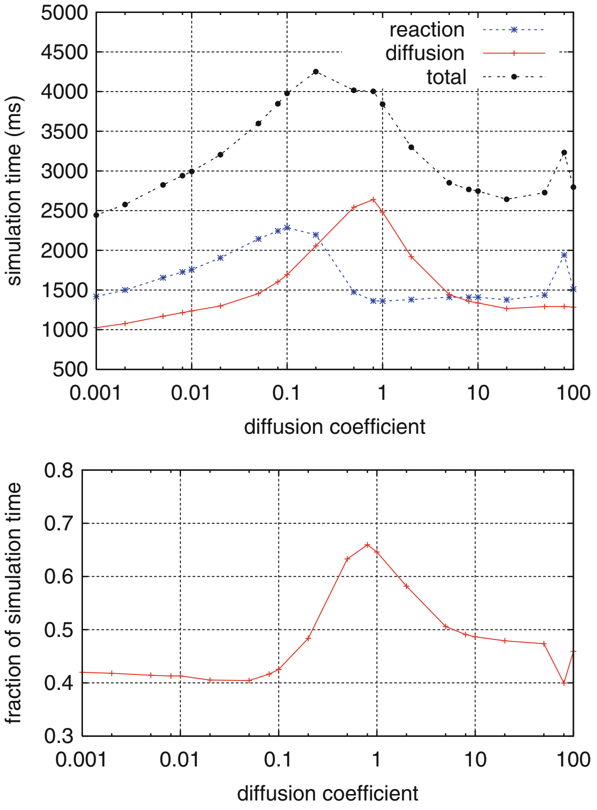

Fig. 5 Measured GPU runtimes for varying diffusion coefficients for predator and prey. *Top*: Runtime for the reaction part, diffusion part, and total runtime. *Bottom*: Fraction of the total runtime spent in the diffusion process alone

reaction to be fired, with a propensity-proportional probability. Here the global state of the system is represented by variables a_0 and τ, and their regular update can be regarded as a compulsory synchronization point in any parallel implementation of the algorithm. Such global state maintenance is what makes the algorithm difficult to parallelize: the best algorithms for parallel implementation are those that can be easily split into independent or loosely connected parts, such that each processor is able to operate alone and correctly without global state information, relying only occasionally on data exchanges with other processors.

Methods to parallelize the original SSA and other exact methods on GPU focus either on the realization of multiple instances of the algorithm in parallel [35, 41, 78] or on the parallelization of the steps to reach the global synchronization points mentioned in the previous paragraph [18, 38]. Such synchronization must be done at every iteration, usually requiring a round trip from the GPU to the CPU for that purpose. The performance of such parallel algorithms is therefore limited by these frequent synchronization events.

The parallelization of approximate methods such as τ-leaping sounds easier: once the step τ is calculated, all the k_j values for the different reactions may be chosen in parallel. In practice, however, the procedure to calculate τ can be computationally expensive, the time to generate poisson or binomial random numbers (needed to obtain the various k_j) may increase with the propensities, and the choice of each k_j value may affect the others, due to substances that participate in multiple reactions. Despite these difficulties, recently new algorithms that parallelize τ-leaping on GPUs have been proposed [38, 74]. Here again, the update of τ represents the global synchronization point that sets a limit on the amount of parallelism that can be achieved.

The parallelization of spatial methods looks even easier: subvolumes can be easily assigned to processors or threads that can run in parallel, occasionally exchanging molecules. However, here again time must be synchronized globally across all processors, since the speed of reactions happening inside each subvolume has an impact on the overall behavior of the system, for instance, on the patterns that may form within it. The GPU implementations of τ-leap [74] and GMP [71] both occur in a spatial chemistry and are examples of this category.

When time constraints are not an issue, impressive speedup figures can be obtained, for instance, when P Systems run on GPU using a maximally parallel way [12]. Since time synchronization is such a critical obstacle to parallelism, some authors [16, 36] have attempted to get around it with techniques from distributed systems applicable to discrete event simulations, essentially based on the rollback of events (undo). This method presents excessive overhead due to the amount of messages that must be exchanged between processors and the amount of events that must be undone. In the case of complex or irreversible chemical reactions with potential side effects, undo can be both computationally expensive and problematic. Needless to say, due to its message passing model, such a technique does not match the GPU architecture very well.

Load balancing is also an issue with any GPU implementation of a stochastic AChem. In order to make full use of the GPU resources, thread divergence must be avoided or at least minimized. Therefore when reactions are executed in parallel, ideally similar reactions must be grouped by thread warps, such that each warp executes a nearly identical code, minimizing divergence. Although promising, algorithms such as GPGMP [71] and the variants of τ-leap on GPU [38, 74] all have the potential problem that thread divergence in reaction execution might occur in the case of complex reaction networks composed of very different reactions. A load balancing strategy that takes into account the GPU architecture is needed. One possible strategy could be to assign groups of reactions to threads based on their similarity and the computation load required to fire them: for instance, one thread warp could receive a group of a few similar but difficult reactions, while another warp would get a group of many similar and easy reactions. How to design an efficient load balancing strategy with minimum computation overhead remains an open issue.

A technical issue of practical importance is the availability of random number generators on GPU. The CURAND library has been recently released, offering a range of pseudorandom number generators on GPUs for CUDA. Before that, researchers used their own homemade random number generator, usually a variant of the well-known Mersenne Twister algorithm [18, 41] and related pseudorandom number generators [39, 68]. Other researchers have exploited the inherent parallelism of the GPU to obtain pseudorandom number generators based on cellular automata [53], as well as true-random number generators that exploit natural sources of randomness on the GPU such as race conditions during concurrent memory access [13]. The computation efficiency of number generators other than uniform (Gaussian, Poisson, binomial) as needed by AChem algorithms still needs to be further assessed and improved on GPUs.

6 AChems for Search and Optimization

Looking at optimization from an Artificial Chemistry perspective is equivalent to explicitly modelling the optimization process as a dynamical system: candidate solutions can be regarded as molecules, and variation can be regarded as a chemical reaction resulting in the transformation of one or more molecules into mutant or recombinant types [7, 37, 75]. Various types of selection pressure may be applied. Two commonly used methods in AChems are inspired by prebiotic evolution: the first one is to kill a random individual whenever a new one is created, and the second one is to apply a dilution flow that randomly discards molecules when the maximum vessel capacity is exceeded [20, 22]. Such non-selective random elimination of individuals leads nevertheless to a selection pressure that favors molecules able to maintain themselves in the population either by self-replication or by being regenerated by others in self-maintaining chemical reaction networks akin to primitive metabolisms [5, 21]. Rather than preprogramming an evolutionary behavior like a genetic algorithm, such AChems favor the emergence of evolution out of molecular interactions in chemical reactions. For instance, the spontaneous emergence of a crossover operator is reported in [20].

Although most of the work in the AChem literature focuses on the dynamics of prebiotically inspired chemistries and their evolutionary potential, without an external objective function to be optimized, such studies are complementary to current effort in evolutionary computation, since they can shed light on the underlying mechanisms of evolution that could potentially be applied to improve or to create new optimization algorithms.

Evolving populations tend to exhibit stiff system dynamics: some mutations might cause waves of change that sweep through the populations, followed by periods of low activity. With some adaptations, the stochastic algorithms discussed in Sect. 3 can also be used to simulate evolutionary dynamics: a hybrid algorithm based on SSA and τ-leap is introduced in [79] and applied to the simulation of evolutionary dynamics of cancer development. The Next Mutation Method [48] is another recent algorithm for simulating evolutionary dynamics. Inspired by NRM and taking into account that mutations are rare events, it aims at reducing computation effort by jumping from one mutation to the next.

As a model of a simple ecology, the Lotka–Volterra example can be naturally extended to an evolutionary context. Indeed, generalized predator–prey systems involving multiple species have been proposed, including cyclic interactions (the predator of one species is the prey for another, and so forth, forming a food chain in an ecosystem), as well as mutations of one or more species, leading to adaptations in individual behavior [1, 25, 66]. In such models, predator and prey species coevolve: for instance, predators may evolve an improved ability to track and capture prey, whereas prey may evolve more efficient escape strategies.

Coevolutionary optimization algorithms [56] have been inspired by the competitive arms race resulting from natural coevolution. Recently, a spatial coevolutionary algorithm inspired by predator–prey interactions has been proposed [15], in which species evolve on a two-dimensional grid in order to solve a function approximation problem. Niches of complementary partial solutions emerge, leading to local specializations for cooperative problem solving, which nevertheless result from competitive predator–prey interactions.

Although promising, the potential of coevolution for optimization remains under-explored, mainly due to the complex dynamics emerging from species interactions. Perhaps this is an example where artificial chemistries running on top of GPUs could help, both to better understand such dynamics and to derive improved algorithms from such knowledge.

To the best of our knowledge, the parallelization of algorithms such as [15, 48, 79] on GPU has not been attempted so far. The simulation of complex ecologies and their evolution seems to be an area where the use of GPU acceleration could bring significant benefits due to the large population sizes involved, their complex interaction patterns, and their potential for an open-ended evolutionary process.

7 Conclusions

The main challenge in parallelizing AChems on GPUs is to parallelize the stochastic algorithms. These algorithms often rely on centralized information such as the total propensity of all reactions in the system and the time interval between reactions, which influence the choice of the next reaction, when it should occur, or how many times it should be fired. Moreover, they require frequent use of random number generators, a facility that only recently became available as a CUDA library.

However, parallelizing such stochastic algorithms is of paramount importance, since these algorithms tend to be computationally intensive and are important when the simulation of reactions is needed, which is often the case in artificial chemistry studies related to synthetic biology, artificial life, and evolution.

In this survey we have shown the state of the art in artificial chemistries on GPUs, discussed applications, and exemplified the usability of recently proposed GPU algorithms for stochastic spatial predator–prey systems. We have highlighted the main issues involved in the efficient parallelization of such algorithms, with attention to their application in the optimization domain. Although many problems remain to be solved, as GPU programming becomes increasingly easier, it is expected to contribute to significant advancements in the understanding of evolutionary and coevolutionary processes in models of natural ecologies or for devising new optimization algorithms able to tackle large and complex problems.

Acknowledgements We would like to thank the University of Strasbourg and the Region d'Alsace for funding a postdoctoral stay of L. Yamamoto and a visiting stay of W. Banzhaf under the EVOL grant. W. Banzhaf also acknowledges support through the NSERC Discovery Grant program under RGPIN 283304-12.

References

1. Abrams, P.A.: The evolution of predator–prey interactions: theory and evidence. Annu. Rev. Ecol. Systemat. **31**, 79–105 (2000)
2. Andrews, S.S., Bray, D.: Stochastic simulation of chemical reactions with spatial resolution and single molecule detail. Phys. Biol. **1**(3), 137–151 (2004)
3. Atkins, P., de Paula, J.: Physical Chemistry. Oxford University Press, Oxford (2002)
4. Bagley, R., Farmer, J., Fontana, W.: Evolution of a Metabolism. In: Artificial Life II, pp. 141–158. Addison-Wesley, Reading (1991)
5. Bagley, R.J., Farmer, J.: Spontaneous Emergence of a Metabolism. In: Artificial Life II, pp. 93–140. Addison-Wesley, Reading (1991)
6. Ballarini, P., Guido, R., Mazza, T., Prandi, D.: Taming the complexity of biological pathways through parallel computing. Brief. Bioinform. **10**(3), 278–288 (2009)
7. Banzhaf, W.: The "molecular" traveling salesman. Biol. Cybern. **64**, 7–14 (1990)
8. Banzhaf, W., Harding, H., Langdon, W.B., Wilson, G.: Accelerating genetic programming on graphics processing units. In: Riolo, R., Soule, T., Worzel, B. (eds.) Genetic Programming Theory and Practice VI, GEC Series, pp. 229–248. Springer, New York (2009)
9. Banzhaf, W., Lasarczyk, C.: Genetic programming of an algorithmic chemistry. In: O'Reilly, et al. (eds.) Genetic Programming Theory and Practice II, Chap. 11, vol. 8, pp. 175–190. Kluwer/Springer, Dordrecht/Berlin (2004)
10. Cao, Y., Gillespie, D.T., Petzold, L.R.: Avoiding negative populations in explicit Poisson tau-leaping. J. Chem. Phys. **123**, 054104 1–8 (2005)
11. Cao, Y., Gillespie, D., Petzold, L.: Efficient step size selection for the tau-leaping simulation method. J. Chem. Phys. **124**, 044109 1–11 (2006)
12. Cecilia, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P Systems with active membranes on CUDA. In: IEEE International Workshop on High Performance Computational Systems Biology (HIBI), pp. 61–70 (2009)
13. Chan, J.J.M., Sharma, B., Lv, J., Thomas, G., Thulasiram, R., Thulasiraman, P.: True random number generator using GPUs and histogram equalization techniques. In: Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications (HPCC), pp. 161–170. IEEE Computer Society, Washington (2011)
14. Chatterjee, A., Vlachos, D.G., Katsoulakis, M.A.: Binomial distribution based τ-leap accelerated stochastic simulation. J. Chem. Phys. **122**, 024112 1–7 (2005)
15. de Boer, F.K., Hogeweg, P.: Co-evolution and ecosystem based problem solving. Ecol. Informat. **9**, 47–58 (2012)
16. Dematté, L., Mazza, T.: On parallel stochastic simulation of diffusive systems. In: Computational Methods in Systems Biology. Lecture Notes in Computer Science, vol. 5307, pp. 191–210. Springer, Berlin (2008)
17. Dematté, L., Prandi, D.: GPU computing for systems biology. Brief. Bioinform. **11**(3), 323–333 (2010)

18. Dittamo, C., Cangelosi, D.: Optimized parallel implementation of Gillespie's first reaction method on graphics processing units. In: IEEE International Conference on Computer Modeling and Simulation (ICCMS), pp. 156–161. IEEE Computer Society, Los Alamitos (2009)
19. Dittrich, P.: Chemical computing. In: Unconventional Programming Paradigms (UPP 2004). Lecture Notes in Computer Science, vol. 3566, pp. 19–32. Springer, Berlin (2005)
20. Dittrich, P., Banzhaf, W.: Self-evolution in a constructive binary string system. Artif. Life **4**, 203–220 (1998)
21. Dittrich, P., Speroni di Fenizio, P.: Chemical organization theory. Bull. Math. Biol. **69**(4), 1199–1231 (2007)
22. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial chemistries—a review. Artif. Life **7**(3), 225–275 (2001)
23. Elf, J., Ehrenberg, M.: Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases, supplementary material: next subvolume method. Proc. IEE Syst. Biol. **1**(2), 230–236 (2004)
24. Fontana, W., Buss, L.W.: 'The arrival of the fittest': toward a theory of biological organization. Bull. Math. Biol. **56**, 1–64 (1994)
25. Frachebourg, L., Krapivsky, P.L., Ben-Naim, E.: Spatial organization in cyclic Lotka–Volterra systems. Phys. Rev. E **54**, 6186–6200 (1996)
26. Fricke, T., Schnakenberg, J.: Monte-Carlo simulation of an inhomogeneous reaction–diffusion system in the biophysics of receptor cells. Z. Phys. B Condens. Matter **83**(2), 277–284 (1991)
27. García-Quismondo, M., Gutiérrez-Escudero, R., Martínez-del-Amor, M.A., Orejuela-Pinedo, E., Pérez-Hurtado, I.: P-Lingua 2.0: a software framework for cell-like P systems. Int. J. Comput. Commun. Control **IV**(3), 234–243 (2009)
28. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. J. Phys. Chem. A **104**(9), 1876–1889 (2000)
29. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. J. Phys. Chem. **81**(25), 2340–2361 (1977)
30. Gillespie, D.T.: Approximate accelerated stochastic simulation of chemically reacting systems. J. Chem. Phys. **115**(4), 1716–1733 (2001)
31. Gillespie, D.T.: Stochastic simulation of chemical kinetics. Ann. Rev. Phys. Chem. **58**, 35–55 (2007)
32. Han, S., Jang, K., Park, K., Moon, S.: PacketShader: a GPU-accelerated software router. SIGCOMM Comput. Commun. Rev. **40**(4), 195–206 (2010)
33. Hattne, J., Fange, D., Elf, J.: Stochastic reaction–diffusion simulation with MesoRD. Bioinformatics **21**(12), 2923–2924 (2005)
34. Hutton, T.J.: Evolvable self-reproducing cells in a two-dimensional artificial chemistry. Artif. Life **13**(1), 11–30 (2007)
35. Jenkins, D., Peterson, G.: AESS: accelerated exact stochastic simulation. Comput. Phys. Commun. **182**(12), 2580–2586 (2011)
36. Jeschke, M., Park, A., Ewald, R., Fujimoto, R., Uhrmacher, A.M.: Parallel and distributed spatial simulation of chemical reactions. In: 22nd Workshop on Principles of Advanced and Distributed Simulation, pp. 51–59. IEEE Computer Society, Washington (2008)
37. Kanada, Y.: Combinatorial problem solving sing randomized dynamic composition of production rules. In: IEEE International Conference on Evolutionary Computation, pp. 467–472 (1995)
38. Komarov, I., D'Souza, R.M., Tapia, J.-J.: Accelerating the Gillespie τ-leaping method using graphics processing units. PLoS ONE **7**(6) (2012)
39. Langdon, W.B.: A fast high quality pseudo random number generator for nVidia CUDA. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference (GECCO): Late Breaking Papers, pp. 2511–2514. ACM, New York (2009)
40. Lenser, T., Hinze, T., Ibrahim, B., Dittrich, P.: Towards evolutionary network reconstruction tools for systems biology. In: Proceedings of EvoBio. Lecture Notes in Computer Science, vol. 4447, pp. 132–142. Springer, Berlin (2007)

41. Li, H., Petzold, L.: Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. Int. J. High Perform. Comput. Appl. **24**, 107–116 (2010)
42. Lu, H., Li, P.: Stochastic projective methods for simulating stiff chemical reacting systems. Comput. Phys. Commun. **183**, 1427–1442 (2012)
43. Lu, T., Volfson, D., Tsimring, L., Hasty, J.: Cellular growth and division in the Gillespie algorithm. Syst. Biol. **1**(1), 121–128 (2004)
44. Lu, P.J.: Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station. J. Real-Time Image Process. **5**(3), 179–193 (2010)
45. Maitre, O.: Understanding NVIDIA GPGPU Hardware. In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37959-8
46. Marquez-Lago, T.T., Burrage, K.: Binomial tau-leap spatial stochastic simulation algorithm for applications in chemical kinetics. J. Chem. Phys. **127**(10) (2007)
47. Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Cecilia, J.M., Guerrero, G.D., García, J.M.: Simulation of recognizer P Systems by using manycore GPUs. In: RGNC REPORT 2/2009, Seventh Brainstorming Week on Membrane Computing, vol. II, pp. 45–57, February 2009
48. Mather, W.H., Hasty, J., Tsimring, L.S.: Fast stochastic algorithm for simulating evolutionary population dynamics. Bioinformatics **28**(9), 1230–1238 (2012)
49. McKinley, P., Cheng, B., Ofria, C., Knoester, D., Beckmann, B., Goldsby, H.: Harnessing digital evolution. IEEE Comput. **41**(1), 54–63 (2008)
50. Mobilia, M., Georgiev, I.T., Täuber, U.C.: Phase transitions and spatio-temporal fluctuations in stochastic lattice Lotka–Volterra models. J. Stat. Phys. **128**(1–2), 447–483 (2007)
51. Molnár Jr., F., Izsák, F., Mészáros, R., Lagzi, I.: Simulation of reaction-diffusion processes in three dimensions using CUDA. ArXiv e-prints, April 2010
52. Nowak, M.A.: Evolutionary Dynamics, Exploring the Equations of Life. The Belknap Press of Harvard University Press, Cambridge (2006)
53. Pang, W.-M., Wong, T.-T., Heng, P.-A.: Generating massive high-quality random numbers using GPU. In: IEEE Congress on Evolutionary Computation (CEC), IEEE World Congress on Computational Intelligence, pp. 841–847 (June 2008)
54. Paun, G.: Computing with membranes. J. Comput. Syst. Sci. **61**(1), 108–143 (2000)
55. Pérez-Jiménez, M.J., Romero-Campero, F.J.: P Systems, a new computational modelling tool for systems biology. In: Transactions on Computational Systems Biology VI. Lecture Notes in Bioinformatics, vol. 4220, pp. 176–197. Springer (2006)
56. Popovici, E., Bucci, A., Wiegand, R.P., de Jong, E.D: Coevolutionary principles. In: Handbook of Natural Computing. Springer, Berlin (2010)
57. Rathinam, M., Petzold, L.R., Cao, Y., Gillespie, D.T.: Stiffness in stochastic chemically reacting systems: the implicit tau-leaping method. J. Chem. Phys. **119**(24), 12784–12794 (2003)
58. Rodríguez, J.V., Kaandorp, J.A., Dobrzynski, M., Blom, J.G.: Spatial stochastic modelling of the phosphoenolpyruvate-dependent phosphotransferase (PTS) pathway in *Escherichia coli*. Bioinformatics **22**(15), 1895–1901 (2006)
59. Romero-Campero, F.J., Twycross, J., Camara, M., Bennett, M., Gheorghe, M., Krasnogor, N.: Modular assembly of cell systems biology models using P systems. Int. J. Found. Comput. Sci. **20**(3), 427–442 (2009)
60. Sanderson, A.R., Meyer, M.D., Kirby, R.M., Johnson, C.R.: A framework for exploring numerical solutions of advection–reaction–diffusion equations using a GPU-based approach. Comput. Vis. Sci. **12**(4), 155–170 (2009)
61. Schnell, S., Turner, T.E.: Reaction kinetics in intracellular environments with macromolecular crowding: simulations and rate laws. Prog. Biophys. Mol. Biol. **85**(2–3), 235–260 (2004)
62. Smith, M.: Using massively-parallel supercomputers to model stochastic spatial predator–prey systems. Ecol. Model. **58**(1–4), 347–367 (1991)

63. Stundzia, A.B., Lumsden, C.J.: Stochastic simulation of coupled reaction-diffusion processes. J. Comput. Phys. **127**(1), 196–207 (1996)
64. Suzuki, H.: An example of design optimization for high evolvability: string rewriting grammar. BioSystems **69**(2–3), 211–221 (2003)
65. Suzuki, Y., Fujiwara, Y., Takabayashi, J., Tanaka, H.: Artificial life applications of a class of P Systems: abstract rewriting systems on multisets. In: Workshop on Multiset Processing (WMP), pp. 299–346. Springer, London (2001)
66. Szabó, G., Czárán, T.: Phase transition in a spatial Lotka–Volterra model. Phys. Rev. E **63**, 061904 (2001)
67. Takahashi, K., Kaizu, K., Hu, B., Tomita, M.: A multi-algorithm, multi-timescale method for cell simulation. Bioinformatics **20**(4), 538–546 (2004)
68. Thomas, D.B., Howes, L., Luk, W.: A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), pp. 63–72. ACM, New York (2009)
69. Tian, T., Burrage, K.: Binomial leap methods for simulating stochastic chemical kinetics. J. Chem. Phys. **121**(21), 10356–10364 (2004)
70. Turing, A.M.: The chemical basis of morphogenesis. Philos. Trans. R. Soc. Lond. B **327**, 37–72 (1952)
71. Vigelius, M., Lane, A., Meyer, B.: Accelerating reaction–diffusion simulations with general-purpose graphics processing units. Bioinformatics **27**(2), 288–290 (2011)
72. Vigelius, M., Meyer, B.: Multi-dimensional, mesoscopic Monte Carlo simulations of inhomogeneous reaction-drift-diffusion systems on graphics-processing units. PLoS ONE, 7(4) (2012)
73. Wolkenhauer, O., Ullah, M., Kolch, W., Cho, K.-H.: Modelling and simulation of intracellular dynamics: choosing an appropriate framework. IEEE Trans. Nano-Biosci. **3**(3), 200–207 (2004)
74. Xu, L., Taufer, M., Collins, S., Vlachos, D.: Parallelization of tau-leap coarse-grained Monte Carlo simulations on GPUs. In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–9 (April 2010)
75. Yamamoto, L., Banzhaf, W.: Catalytic search in dynamic environments. In: Artificial Life XII, Proceedings of the Twelfth International Conference on the Synthesis and Simulation of Living Systems, pp. 277–285. MIT Press, Cambridge (August 2010)
76. Yamamoto, L., Banzhaf, W., Collet, P.: Evolving reaction–diffusion systems on GPU. In: Proceedings of XV Portuguese Conference on Artificial Intelligence (EPIA), Thematic Track on Artificial Life and Evolutionary Algorithms (ALEA). Lecture Notes in Artificial Intelligence, vol. 7026, pp. 208–223. Springer, Berlin (2011)
77. Yamamoto, L., Miorandi, D., Collet, P., Banzhaf, W.: Recovery properties of distributed cluster head election using reaction–diffusion. Swarm Intell. **5**(3–4), 225–255 (2011)
78. Zhou, Y., Liepe, J., Sheng, X., Stumpf, M., Barnes, C.: GPU accelerated biochemical network simulation. Bioinformatics **27**(6), 874–876 (2011) [Applications Note].
79. Zhu, T., Hu, Y., Ma, Z.-M., Zhang, D.-X., Li, T., Yang, Z.: Efficient simulation under a population genetics model of carcinogenesis. Bioinformatics **27**(6), 837–843 (2011)