BRIEFINGS IN BIOINFORMATICS. VOL 11. NO 3. 323–333 doi:10.1093/bib/bbq006
Advance Access published on 7 March 2010

GPU computing for systems biology

Lorenzo Demattè and Davide Prandi

Submitted: 20th November 2009; Received (in revised form): 30th January 2010

Abstract
The development of detailed, coherent, models of complex biological systems is recognized as a key requirement for integrating the increasing amount of experimental data. In addition, in-silico simulation of bio-chemical models provides an easy way to test different experimental conditions, helping in the discovery of the dynamics that regulate biological systems. However, the computational power required by these simulations often exceeds that available on common desktop computers and thus expensive high performance computing solutions are required. An emerging alternative is represented by general-purpose scientific computing on graphics processing units (GPGPU), which offers the power of a small computer cluster at a cost of ~\$400. Computing with a GPU requires the development of specific algorithms, since the programming paradigm substantially differs from traditional CPU-based computing. In this paper, we review some recent efforts in exploiting the processing power of GPUs for the simulation of biological systems.

Keywords: *systems biology; simulation; agent-based modelling; cellular automata; GPGPU; CUDA*

INTRODUCTION
Data collected by high-throughput tools and genome sequencing give precise information on the basic constituents of life; this large amount of data calls for a shift from a reductionist approach to a systematic view of biological systems. An accurate description of the components and of the interactions among them, supported by the use of computational methods, can lead to a better understanding of living systems [1]. Modelling, simulation and analysis are the tools of a new kind of multidisciplinary scientist, working in the field between biology, mathematics, computer science and engineering. This new kind of science needs computationally intensive applications. The high parallelism expressed by the biochemical reactions underlying life leads to the idea of using parallel computing techniques to tackle the complexity of biological systems (see ref. [2] for a review). Parallel computing techniques require dedicated architectures. Multiple instruction multiple data (MIMD) architectures consist of multiple independent processors simultaneously executing different instructions on different data. Examples of MIMD platforms are *clusters* of computers and *GRID computing*. The main drawback of MIMD platforms is the cost: the expense of MIMD architecture is such that only large institutions can offered it. Therefore, MIMD platforms look unlikely to be a practical solution for everyday research in systems biology. An alternative to MIMD platforms are single instruction multiple data (SIMD) architectures.

SIMD is a type of architecture in which many processing units execute the same instruction on different data elements. Supercomputers built ~70s and 80s were based on the SIMD paradigm. The 90s saw the advent of cheaper and more powerful MIMD platforms, i.e. clusters and GRIDs, with the consequent abandonment of SIMD architectures. Only recently, with the increase of the computational power and the low costs of the new processing architectures, has the attention of the scientific community moved back to SIMD platforms [3]. In particular, an interesting alternative is represented by general-purpose scientific computing on graphics processing units (GPGPUs [4]). A graphic processing unit (GPU) [5] is a processing unit developed for

Corresponding author. Davide Prandi, The Microsoft Research, University of Trento, Centre for Computational and Systems Biology, Piazza Manci 17 38123 Povo, Trento, Italy. Tel: +39-0461-882834; Fax: +39-0461-882814; E-mail: prandi@cosbi.eu
Davide Prandi is a researcher of the CoSBi centre. His research interests are in complexity reduction of biological systems, stochastic simulation, process algebras, mathematics and logics. He held positions before at the University of Catanzaro.
Lorenzo Demattè is a PhD student at the University of Trento, carrying out its research at the CoSBi centre. His research interests include languages for systems biology, GPU computing, parallel and concurrent languages, and compilers.

accelerating graphic applications. It provides a large level of parallelism using a fraction of the budget required by usual MIMD architectures. GPU computing requires the development of specific algorithms, since the programming paradigm substantially differs from the traditional CPU-based computing, and therefore specific programming skills are needed.

The literature about modelling, simulation and analysis of biological systems covers a wide spectrum of different issues. In this work we mainly focus on the simulation of the dynamics of biochemical systems. Even with this simplification, a variety of models of biological systems can be identified. We roughly classify biological models into species and individual based. Species-based models group the representations of biological systems in which the identity of any single entity is not considered. An example of such a model is a system of chemical reactions in the form $A + B \rightarrow C + D$, where only the amount of each chemical is represented. Individual-based models cover those cases where more details, such as the position and the mass of each element of the model, are needed. The structure of the article is based on this classification.

In the next section, we briefly introduce the GPU architecture and the main ideas behind general purpose programming on GPUs. Then, species- and individual-based systems are introduced, together with a survey of the more recent and interesting GPGPU applications. The article concludes with a discussion about performance improvements of GPUs over conventional architectures.

GPU

A GPU is a processor designed to accelerate the computation of graphics operations. The term GPU is often used in contrast or comparison with central processing unit (CPU), the main general purpose processor at the core of every computer. Specifically, GPUs are placed on graphics boards where they are used to speed up 3D graphics *rasterization,* the task of taking an image described as a series of shapes and converting it into a raster image for output on a video display. This process can be controlled using small programs called *shaders.* The shader instruction set has evolved over the years to the point that it is now possible to use GPUs for general purpose computations.

GPU computing started as an effort by the scientific community to exploit the raw processing power of GPUs to make intensive computations. In fact, the power of the most recent GPUs is comparable to the computational power of a cluster with hundreds of CPU cores. However, due to their architecture, this power can be exploited by only a few specialized algorithms.

In general, the architecture of a GPU is tailored to 3D graphics computations. The characteristics of graphics computations (highly parallel, very high arithmetic intensity,[1] simple stream of mathematical instructions executed on the same data types) dictated the design of GPUs: little or no cache at all, a cluster of SIMD cores and a memory with high bandwidth. Overall, compared to CPUs, GPUs are relatively simple: CPUs are designed to run a very wide variety of programs, even purely serial programs, as quick as possible, and therefore they include very complex logic and large caches. GPUs instead are very specialized: most of their silicon is used to perform arithmetic computations.

The architecture details vary from vendor to vendor, and sometimes even from one model to another. In this survey we will focus on the NVIDIA GPU architecture, as it is the most used for GPU computing. NVIDIA was the first manufacturer to address GPU computing specifically, with the introduction of compute unified device architecture (CUDA) [6]. CUDA GPUs are organized in *multiprocessors,* which group multiple *streaming processors*, the basic execution units (Figures 1 and 2). CUDA executes the same program on all the *multiprocessors*: the code for the program (*kernel*) is the same but both the data and the execution flow can be different and diverge. CUDA launches multiple instances of the same kernel, called *threads*. Threads are grouped in *warps* (Table 1) for execution on a multiprocessor. Threads are runtime instances of the same kernel, and therefore they execute the same program code; furthermore, all the threads in a warp are executed by one multiprocessor in an SIMD fashion, and therefore they must execute exactly the same instruction at the same time, although on different data. If threads diverge (taking, for example, different branches of an *if* statement), they will be split into different warps, leading possibly to under-utilization of the multiprocessors. These restrictions help in keeping the architecture simple but powerful: thanks to the big amount of silicon allocated to arithmetic operations, the

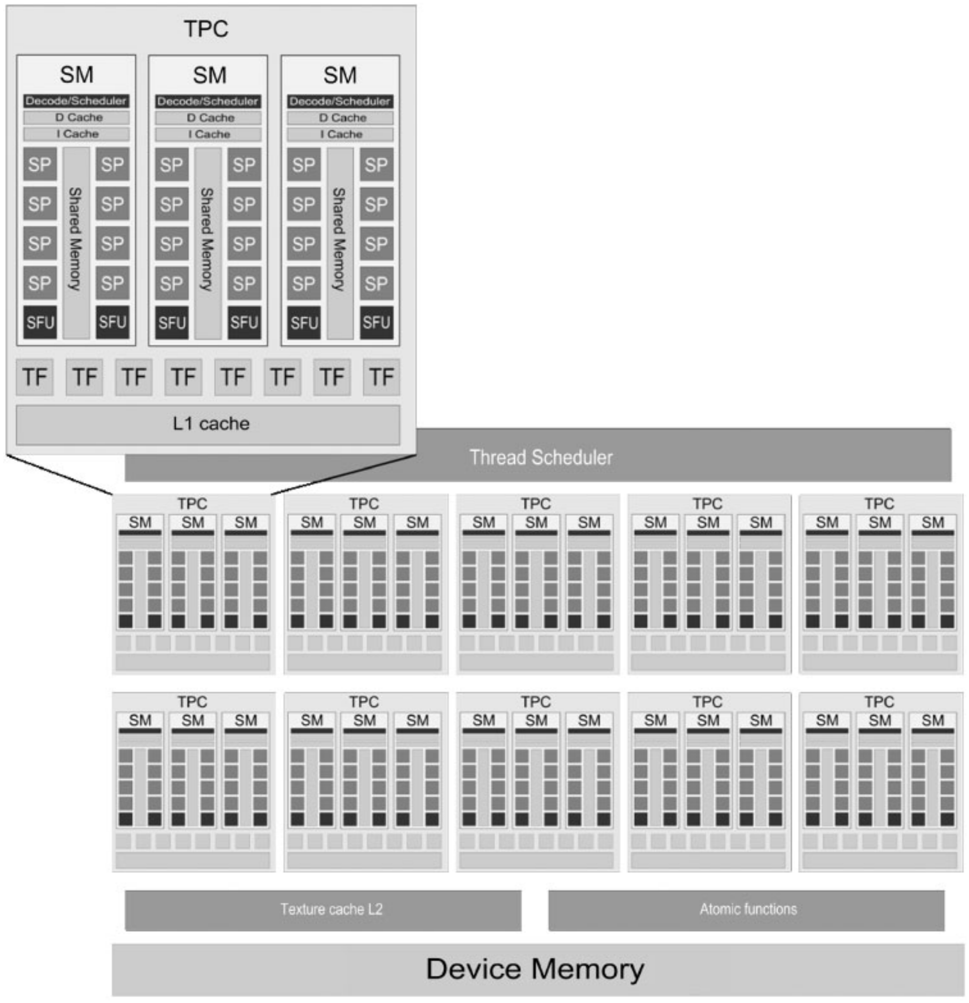

Figure 1: The structure and computing resources of a NVIDIA GT200 chip. Notice the 10 processor clusters, each containing three multiprocessors.

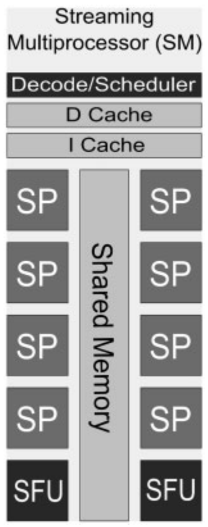

Figure 2: The GT200 multiprocessor, with its own instruction unit and eight streaming processors.

raw power of GPUs is enormous, but this power can be exploited only by programs that are well-suited to this architecture.

Those applications that process large amounts of data or objects, and perform the same operations on all of them, will fit well on a GPU: to keep all the *streaming processors* busy, and therefore to obtain good performances, *tens of thousands* threads need to be executed concurrently. Therefore, the applications based on the execution of disparate, short tasks will cause the fragmentation of warps and lead to the under-utilization of *multiprocessors*. Similarly, the applications that process a small subset of data at each time will fail in feeding the *streaming processors* with enough data. Finally, applications requiring double precision floating point numbers are currently severely limited: the support for double precision has been added only in the latest generation of

Table 1: CUDA terminology

Device/host	GPU/CPU
Kernel	Function called from the host and executed on device Kernels are executed one at time, by many *threads*.
Thread	Instruction stream flowing into a single execution unit. Note that they are *not* like CPU threads, since, (e.g.) they are free of context switch.
Warp	Set of threads (currently 32). The Warp is the scheduling unit (one warp is scheduled on one *multiprocessor*).
Block	Set of *threads* that can cooperate via shared memory and synchronize to each other.
Grid	The 'structure' on which blocks of threads are launched (only a facility for decomposing your domain, for having threads that access different parts of your data).

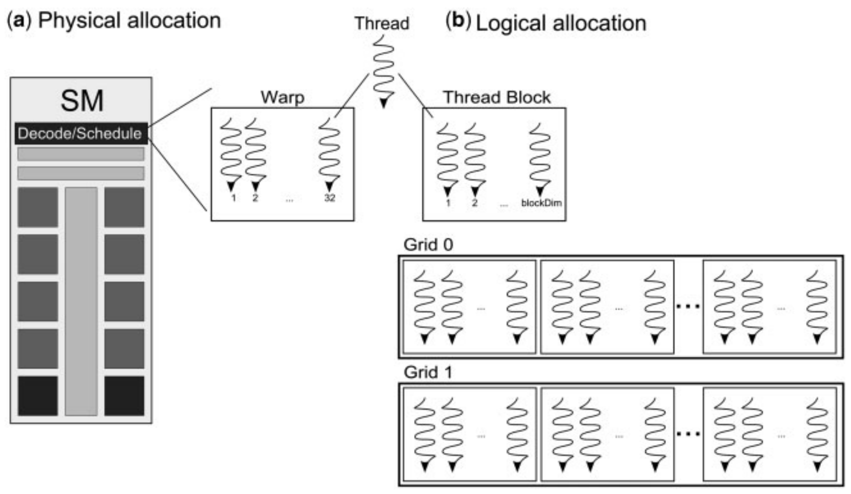

Figure 3: Physical and logical allocation of a thread.

GPUs, and in a reduced way. For example, on NVIDIA GPUs only one streaming processor for each multiprocessor is capable of operating in double precision; this leads to performances that are at best one eighth of the single precision performances. Double precision is very important in some scenarios; in Monte-Carlo simulations and in numerical integration single precision is sometimes not enough.

PROGRAMMING GPUS

The first GPUs where programmed by submitting a string containing the shader program to the GPU driver through a graphics API like DirectX or OpenGL. Later, C-like higher level languages (HLSL and GLSL) were introduced, making the overall programming easier. However, these languages were still targeted at 3D graphics applications: the code had still to be submitted explicitly to the GPU via graphics API calls, data had to be mapped to graphics concepts and moved explicitly (sometimes inefficiently) back and forth from the GPU to the central memory, again using counter-intuitive graphics APIs. With the advent of GPU computing, several other languages or libraries were introduced: the latest example are Brook [7], OpenCL [8] and CUDA [6].

The term CUDA usually refers to both an architecture and its associated programming model. The CUDA GPUs are programmed through an API and a set of C language extensions. CUDA embeds the GPU code inside C++ code, using the language extensions to indicate whether a function should be executed on the CPU (called 'host') or on the GPU ('device'). It is therefore independent of graphics libraries.

All the architectural details (threads, warps, multiprocessor, etc.) are hidden to the end user; CUDA instead exposes the notions of *blocks, grids* and *threads* (see Table 1) to ease the decomposition of the problem domain. As depicted in Figure 3, threads are both the 'physical' and 'logical' basic units of execution; the GPU groups and schedules threads in

Table 2: Plain C code versus CUDA code for implementing a simple algorithm

```
void increment_cpu(float *a, float
b,
   int N)
{
   for (int idx = 0; idx<N; idx++)
      a[idx] = a[idx] + b;
}
```

```
__device__
void increment_gpu(float *a, float
b,
   int N)
{
   int idx = blockIdx.x *
blockDim.x +
             threadIdx.x;
   if (idx < N)
      a[idx] = a[idx] + b;
}
```

Notice how the loop is unrolled; calling the kernel on the right will require spawning N threads, each of them incrementing a single item.

warps, while CUDA offers a higher level view of grids and blocks. Grids and blocks can be used by the programmer to map the subdivisions inherent in the problem domain (in particular, spatial subdivisions) in a convenient way. Each thread is then provided with variables representing the block and grid coordinates on which it needs to operate; using these coordinates, a thread can access and process a single item or subset of the problem domain.

As an example, consider the simple and common scenario of porting computationally intensive loops to the GPU. In order to enable efficient execution, loops have to be *transformed*, *strip-mining* or *unrolling* them. After unrolling each thread executes a single, distinct iteration of the original loop. For instance, Table 2 shows a simple algorithm that takes a vector ‘a’ of length ‘N’ and a value ‘b’ and increments each value of ‘a’ by ‘b’. As expected, the sequential algorithm on the left accesses the elements of ‘a’ one by one. Instead, the kernel code on the right spawns ‘N’ parallel threads, each of them incrementing a single value of ‘a’. The position in the array ‘a’ that the thread T has to increment is obtained using a common pattern to compute a linear index: multiply the block index of T (**`blockIdx`**.x) by the number of threads per block (**`blockDim`**.x) and finally add the current index of T within the block (**`threadIdx`**.x).

SPECIES-BASED SYSTEMS

Species-based systems organize biological entities into classes where the elements of each class cannot be distinguished. The quantity of a molecule in a class could be represented as either a *continuous* or a *discrete* variable.

In the continuous case, molecules are modelled as time-dependent variables representing concentrations. Interactions are rendered as differential relations between variables. This enables the use of the *Ordinary Differential Equation* (*ODE*) machinery. In particular, a *reaction-rate equation* is used to describe the rate of change of the concentration of a molecule as a function f of the concentrations of the other components. Usually, the function f is not linear and the common way to work around analytical intractability is to exploit numerical techniques. Numerical methods involve the use of linear algebra tools, both when performing matrix–matrix/matrix–vector calculations, and when implementing methods that require the solution of a system of linear equations. The basic linear algebra subprogram (BLAS) is the de facto standard API that provides basic building blocks for performing linear algebra operations. The CUBLAS library [9] and the MAGMA project [10] are implementations of BLAS on GPU architectures. These offer the basic routines on top of which ODE solvers are designed. GPU power is also exploited to speed up the simulation of specific ODE systems. In ref. [11], the authors optimize to run on a GPU the MATLAB code of two typical systems biology applications, namely, Heart Wall Tracking and Cardiac Myocyte Simulation, obtaining good performances. The application presented in ref. [12] is more general: an SBML [13] model is automatically compiled into the CUDA code; the code is simulated with a large number of varying parameters to understand the available parameter space of the underlying ODEs.

In a discrete setting, the evolution of a biological system could be characterized as a stochastic process, where components are present in an enumerable quantity. A system is represented as a vector $\boldsymbol{X}$ of discrete random variables: the integer amount of a molecule i at time t is expressed as a random variable $X_i(t)$. The stochastic simulation algorithm (SSA) [14] generates a trajectory, i.e. a possible evolution history of the considered system, relying on Monte Carlo methods. The key tool of SSA is the definition of a *propensity function* for each reaction *j* in the system: the likelihood that a reaction *j* fires in the next infinitesimal interval is a function of the number of molecules involved in reaction *j* and of a constant specific to *j*. For instance, given a reaction $X_1 + X_2 \xrightarrow{c} X_3$, the propensity function is $c \times |X_1| \times |X_2|$, where $|X|$ represents the number of molecules of *X*. SSA implementations [15] follow a common template:

(1) Data structure initialization.
(2) Random selection of a reaction according to the propensity function.
(3) Execution of the selected reaction.
(4) Update of the data structures.
(5) Return to step 2 or Terminate.

SSA is structurally a sequential algorithm and therefore hard to parallelize. However, GPUs can impact significantly on the time simulation process. First of all, SSA requires generating a large quantity of random numbers, a time consuming task. Using GPUs as a fast random number generator [16] reduces the time needed for a run [17]. Moreover, SSA is used to collect statistics on a certain system by generating a large collection of stochastic realizations; the streaming architecture of GPUs is well suited for this kind of parallelism[2] as shown in refs. [17, 18]. Finally, a promising attempt to parallelize a single instance of SSA exists [19]: the authors reorganize the structure of SSA in order to reduce the complexity in space of the algorithm. In this way it is possible to split the reaction set among blocks and to obtain a certain level of parallelism inside a single simulation.

COMPARTIMENTIZED SYSTEMS

The models presented above share the view of biological systems as boxes containing all the molecules without physical barriers. However, it may be important to represent compartments, as in the case where translocation of proteins from the cytosol to the nucleus is essential in the model. Compartments can be managed either implicitly or explicitly. In the first case a variable, continuous or discrete, which represents an entity which may exist in two different compartments, is split into two different variables. This results in a larger model, since an entity gives rise to more than one variable. However, the techniques presented in the previous section can be exploited. In the second case, there are many different representations of compartments. Here, we focus on *P systems* [20] because they are quite general and many GPU implementations are available.

A P system is a computational model inspired by the structure of the cell. The use of P systems to model biological processes is pioneered in ref. [20] and has received increasing attention since, because it offers a suitable abstraction for many biological compartimentized systems. A P system configuration is made up of three components: (i) a set of *membranes* (a membrane may contain other membranes); (ii) a set of *chemicals* inside each membrane; (iii) a set of *evolution rules* (i.e. chemical reactions). A computation is given by a sequence of transitions between configurations performed by applying evolution rules to the chemicals placed inside membranes. Starting from this common definition, a number of different models were derived, varying, for instance, the order of application of the evolution rules, or the capability of membranes to divide. P systems express two levels of parallelism, one among membranes and other among the chemicals inside a membrane; this model fits very nicely on CUDA, where there are two levels as well: a grid organizes on a first level several blocks; on a second level, concurrent execution among threads takes place inside a block. The double parallel nature expressed by both CUDA and P systems suggests that a GPU implementation of P systems would be effective. A valuable example is given by P-Lingua [21], a programming language for specifying membrane systems that can be compiled and executed directly on a GPU without requiring specific skills of the user [22].

INDIVIDUAL-BASED SYSTEMS

Many biological processes take place in a non-homogeneous, crowded environment in which spatially localized fluctuations of inorganic catalysts and slow intracellular diffusion have an important role. In these cases it is important to consider each molecule in the system as an *individual entity*. To deal with such processes it is mandatory to explicitly consider the cell geometry, and in general the spatial conformations and the diffusion processes. Available simulation algorithms work at different levels of abstraction, which influence both accuracy and performances.

Molecular dynamics (MD) works at the level of the atoms. Methods that simulate quantum mechanical and molecular mechanical dynamics have been applied to a wide range of problems of biological interest (see ref. [23] for a review): these simulations explicitly represent every detail of the considered chemical reaction, such as the position and the energy of every atom in the system. MD methods map well on GPUs, and many solutions are proposed. Here, it is worth mentioning the pioneering work on Namd [24], VMD [25] and HOOMD [26].

Brownian dynamics (BD) [27] methods operate at a slightly coarser level of detail, where molecules have an identity and an exact position in a continuous space, but no volume, shape or inertia. Each molecule of interest is represented as an individual point. Brownian dynamics simulation generally adopts a stochastic simulation approach based on the solution of the Smoluchowski equation, which describes the diffusive encounter of the molecules in the solution. An alternative approach, proposed by ref. [28], is to represent the dynamics of globally interacting Brownian particles with the Kuramoto model; in this way, the simulation is reduced to the numerical solution of some stochastic differential equations. The integration is performed using a stochastic scheme of the second order. Time steps are discrete; at each step the equations are computed and the positions of all particles are updated.

LATTICE-BASED METHODS

At a coarse level of detail we present some *lattice-based methods*. The simulated space is divided into three dimensional elements. Particularly interesting for GPU computing are *cellular automata* (CA) based methods. Here, space and time are discrete, and the evolution in time of the system is governed only by local information, instead of obeying a global equation. Therefore, CA models fit nicely on the GPU model of computation (see ref. [29] for a survey on CA simulation algorithm and a CPU/GPU comparison). Two methods are of notable interest for systems biology applications: *coupled map lattices* (CML) [30] and the *multiparticle model* [31].

CML is an extension of a CA where the discrete state values of the CA cells are replaced by continuous real values. Efficient implementation of the Gray-Scott model [32] and of the Turing pattern models [33] are obtained running CML on GPUs. They are usually implemented as partial differential equations that describe the concentrations of chemical reactants at each lattice site over time; their GPU implementation consists of a single data stream where the concentrations of the chemical species are stored in different channels of a single texture that represents the discrete spatial grid. This stream serves as input to a kernel, which implements the partial differential equations in a discrete form.

The multiparticle diffusion model is more complex and more realistic. In this model, multiple particles per lattice site are permitted; particles move in a stochastic way by following independent random walks between positions in the lattice. Brownian diffusion is therefore modelled as a series of independent random choices for the movement of particles on a regular, uniform grid. The algorithm described in ref. [31] implements a multiparticle model on GPU in an efficient way using a novel data structure; the authors apply the method to a 3D model of *in-vivo* diffusion inside the *Escherichia coli* cell.

AGENT-BASED MODELS

The *Agent-based model* (ABM) generalizes the CA model. ABMs are computational representations of dynamic systems where a number of individual, autonomous constituent entities (called *Agents*) interact locally in order to recreate a higher level, group behaviour. This ability to simulate the emergent behaviour of complex systems from local interactions makes agents attractive for systems biology. Indeed, ABMs have been used to model and simulate inflammatory cell tracking, tumour growth, intracellular processes, wound healing, morphogenesis, microvascular patterning, pharmacodynamics and tuberculosis (see ref. [34] for a survey).

Even if Agents are concurrent, independent objects, historically only sequential simulation algorithms have been implemented. One of the first parallel implementations running on graphics hardware was performed by De Chiara *et al.* [35]. Notably, they studied the distributed behaviour of a flock, a widely-known problem in systems biology. Recently, several research efforts concentrated on ABM simulation on GPUs. Perumalla *et al.* [36], for example, used an extended cellular automata approach to simulate ABMs on the GPU. However, being based on CA and therefore on lattice sites, they have limitations in the number of agents and on replications. Two groups, in particular, pushed the state of art in large-scale ABM simulation, by extending existing ABM frameworks with rich and complete support for simulation on a GPU: Richmond *et al.* [37] with FLAME and D'Souza *et al.* [38] with SugarScape. They rely on existing agent frameworks supporting a number of key ABM features, such as, e.g. birth and death allocation, agent replacement and movement, pollution formation and diffusion, collision detection. Of particular relevance for systems biology application is the application of SugarScape to the 3D simulation of granuloma formation in TB infection [39]. The authors demonstrated that ABM frameworks running on GPUs are flexible and mature enough to run complex simulations, with a speed that is three orders of magnitude faster than the sequential algorithm.

DISCUSSION

The advent of systems biology calls for an urgent development of new techniques to tackle the time required by the simulation of biological systems. The kind of parallelism expressed by biological systems fits well with the streaming programming paradigm, making GPUs appealing as hardware dedicated to their simulation. Moreover, a GPU combines high performance parallel computing with low budget requirements, making GPGPU a valuable tool for systems biology. However, not all applications are well suited for a GPU implementation and performances vary considerably depending on the considered biological system. In Table 3 we relate biological systems and GPU speed-up; the table reports the improvements of combining GPUs and CPUs in comparison to CPUs alone, together with the GPU and the software package used. The column 'speed-up' refers to the simulation execution time; for instance, a 10× speed-up means that the simulation time required by a CPU-only system is 10 times greater than that of a CPU/GPU configuration. These data have to be considered carefully, since the way in which performance measurements are taken varies greatly; furthermore, GPU performances vary greatly from one model to another; Figure 4 reports a comparison of the GPUs listed in Table 3 in terms of GFLOPS, i.e. billion of floating point operations per second, a common measure of performance. For instance, looking at the Table 3 in the light of Figure 4, it emerges that the 2× speedup of MAGMA [10] and of single SSA simulation [19] have not the same value, since GXT280 outperforms 8600M GS by three orders of magnitude. For this reason Table 3 has to be considered only as a sketch of GPU computing power, without any intention of comparing algorithms or implementations.

We first examined those systems where the identity of a single entity is not considered, namely species-based systems. In this context we distinguished between continuous and discrete representation. The first is characterized by ODE systems that are well suited for an implementation on the GPUs because ODE solvers are based on linear algebra. The performances of MAGMA are quite disappointing in this context, but it has to be considered that the project is new[3] and the speedup is computed against a powerful quad core server processor. The report in ref. [40] also considers a dual core processor and, in this case, the speed-up is ∼5×. CUBLAS [9], the more mature NVIDIA implementation of BLAS, shows slightly better performance, especially when dealing with single precision floating point operations.

Table 3: GPU performances

Species based				
ODE	CUBLAS	GTX280	4.1–10×	Measured
	MAGMA	GTX280	2×	[40]
	SBML based	GTX280	59×	[40]
SSA	Multiple simulations	8800GTX	50×	[17]
	Single simulation	8600M GS	2×	[19]
P systems	P-lingua	Tesla C1060	1000×	[22]
Individual based				
MD	Namd	8800GTX	10×	[30]
	VMD	n.a.	125×	[40]
	HOOMD	n.a.	15×	[40]
BD	SDE	Tesla C1060	675×	[40]
CA	CML	Xenos	25×	[33]
ABM	FLAME	9800 GX2	250×	[40]

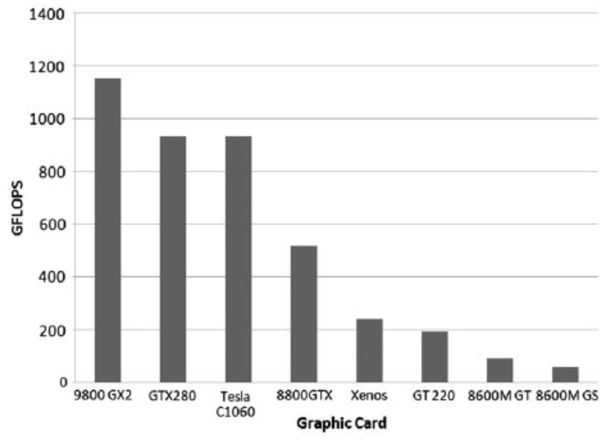

Figure 4: Peak performance for different GPUS.

We measured CUBLAS performances using the widely used GEMM and SYMM operations; a more detailed analysis of CUBLAS performances and how to tune them is available in ref. [41]. The results of ref. [12] are particularly interesting for the systems biology community; the system described offers a 59× speed-up for the simulation of a system of ODEs expressed as SBML code. The application offers useful performance, while at the same time being accessible to researchers without skills in GPGPU programming. Conversely, discrete representations of biological systems do not offer such impressive performance improvements, mainly because the stochastic simulation algorithm (SSA) is hard to parallelize. The situation is better in the (common) case of multiple simulations [17], where, for example, 50 simulations of a GPU require the same time of a single simulation performed with a CPU. A particular note of praise was deserved by discrete systems with compartments; in particular, P Systems parallelism resembles GPUs architecture, offering a natural high performance platform for the simulation of systems with membranes. The 1000× speedup reported in Table 3 is impressive, but the datum may not be representative of the average case: the value is obtained using of an optimized mapping between the number of membranes/objects and the number of blocks/threads on the GPU, therefore the GPU is fully utilized.

Individual-based systems offer specific tools to describe those models where many details, such as the position and the mass of each element in the model, are needed. We first examined molecular dynamics methods that map naturally on GPUs. The methods presented offer good performances, especially the VMD software. The field of MD on GPUs is receiving great attention from the community and new applications are released every month. In contrast, it is quite surprising that Brownian dynamics methods are not supported; as we mentioned earlier in the text, algorithms with a good amount of loop level parallelism fit well with the streaming programming paradigm. A notable exception is ref. [28], which achieves an impressive 675×; even if the speed-up was obtained on a very specific application, it calls for more investigations about GPGPU for BD. Finally, we considered lattice-based methods and agent-based models. GPU implementations of these methods have reached a mature state. As in the case of ODEs and SBML, a key feature of these implementations is the possibility of using the GPU's computing power without having specific programming skills. For instance, the FLAME framework uses an XML specification language for Agents that is automatically compiled into CUDA code. This makes the 250× speedup more interesting, because this computing power is available to all the ABM community; in this case the value is a little overstated since the 9800 GX2 consists of two paired GPUs.

In conclusion, general purpose scientific computing on GPUs is promising but also challenging. Currently, the main bottleneck is in the programming skills required. Even if the release of CUDA-like programming languages makes programming easier, the development of new applications requires the consideration of many specific details, like memory usage or communication bandwidth between the CPU and the GPU, that are not necessarily related to the application domain. This makes simulation of biological systems on GPUs a small niche for specialists. In our opinion, two ingredients are critical for spreading GPU computing to a larger portion of the systems biology community: *abstractions* and *architectures*.

With abstraction we mean that a user has to be able to access GPU power without knowing the details of the underlying hardware. The definition of suitable abstractions would attract more scientists in order to reach a critical mass of users. The SBML interface to ODE, the P-lingua language, and the FLAME framework are good examples. Fermi [42], the next generation of NVIDIA GPGPU architecture and the associated programming APIs, promises ease of use with more power.

The successes of NVIDIA CUDA and the increasing interest of the scientific community invite other big hardware vendors to invest in GPGPU architectures; an example is the ATI FireStream processor, which currently uses the Brook+ language [43]. The hardware limitations, such as the support for double precision, will probably benefit from the competition between GPU vendors. Indeed, the Fermi architecture already promises to solve the double precision issue.

Finally, we would like to report a success story on the combination of GPU computing with cluster or GRID architectures. Folding@Home is a project designed to perform computationally intensive simulations of MD using a grid of voluntary, heterogeneous computing devices. GPU devices attached to the GRID account for roughly 67% of the project processing power, despite being only 7% of the total active clients [44].

Key Points

- Simulation of biological systems calls for high performance computing.
- GPUs combine high performance computing with low budget.
- GPU streaming programming fits well with biological parallelism.
- The application of GPU computing to the simulation of biological systems is promising.

Notes

[1]The ratio of computation to bandwidth, or more formally arithmetic intensity = operations / words transferred.
[2]Named parallelism across the simulation [2].
[3]At the moment, the released version is the 0.2.

References

1. Kitano H. *Foundations of Systems Biology*. Cambridge, MA: MIT Press, 2001.
2. Ballarani P, Guido R, Mazza T, *et al*. Taming the complexity of biological pathways through parallel computing. *Briefings in Bioinformatics* 2009;**10**(3):278–88.
3. Meredith JS, Alam SR, Vetter JS. Analysis of a computational biology simulation technique on emerging processing architectures. In: *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*. Long Beach, California, USA: IEEE, 2007;1–8.
4. GPGPU. [Online]. Available at: http://gpgpu.org/ (18 February 2010, date last accessed).
5. Kayvon F, Houston M. GPUs: a closer look. *Queue* 2008; **6**(2):11–28.
6. Nickolls J, Buck I, Garland M, *et al*. Scalable Parallel Programming with CUDA. *Queue* 2008;**6**(2):40–53.
7. Buck I, Foley T, Horn D, *et al*. Brook for GPUs: stream computing on graphics hardware. In: *Proccedings of the 31st International Conference on Computer Graphics and Interactive Techniques*. Los Angeles, California, USA: ACM, 2004;768.
8. Munshi A. OpenCL: parallel computing on the GPU and CPU. In: *Proceedings of the 35th International Conference on Computer Graphics and Interactive Techniques*. Los Angeles, California, USA: ACM, 2008.
9. C. NVIDIA. CUBLAS Library Document, 2008.
10. Dongarra J, Tomov S, Baboulin M, *et al*. MAGMA. [Online]. Available at http://icl.cs.utk.edu/magma/ (2008) (18 February 2010, date last accessed).
11. Szafaryn L, Skadron K, Saucerman J. Experiences accelerating MATLAB systems biology applications. In: *Biomedicine in Computing: Systems, Architectures, and Circuits (BiC)*. Austin, Texas, USA: ACM/IEEE, 2009;1–4.
12. Ackermann J, Baecher P, Franzel T, *et al*. Massively-parallel simulation of biochemical systems. In: *Proceedings of Massively Parallel Computational Biology on GPUs*. Lecture Notes in Informatics (LNI), Lübeck, Germany, 2009.
13. Hucka M, Finney A, Sauro HM, *et al*. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 2003;**19**(4):524–31.
14. Gillespie DT. Exact stochastic simulation of coupled chemical reactions. *J Phys Chem* 1977;**81**(25):2340–61.
15. Li H, Cao Y, Petzold LR, *et al*. Algorithms and software for stochastic simulation of biochemical reacting systems. *Biotechnol Progr* 2008;**24**(1):56–61.
16. Langdon W. 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: Late Breaking Papers*. Montreal, Canada, 2009, 2511–4.
17. Li H, Petzold L. Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. *Int J High Performance Comp Appl* 2009; doi:10.1177/1094342009106066 [Epub ahead of print].
18. Jenkins D, Peterson G. SAAHPC. [Online]. Available at http://saahpc.ncsa.illinois.edu/09/papers/Jenkins_paper.pdf (July 2009) (18 February 2010, date last accessed).
19. Dittamo C, Cangelosi D. Optimized parallel implementation of Gillespie's first reaction method on graphics processing units. In: *Proceedings of the International Conference on Computer Modeling and Simulation, 2009. ICCMS'09*. Macau, China: UK Simulation Society, 2009;156–61.
20. Ioan AI, Matteo C. Modelling biological processes by using a probabilistic P system software. *Natural Comp* 2003;**2**(2): 173–97.
21. García-Quismondo M, Gutiérrez-Escudero R, Martínez-del-Amor M, *et al*. P–Lingua 2.0: a software framework for cell–like P systems. *Int J Comp, Commun & Control (IJCCC)* 2009;**4**(3):234–43.
22. María Canales J, David Guerrero Hernandez G, Manuel García Carrasco J, *et al*. Simulation of P systems with active membranes on CUDA. In: *Proceedings of the International Workshop on High Performance Computational Systems Biology, 2009. HIBI '09*. Trento, Italy: IEEE Computer Society, 2009;61–70.
23. Grubmüller H, Schulten K, (eds). Advances in molecular dynamics simulations. *J. Struct. Biol.* 2007;**157**(3):443–616.
24. Stone JE, Phillips JCP, Freddolino PL, *et al*. Accelerating molecular modeling applications with graphics processors. *J Comp Chem* 2007;**28**:2618–40.
25. Liu W, Schmidt BS, Vossa G, *et al*. Accelerating molecular dynamics simulations using graphics processing units with CUDA. *Comp Phys Commun* 2008;**179**(9):634–41.
26. Joshua A, Travesset A, Travesset A. 'Molecular dynamics on graphic processing units: HOOMD to the Rescue. *Comp Sci Eng* 2008;**227**(10):5342–59.
27. Langone JJ. *Molecular Design and Modeling: Concepts and Applications. Part A, Proteins, Peptides, and Enzymes*. New York: Academic Press, 1991.
28. Januszewski M, Kostur M. Accelerating numerical solution of stochastic differential equations with CUDA. *Comp Phys Commun* 2010;**181**(1):183–8.
29. Rybacki S, Himmelspach J, Uhrmacher AM. Experiments with single core, multi-core, and GPU based computation of cellular automata. In: *Proceedings of the SIMUL '09. First International Conference on Advances in System Simulation,*. Porto: IEEE Computer Society, 2009;62–7.
30. Harris MJH, Coombe G, Scheuermann T, *et al*. Physically-based visual simulation on graphics hardware. In: *Proceedings*

of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. Saarbrucken, Germany: ACM, 2002; 109–18.

31. Roberts E, Stone JE, Sepulveda L, *et al*. Long time-scale simulations of in vivo diffusion using GPU hardware. In: *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. Rome, Italy: IEEE, 2009;1–8.
32. Pearson J. Complex patterns in a simple system. *Science* 1993;**261**:189–92.
33. Scarl S. Implications of the Turing completeness of reaction-diffusion models, informed by GPGPU simulations on an XBox 360: Cardiac arrhythmias, re-entry and the Halting problem. *Comp Biol Chem* 2009;**33**(4):253–60.
34. Merelli E, Armano G, Cannata N, *et al*. Agents in bioinformatics, computational and systems biology. *Brief Bioinformatics* 2007;45–59.
35. De Chiara R, Erra U, Scarano V, *et al*. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In: *Proceedings of the Vision, Modeling, and Visualization Conference*. Stanford (California), USA: Aka GmbH, 2004;233–40.
36. Perumall KS, Aaby BG. Data parallel execution challenges and runtime performance of agent simulations on GPUs. In: *Proceedings of the SpringSim'08: 2008 Spring Simulation Multiconference.*. Ottawa, Canada: Society for Computer Simulation International, 2008;116–23.
37. Richmond P, Coakley S, Romano DM. A high performance agent based modelling framework on graphics card hardware with CUDA. In: *AAMAS'09: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems*. Hungary, Budapest: International Foundation for Autonomous Agents and Multiagent Systems, 2009;1125–6.
38. D'Souza RM, Lysenko M, Rahmani K. SugarScape on steroids: simulating over a million agents at interactive rates. In: *Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence*. Chicago, IL: ANL/DIS-07-2, 2007.
39. D'Souza RM, Marino S, Kirschner D. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. *SpringSim'09*. San Diego, CA: Society for Computer Simulation International, 2009.
40. NVIDIA, C. CUDA Zone. [Online]. Available at: http://www.nvidia.com/object/cuda_home.html (18 February 2010, date last accessed).
41. Barrachina S, Castillo M, Igual FD, *et al*. Evaluation and tuning of the Level 3 CUBLAS for graphics processors. In: *Proceedings of the Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*. Miami, Florida USA: IEEE, 2008;1–8.
42. C. NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: FERMI. [Online]. Available at: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (2009) (18 February 2010, date last accessed).
43. C. AMD. ATI Stream Computing – Technical Overview. http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf. (2009) (18 February 2010, date last accessed).
44. Stanford University. Folding@home distributed computing. [Online]. HYPERLINK http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats (December 2009) (18 February 2010, date last accessed).