Graphics processing units in bioinformatics, computational biology and systems biology

Marco S. Nobile, Paolo Cazzaniga, Andrea Tangherloni and Daniela Besozzi

Abstract

Several studies in Bioinformatics, Computational Biology and Systems Biology rely on the definition of physico-chemical or mathematical models of biological systems at different scales and levels of complexity, ranging from the interaction of atoms in single molecules up to genome-wide interaction networks. Traditional computational methods and software tools developed in these research fields share a common trait: they can be computationally demanding on Central Processing Units (CPUs), therefore limiting their applicability in many circumstances. To overcome this issue, general-purpose Graphics Processing Units (GPUs) are gaining an increasing attention by the scientific community, as they can considerably reduce the running time required by standard CPU-based software, and allow more intensive investigations of biological systems. In this review, we present a collection of GPU tools recently developed to perform computational analyses in life science disciplines, emphasizing the advantages and the drawbacks in the use of these parallel architectures. The complete list of GPU-powered tools here reviewed is available at http://bit.ly/gputools.

Key words: graphics processing units; CUDA; high-performance computing; bioinformatics; computational biology; systems biology

Introduction

Typical applications in Bioinformatics, Computational Biology and Systems Biology exploit either physico-chemical or mathematical modeling, characterized by different scales of granularity, abstraction levels and goals, which are chosen according to the nature of the biological system under investigation—from single molecular structures up to genome-wide networks—and to the purpose of the modeling itself.

Molecular dynamics, for instance, simulates the physical movements of atoms in biomolecules by calculating the forces acting on each atom, considering bonded or non-bonded interactions [1, 2]. Sequence alignment methods scale the abstraction level from atoms to RNA or DNA molecules, and then up to whole genomes, to the aim of combining or interpreting nucleotide sequences by means of string-based algorithms [3]. Systems Biology considers instead the emergent properties of complex biological systems—up to whole cells and organs [4, 5]—focusing either on topological properties or flux distributions of large-scale networks, or on the dynamical behavior of their molecular components (e.g. genes, proteins, metabolites).

Although these disciplines are characterized by different goals, deal with systems at different scales of complexity and require completely different computational methodologies, they share an ideal trait d’union: all of them are computationally challenging [6–8]. Computers based on Central Processing Units...
(CPUs) are constantly improving, offering improved performances thanks to the parallelism granted by multi-threading and the vector instructions provided by e.g. Streaming SIMD Extensions (SSE) [9]. Still, computational analyses in life science disciplines often lie on the boundary of feasibility because of the huge computational costs they require on CPUs. Hence, an intense research is focusing on the optimization of algorithms and data structures in these fields; anyway, many computational methods can already benefit from non-conventional computing architectures. In particular, parallel infrastructures can be used to strongly reduce the prohibitive running times of these methods, by distributing the workload over multiple independent computing units. It is worth noting, however, that not all problems can be parallelized, as they are inherently sequential.

In the context of high-performance computing (HPC), the traditional solutions for distributed architectures are represented by computer clusters and grid computing [10, 11]. Although these infrastructures are characterized by some considerable drawbacks, in general they are largely used by the scientific community because they allow to execute the available computational methods with minimal changes to the existing CPU code. A third way to distributed computation is the emergent field of cloud computing, whereby private companies offer a pool of computation resources (e.g. computers, storage) attainable on demand and ubiquitously over the Internet. Cloud computing mitigates some problems of classic distributed architectures; however, it is affected by the fact that data are stored on servers owned by private companies, bringing about issues of privacy, potential piracy, continuity of the service, ‘data lock-in’, along with typical problems of Big Data e.g. transferring terabyte-scale data to and from the cloud [12]. An alternative option for HPC consists in the use of reconfigurable hardware platforms such as Field Programmable Gates Arrays (FPGAs) [13], which require dedicated hardware and specific programming skills for circuits design.

In the latter years, a completely different approach to HPC gained ground: the use of general-purpose multi-core devices like Many Integrated Cores (MIC) co-processors and Graphics Processing Units (GPUs). In particular, GPUs are gaining popularity, as they are pervasive, relatively cheap and extremely efficient parallel multi-core co-processors, giving access to low-cost, energy-efficient means to achieve tera-scale performances on common workstations (and peta-scale performances on GPU-equipped supercomputers [14, 15]). However, tera-scale performances represent a theoretical peak that can be achieved only by distributing the whole workload across all available cores [16] and by leveraging the high-performance memories on the GPU, two circumstances that are seldom simultaneously verified. Even in sub-optimal conditions, though, GPUs can achieve the same performances of other HPC infrastructures, albeit with a single machine and, remarkably, without the need for job scheduling or the transfer of confidential information. Being GPU’s one of the most efficient and largely exploited parallel technology, in this article we provide a review of recent GPU-based tools for biological applications, discussing both their strengths and limitations. Indeed, despite its relevant performance, also general-purpose GPU (GPGPU) computing has some drawbacks. The first is related to the fact that GPUs are mainly designed to provide the ‘Same Instruction Multiple Data’ (SIMD) parallelism, that is, all cores in the GPU are supposed to execute the same instructions on different input data (For the sake of completeness, we report that, on the most recent architectures, concurrent kernels can be executed on a single GPU, providing a hybrid SIMD-MIMD execution. Additional information about concurrent kernels is provided in Supplementary File 2). This is radically different from the ‘Multiple Instruction Multiple Data’ (MIMD) paradigm of computer clusters and grid computing, whereby all computing units are independent, asynchronous, can work on different data and execute different code. As SIMD is not the usual execution strategy for existing CPU implementations, the GPU code cannot be directly ported to the GPU’s architecture. In general, the CPU code needs to be rewritten for GPUs, which are completely different architectures and support a different set of functionalities, as well as different libraries. In addition, the complex hierarchy of memories and the limited amount of high-performance memories available on GPUs generally require a redesign of the existing algorithms, to better fit and fully leverage this architecture. Thus, from the point of view of the software developer, GPU programming still remains a challenging task [17]. Table 1 presents an overview of various HPC infrastructures, together with their architectural features, advantages and limits.

In the context of GPGPU computing, Nvidia’s CUDA (Compute Unified Device Architecture) is the most used library for the development of GPU-based tools in the fields of Bioinformatics, Computational Biology and Systems Biology, representing the standard de facto for scientific computation. CUDA can only exploit Nvidia GPUs, but alternative solutions exist, such as Microsoft DirectCompute (which can be used only with Microsoft’s Windows operating system) and the platform-independent library OpenCL (which can also leverage AMD/ATI GPUs). In this review we focus on available GPU-powered tools, mainly based on CUDA, for computational analyses in life-sciences fields. In particular, we present recent GPU-accelerated methodologies developed for sequence alignment, molecular dynamics, molecular docking, prediction and searching of molecular structures, simulation of the temporal dynamics of cellular processes and analysis methods in Systems Biology. Owing to space limits, a collection of additional applications of GPUs developed to deal with other life-sciences problems—spectral analysis, genome-wide analysis, Bayesian inference, movement tracking, quantum chemistry—is provided in Supplementary File 1. The complete list of the GPU-powered tools presented in this review is also available at http://bit.ly/gptools. Developers of GPU-based tools for the aforementioned disciplines are invited to contact the authors to add their software to the webpage.

This review is structured in a way that each section can be read independently from the others, so that the reader can freely skip topics not related to his/her own interests, without compromising the comprehension of the overall contents. The works presented in this review were chosen by taking into account their chronological appearance, preferring the most recent implementations over earlier tools, some of which were previously reviewed elsewhere [19–21]. Among the cited works, we identified, when possible, the most performing tool for each specific task, and report the computational speed-up claimed by the authors. Except where stated otherwise, all tools are assumed to be implemented using the C/C++ language.

The review is mainly conceived for end users of computational tools in Bioinformatics, Computational Biology and Systems Biology—indeedly of their educational background or research expertise—who can be well-acquainted with available CPU-based software in these fields, but might profitably find out how GPUs can give a boost to their analyses and research outcomes. In particular, end users with a main biological background can take advantage of this review to get a
The aim of Supplementary File 2 is to make this review self-archery, streaming multiprocessor, warp voting, coalesced patterns). Related to this HPC solution (e.g. thread, block, grid, memory hierarchy, streaming multiprocessor, warp voting, coalesced patterns). The aim of Supplementary File 2 is to make this review self-contained with respect to all GPU-related issues that are either mentioned or discussed in what follows. Finally, Supplementary File 3 provides more technical details (e.g. peak processing power, global memory size, power consumption) about the Nvidia GPUs that have been used in the papers cited in this review.

This work ends with a discussion about future trends of GPU-powered analysis of biological systems. We stress the fact that, except when authors of reviewed papers performed themselves a direct comparison between various GPU-powered tools, the architectural differences of the workstations used for their tests prevented us from performing a fair comparison among all different implementations. As a consequence, we shall not provide here a ranking of the different tools according to their

<table>
<thead>
<tr>
<th>HPC type</th>
<th>Architecture</th>
<th>Advantages</th>
<th>Drawbacks</th>
<th>Computing paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer cluster</td>
<td>Set of interconnected computers controlled by a centralized scheduler</td>
<td>Require minimal changes to the existing source code of CPU programs, with the exception of possible modifications necessary for message passing</td>
<td>Expensive, characterized by relevant energy consumption and requires maintenance</td>
<td>MIMD</td>
</tr>
<tr>
<td>Grid computing</td>
<td>Set of geographically distributed and logically organized (heterogeneous) computing resources</td>
<td>Require minimal changes to the existing source code of CPU programs, with the exception of possible modifications necessary for message passing</td>
<td>Generally based on ‘volunteering’: computer owners donate resources (e.g. computing power, storage) to a specific project; no guarantee about the availability of remote computers: some allocated tasks could never be processed and need to be reassigned; remote computers might not be completely trustworthy</td>
<td>MIMD</td>
</tr>
<tr>
<td>Cloud computing</td>
<td>Pool of computation resources (e.g. computers, storage) offered by private companies, available on demand and ubiquitously over the Internet</td>
<td>Mitigate some problems like the costs of the infrastructure and its maintenance</td>
<td>Data are stored on servers owned by private companies; issues of privacy, potential piracy, espionage, international legal conflicts, continuity of the service (e.g. owing to some malfunctioning, DDoS attacks, or Internet connection problems)</td>
<td>MIMD</td>
</tr>
<tr>
<td>GPU</td>
<td>Dedicated parallel coprocessor, formerly devoted to real-time rendering of computer graphics, nowadays present in every common computer</td>
<td>High number of programmable computing units allow the execution of thousands simultaneous threads. Availability of high-performance local memories</td>
<td>Based on a modified SIMD computing paradigm; conditional branches imply serialization of threads’ execution. GPU’s peculiar architecture generally requires code rewriting and algorithms redesign</td>
<td>SIMD (although temporary divergence is allowed)</td>
</tr>
<tr>
<td>MIC</td>
<td>Dedicated parallel co-processor installable in common desktop computers, workstations and servers</td>
<td>Similar to GPUs but based on the conventional x86 instructions set: existing CPU code, in principle, might be ported without any modification. All cores are independent</td>
<td>Fewer cores with respect to latest GPUs. To achieve GPU-like performances, modification of existing CPU code to exploit vector instructions are required</td>
<td>MIMD</td>
</tr>
<tr>
<td>FPGA</td>
<td>Integrated circuits containing an array of programmable logic blocks</td>
<td>Able to implement a digital circuit, which directly performs purpose-specific tasks (unlike general-purpose software tools). Such tasks are executed on a dedicated hardware without any computational overhead (e.g. those related to the operating system)</td>
<td>Generally programmed using a descriptive language (e.g. VHDL, Verilog [18]), which can be cumbersome. Debugging using digital circuits simulators might be complicated and not realistic. Experience with circuit design optimization might be necessary to execute tasks using the highest clock frequency</td>
<td>Dedicated hardware</td>
</tr>
</tbody>
</table>

Table 1. High-performance computing architectures: advantages and drawbacks
computational performance. Indeed, such a ranking would require the re-implementation and testing of all algorithms by using the same hardware as well as different problem instances, which is far beyond the scope of this review.

**Sequence alignment**

The use of parallel co-processors has proven to be beneficial for genomic sequence analysis (Table 2). In this context, the advantages achievable with GPU-powered tools is of particular importance when considering next-generation sequencing (NGS) methodologies, which allow to parallelize the sequencing process by producing a huge number of subsequences (named ‘short reads’) of the target genome, which must be realigned against a reference sequence. Therefore, in the case of high-throughput NGS methods, a typical run produces billions of reads, making the alignment problem a challenging computational task, possibly requiring long running times on CPUs.

Regardless of the used sequencing methodology, existing aligners can be roughly partitioned into two classes, according to the data structure they exploit: hash tables and suffix/prefix trees. The latter approach requires particular algorithms and data structures like the Burrows-Wheeler Transform (BWT) [43] and the FM-index [44]. In this context, multiple tools based on CUDA have already been developed: BarraCUDA [22], CUSHAW [23], GPU-BWT [24] and SOAP3 [25] (all based on BWT), and SARUMAN [26] (based on hashing).

SOAP3 is based on a modified version of BWT tailored for GPU execution—named GPU-2BWT—which was redesigned to reduce the accesses to the global memory; the access time to GPUs is slower than that of CPUs. Moreover, SOAP3 performs a pre-processing of sequences to identify those patterns—named ‘hard patterns’—that would cause a high level of branching in CUDA kernels: hard patterns are processed separately, thus reducing the serialization of threads execution. SOAP3 is also able to perform heterogeneous computation, by simultaneously leveraging both CPU and GPU. In 2013, a special version of SOAP3, named SOAP3-dp [27], able to cope with gapped alignment and implementing a memory-optimized dynamic programming methodology, was proposed and compared against CUSHAW and BarraCUDA. According to this comparison on both real and synthetic data, SOAP3-dp turned out to be the fastest implementation to date, outperforming the other methodologies, also from the point of view of the sensitivity. SOAP3-dp represents the foundation of G-SNPM [28], another GPU-based tool for mapping single nucleotide polymorphisms (SNP) on a reference genome. Moreover, SOAP3-dp is also exploited by G-CNV [29], a GPU-powered tool that accelerates the preparatory operations necessary for copy number variations detection (e.g. low-quality sequences filtering, low-quality nucleotides masking, removal of duplicate reads and ambiguous mappings). Thanks to GPU acceleration, G-CNV offers up to 18× acceleration with respect to state-of-the-art methods.

At the beginning of 2015, Nvidia published the first official release of its NVBIO [45] library, which gives access to a variety of data structures and algorithms useful for sequence alignment (e.g. packed strings, FM-index, BWT, dynamic programming alignment), providing procedures for transparent decompression and processing of the most widespread input formats (e.g. FASTA, FASTQ, BAM). Built on top of the NVBIO library, nvBowtie is a GPU-accelerated re-engineering of the Bowtie2 algorithm [30] for the alignment of gapped short reads. According to Nvidia, nvBowtie allows an 8× speed-up with respect to the highly optimized CPU-bound version. In addition to this, MaxSSmap [31] was proposed as a further GPU-powered tool for mapping short reads with gapped alignment, designed to attain a higher level of accuracy with respect to competitors.

When the reference genome is not available, the problem becomes to re-assembly *de novo* a target genome from the reads. Two GPU-based software tools are available for reads assembly: GPU-Euler [32] and MEGAHIT [33], both exploiting a de Bruijn approach, whereby the overlaps between input reads are identified and used to create a graph of contiguous sequences. Then,
the Eulerian path over this graph represents the re-assembled genome. The speed-up of GPU-Euler is about 5 $\times$ with respect to the sequential version, using a Nvidia QUADRO FX 5800. According to the authors, GPU-Euler’s reduced speed-up is owing to memory optimization: none of the high-performance memories (e.g. shared memory, texture memory) were exploited in the current implementation, although they could reduce the latencies owing to the hash table look-up. MEGAHit, instead, halves the running time of the re-assembly with respect to a sequential execution. Unfortunately, the performances of the two algorithms had never been compared.

As in the case of the problem of short reads alignment against a reference genome, the alignment of primary sequences consists in a query sequence that is compared with a library of sequences, to identify ‘similar’ ones. The most widespread algorithm to tackle this problem is the BLAST heuristic [46, 47]. The first attempts in accelerating BLAST on GPUs [34, 35] were outperformed by G-BLASTN [36], which offers a 14.8 $\times$ speed-up and guarantees identical results to traditional BLAST. An alternative algorithm for sequence alignment is the Smith-Waterman [48] dynamic programming method, which is usually impracticable for long DNA sequences owing to its quadratic time and space computational complexity. Thanks to advanced space optimization and the adoption of GPU acceleration, SW* [37] offers genome-wide alignments based on Smith-Waterman with a speed-up of two orders of magnitude with respect to equivalent CPU implementations, using a Nvidia GeForce GTX 570. Smith-Waterman is also the basis of CUDA-SW $+3$ [49], used to provide protein sequence search, based on pairwise alignment. This tool—which is the result of a long series of optimizations, outperforming all previous solutions [50–52]—represents a heterogeneous implementation able to carry out concurrent GPU and CPU executions. Both architectures are intensively exploited to maximize the speed-up: on the one hand, CUDA-SW $+3$ leverages SSE vector extensions and multi-threading on the CPU; on the other hand, it exploits PTX SIMD instructions (i.e. vector assembly code) to further increase the level of parallelism (see Supplementary File 2). According to the authors, CUDA-SW $+3$ running on a GTX690 is up to 3.2 $\times$ faster than CUDA-SW $+2$; it is also 5 $\times$ faster than SWIPE [53] and 11 $\times$ faster than BLAST+ [54], both running in multi-threaded fashion on an Intel i7 2700K 3.5 GHz CPU.

MUMmer uses an alternative approach, based on a suffix tree, requiring linear space and enabling substring matching in linear time [55]. Thanks to GPU acceleration and a careful data layout optimization, MUMmerGPU 2.0 [38] provides a 4 $\times$ speed-up with respect to classic MUMmer.

The problem of sequence similarity is also tackled by GPU/C2 CAST [39], a parallel version of the CAST software [56] ported to CUDA. CAST performs optimized local sequence similarities by detecting the ‘low-complexity regions’ (LCR), i.e. biologically unrelated sequences owing to compositionally biased sequence pairs. By masking LCR, CAST significantly improves the reliability of homology detection. Thanks to GPU acceleration, GPU/C2 CAST allows a speed-up ranging from 5 $\times$ up to 10 $\times$ with respect to the classic multi-threaded version, with a relevant part of the execution time (30% on average) owing to memory transfers.

The problem of sequence similarity, for the detection of common motifs, is tackled by the HMMER3 pipeline, which is based on profile Hidden Markov Models [57]. HMMER3 is a strongly optimized tool, fully leveraging CPU’s multi-threading and vector instructions. Hence, repeated parallelization attempts did not lead to a significant speed-up, except in the case of CUDAMPF [40], a careful implementation, which leverages multiple recent CUDA features (at the time of writing) like vector instructions, real-time compilation for loop unrolling and dynamic kernel switching according to task workloads. The reported speed-up of CUDAMPF ranges between 11 $\times$ and 37 $\times$ with respect to an optimized CPU version of HMMER3, while the GPU implementation of HMMER presented by Ganesan et al. [58] does not achieve any relevant speed-up.

The last problem we consider is the alignment of multiple sequences (MSA) for the identification of similar residues. This problem could be tackled by means of dynamic programming, but this strategy is generally unfeasible because of its exponen- tial space computational complexity [41]. An alternative approach to MSA is the progressive three-stage alignment performed by Clustal [59]: (i) pair-wise alignment of all sequences; (ii) construction of the phylogenetic tree; (iii) use of the phylogenetic tree to perform the multiple alignments. The GPU-accelerated version CUDAclustal [41] globally improved the performances by 2 $\times$ using a GeForce GTX 295, although the parallelization of the first stage—implemented by means of strip-wise parallel calculation of the similarity matrices—allows a 30 $\times$ speed-up with respect to the sequential version. In a similar vein, GPU-REMuSiC [42] performs GPU-accelerated progressive MSA. However, differently from CUDAclustal, this tool allows to specify regular expressions to apply constraints during the final alignment phase. According to [42], the speed-up of GPU-REMuSiC is relevant, especially because it is natively able to distribute the calculations across multiple GPUs.

### Molecular dynamics

The physical movements of macromolecules, such as proteins, can be simulated by means of molecular mechanics methods. This computational analysis is highly significant, as large-scale conformational rearrangements are involved in signal transduction, enzyme catalysis and protein folding [60].

Molecular dynamics [2] describes the movements of molecules in space by numerically solving Newton’s laws of motion, i.e. by calculating the force, position and velocity of each atom over a series of time steps. Molecular dynamics is computationally challenging: the length of the time step of a simulation is generally limited to $\leq 5$ fs, while the overall time of the phenomenon is, typically, in the order of ns or s. Molecular dynamics methods have been improved over the years, starting from the first 10 ps-long simulation of a molecule consisting of 500 atoms [61], passing through experiments where the movement of small enzymes was simulated on a s time scale [62], up to proteins composed of millions of atoms [63]. Being computationally intensive, many implementations of molecular dynamics algorithms started to exploit CPU-based large-scale supercomputers [64, 65]. The main limitations of these solutions regard the high costs of supercomputers, the necessity of implementing a scheduler to handle the parallel execution of the code and the maintenance issues (see Table 1).

Nowadays, there exist different molecular dynamics simulators, implemented by means of CUDA, that completely rely on GPUs (Table 3). Molecular dynamics can be parallelized at the level of atoms, or considering either the interactions among atoms or some spatial partitioning of the molecules [73]. For instance, a new algorithm for non-bonded short-range interactions within the atoms system was introduced by Liu et al. [66]. Tested on protein systems with up to 131 072 atoms, it achieved a 11 $\times$ speed-up exploiting a Nvidia GeForce 8800 GTX compared with an optimized code exploiting the SSE instruction.
set on a Pentium IV 3.0 GHz. A CUDA implementation of general-
ized explicit solvent all-atom classic molecular dynamics
within the AMBER package was introduced in [67]. The feasibil-
ity of different GPUs for molecular dynamics simulations was
evaluated considering the maximum number of atoms that
video cards could handle, according to the available memory.
Then, performance tests were conducted on protein systems
with up to 408 576 atoms; the achieved speed-up was 2–5× com-
paring the execution on different GPUs (i.e. GTX 580, M2090,
K10, GTX 680, K20X, GTX TITAN), with respect to the parallel
CPU-based implementation using up to 364 Intel Sandy Bridge
ES-2670 2.6 GHz.

Mashimo et al. [68] presented a CUDA-based implementation
of non-Ewald scheme for long-range electrostatic interactions,
whose performances were assessed by simulating protein sys-
tems with a number of atoms ranging from 38 453 to 1 004 847.
This implementation consists in a MPI/GPU-combined parallel
program, whose execution on a rack equipped with 4 Nvidia
M2090 achieved a 100× speed-up with respect to the sequential
counterpart executed on a CPU Intel E5 2.6 GHz. Finally,
OpenMM [69] is an open-source software for molecular dy-
namics simulation for different HPC architectures (it supports
GPUs with both CUDA and OpenCL frameworks). OpenMM was
tested on a benchmark model with 23 558 atoms, allowing the
simulation of tens of ns/day with a Nvidia GTX 580 and a Nvidia
K20X, (no quantitative results about the speed-up with respect
to the parallel GPU implementation using up to 364 Intel Sandy Bridge
ES-2670 2.6 GHz.

We highlight that, when implementing molecular dynamics
methods on GPUs, some general issues should be taken into ac-
count. First, GPUs are not suitable for the parallelization of every
task of kind. Some attempts tried to implement the entire mo-
elar dynamics code with CUDA, resulting in a lack of per-
formance, caused by frequent access to high-latency memories
or by functions requiring more demanding double precision ac-
curacy (to this aim, some work focused on the definition of ‘pre-
cision’ methods to avoid the necessity of double-precision
arithmetic on the GPU [74]). Other approaches exploited GPUs to
generate random numbers required by specific problems of
Dissipative Particle Dynamics (an extension of molecular dy-
namics), achieving a 2–7× speed-up with respect to CPUs [75].
Second, the optimal number of threads per block should be
carefully evaluated considering the application [76], as well as
the number of threads per atom that should be launched ac-
cording to the kernel, to the aim of increasing the speed-up
(see, for instance, the GPU implementation of PuReMD [70]).
Third, the load between CPU and GPU should be balanced so
that both devices would spend the same amount of time on
their assigned task. However, this is challenging and not every
molecular dynamics implementation that exploits both GPU
and GPU is able to fulfill this requirement. Fourth, different lan-
guages (e.g. CUDA, C, C++, Fortran) are typically used when de-
v eloping code, resulting in a hardware-specific source code,
usually hard to maintain. In these cases, minor changes in the
operating system, compiler version or hardware could lead to
 dramatic source code and compilation changes, possibly im-
pairing the usability of the application.

Having this in mind, different kinds of molecular dynamics
methods rely on hybrid implementations that exploit both CPUs
and GPUs. For instance, a hybrid CPU-GPU implementation with
CUDA of MOIL (i.e. energy-conserving molecular dynamics) was
proposed in [71]. This implementation was tested by using a
quad-core AMD Phenom II X4 965 3.4 GHz coupled with a Nvidia
GTX 480, for the simulation of molecular systems with up to
23 556 atoms, and it achieved a 10× speed-up with respect to a
strictly CPU-bound multi-threaded counterpart. As a final ex-
ample, a long time step molecular dynamics with hybrid CPU-
GPU implementation was described by Sweet et al. [72]. In this
work, GPUs accelerate the computation of electrostatics and
generalized Born implicit solvent model, while the CPU handles
both the remaining part of the computation and the communi-
cations. The performance of this method was tested on molecu-
lar systems with up to 1251 atoms, achieving a 5.8× speed-up
with respect to implementations entirely based on the GPU.

We refer the interested reader to the review presented by
Loukatou et al. [77] for a further list of GPU-based software for
molecular dynamics.

### Molecular docking

The aim of molecular docking is to identify the best ‘lock-and-
key’ matching between two molecules, e.g. protein–protein,
protein–ligand or protein–DNA complex [78]. This method rep-
resents indeed a fundamental approach for drug design [79].
Computational approaches for molecular docking usually as-
sume that the molecules are rigid, semi-flexible or flexible; in
any case, the major challenge concerns the sampling of the con-
formational space, a task that is time-consuming. In its general
formulation, no additional data other than the atomic coordi-
nates of the molecules are used; however, further biochemical
information can be considered (e.g. the binding sites of the
molecules).

One of the first attempts in accelerating molecular docking
on GPUs was introduced by Ritchie and Venkataraman [80], who
presented an implementation of the Hex spherical polar Fourier
protein docking algorithm to identify the initial rigid body stage
of the protein–protein interaction. The Fast Fourier transform
(FFT) represents the main GPU-accelerated part of the imple-
mentation, and relies on the cuFFT library [81] (see also
Supplementary File 2). FFT is calculated by means of a divide et impera algorithm, which is perfectly suitable to distribute calculations over GPU’s multiple threads. Because of that, results showed a 45× speed-up on a Nvidia GeForce GTX 285 with respect to the CPU, reducing to the order of seconds the time required for protein docking calculations.

A different GPU-powered strategy for conformation generation and scoring functions was presented by Korb et al. [82]. Considering protein–protein and protein–ligand systems (with rigid protein and flexible ligand), the authors achieved a 50× and a 16× speed-up, respectively, by using a Nvidia GeForce 8800 GT with respect to a highly optimized CPU implementation. The main bottleneck of this work concerns the performance of the parallel ant colony optimization algorithm to identify the best conformation that, compared with the CPU-based counterpart, requires a higher number of scoring function evaluations to reach a comparable average success rate.

Simosen et al. [83] presented a GPU implementation of MolDock, a method for performing high-accuracy flexible molecular docking, focused on protein–ligand complexes to search drug candidates. This method exploits a variant of differential evolution to efficiently explore the search space of the candidate binding modes (i.e. the possible interactions between ligands and a protein). This implementation achieved a speed-up of 27.4× by using a Nvidia GeForce 8800 GT, with respect to the CPU counterpart. Authors also implemented a multi-threaded version of MolDock, which achieved a 3.3× speed-up on a 4 cores Intel Core 2 with respect to the single-threaded CPU implementation. According to this result, the speed-up of the GPU implementation is roughly reduced to about 8× if compared with the multi-threaded version of MolDock.

More recent applications for molecular docking are ppsAlign [84], the protein–DNA method proposed by Wu et al. [85] and MEGADOCK [86]. ppsAlign is a method for large-scale protein structure alignment, which exploits the parallelism provided by GPU for the sequence alignment steps required for structure comparison. This method was tested on a Nvidia Tesla C2050, achieving up to 39× speed-up with respect to other state-of-the-art CPU-based methods. The protein–DNA method is a semi-flexible molecular docking approach implemented on the GPU, which integrates Monte Carlo simulation with simulated annealing [87] to accelerate and improve docking quality. The single GPU version achieved a 28× speed-up by using a Nvidia M2070 with respect to the single CPU counterpart; other tests on a cluster of GPUs highlighted that the computational power of 128 GPUs is comparable with that of 3600 CPU cores.

MEGADOCK is an approach for rigid protein–protein interactions implementing the Katchalski-Katzir algorithm with the traditional Fast Fourier transform rigid-docking scheme, accelerated on supercomputers equipped with GPUs (in particular, MEGADOCK was implemented for single GPU, multi-GPUs and CPU). The computational experiments were performed on the TSUBAME 2.5 supercomputer—having each node equipped with 3 Nvidia Tesla K20X—considering 30 976 protein pairs of a cross-docking study between 176 receptors and 176 ligands. The claimed speed-up reduces the computation time from several days to 3 h.

Finally, the docking approach using Ray Casting [88] allows a virtual screening by docking small molecules into protein surface pockets; it can be used to identify known inhibitors from large sets of decoy compounds and new compounds that are active in biochemical assays. Compared with the CPU-based counterpart, the execution on a mid-range price GPU allowed a 27× speed-up.

Table 4 lists the GPU-enabled molecular docking tools described in this section.

### Table 4. GPU-powered tools for molecular docking, along with the speed-up achieved and the solutions used for code parallelization

<table>
<thead>
<tr>
<th>Molecular docking</th>
<th>Tool name</th>
<th>Speed-up</th>
<th>Parallel solution</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex spherical polar Fourier protein docking algorithm for rigid molecules</td>
<td>–</td>
<td>45×</td>
<td>CPU-GPU</td>
<td>[80]</td>
</tr>
<tr>
<td>Conformation generation and scoring function for rigid and flexible molecules</td>
<td>–</td>
<td>50×</td>
<td>CPU-GPU</td>
<td>[82]</td>
</tr>
<tr>
<td>High accuracy flexible molecular docking with differential evolution</td>
<td>MolDock</td>
<td>27.4×</td>
<td>GPU</td>
<td>[83]</td>
</tr>
<tr>
<td>Large-scale protein structure alignment</td>
<td>ppsAlign</td>
<td>39×</td>
<td>GPU</td>
<td>[84]</td>
</tr>
<tr>
<td>Protein-DNA docking with Monte Carlo simulation and simulated annealing</td>
<td>–</td>
<td>28×</td>
<td>GPU</td>
<td>[85]</td>
</tr>
<tr>
<td>Katchalski-Katzir algorithm with traditional Fast Fourier transform rigid-docking scheme</td>
<td>MEGADOCK</td>
<td>–</td>
<td>GPU</td>
<td>[86]</td>
</tr>
<tr>
<td>Docking approach using Ray Casting</td>
<td>–</td>
<td>27×</td>
<td>CPU-GPU</td>
<td>[88]</td>
</tr>
</tbody>
</table>
distances for all candidate solutions, the computational cost of the methodology was strongly reduced [95].

Another issue in structural Computational Biology is related to the identification of proteins in databases, according to their three-dimensional conformation. The similarity between two molecules is generally assessed by means of structural alignment, which is characterized by a high computational complexity. GPU-CASSERT [96] mitigates the problem with GPUs, performing a two-phase alignment of protein structures with an average 180× speed-up with respect to its CPU-bound and single-core implementation.

Another methodology for protein searching, working at the level of secondary structures, was proposed by Stivala et al. [97]. In this work, the authors performed multiple parallel instances of simulated annealing on the GPU, strongly reducing the computational effort and obtaining a fast methodology that is comparable in accuracy with the state-of-the-art methods.

Table 5 lists the tools presented in this section, along with the speed-up obtained.

### Table 5. GPU-powered tools for prediction and searching of molecular structures

<table>
<thead>
<tr>
<th>Tool name</th>
<th>Speed-up</th>
<th>Parallel solution</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNA secondary structure with dynamic programming</td>
<td>–</td>
<td>17×</td>
<td>GPU</td>
</tr>
<tr>
<td>RNA secondary structure with Zucker algorithm</td>
<td>–</td>
<td>6.75–15.93×</td>
<td>GPU-GPU</td>
</tr>
<tr>
<td>Molecular distance geometry problem with a memetic algorithm</td>
<td>memHPG</td>
<td>–</td>
<td>GPU-GPU</td>
</tr>
<tr>
<td>Protein alignment</td>
<td>GPU-CASSERT</td>
<td>180×</td>
<td>GPU</td>
</tr>
<tr>
<td>Protein alignment based on Simulated Annealing</td>
<td>–</td>
<td>–</td>
<td>GPU</td>
</tr>
</tbody>
</table>

### Table 6. GPU-powered tools for simulation of spatio-temporal dynamics and related applications in Systems Biology

<table>
<thead>
<tr>
<th>Tool name</th>
<th>Speed-up</th>
<th>Parallel solution</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse-grain deterministic simulation with Euler method</td>
<td>–</td>
<td>63×</td>
<td>GPU</td>
</tr>
<tr>
<td>Coarse-grain deterministic simulation with LSODA</td>
<td>cupSODA</td>
<td>86×</td>
<td>GPU</td>
</tr>
<tr>
<td>Coarse-grain deterministic and stochastic simulation with LSODA and SSA</td>
<td>cuda-sim</td>
<td>47×</td>
<td>GPU</td>
</tr>
<tr>
<td>Coarse-grain stochastic simulation with SSA (with CUDA implementation of Mersenne-Twister RNG)</td>
<td>–</td>
<td>50×</td>
<td>GPU</td>
</tr>
<tr>
<td>Coarse- and fine-grain stochastic simulation with SSA</td>
<td>–</td>
<td>130×</td>
<td>GPU</td>
</tr>
<tr>
<td>Coarse-grain stochastic simulation with SSA</td>
<td>–</td>
<td>–</td>
<td>GPU</td>
</tr>
<tr>
<td>Fine-grain stochastic simulation of large scale models with SSA</td>
<td>GPU-ODM</td>
<td>–</td>
<td>GPU</td>
</tr>
<tr>
<td>Fine-grain stochastic simulation with τ-leaping</td>
<td>–</td>
<td>60×</td>
<td>GPU</td>
</tr>
<tr>
<td>Coarse-grain stochastic simulation with τ-leaping</td>
<td>cuTauLeaping</td>
<td>1000×</td>
<td>GPU</td>
</tr>
<tr>
<td>RD simulation with SSA</td>
<td>–</td>
<td>–</td>
<td>GPU</td>
</tr>
<tr>
<td>Spatial τ-leaping simulation for crowded compartments</td>
<td>STAUCC</td>
<td>24×</td>
<td>GPU</td>
</tr>
<tr>
<td>Particle-based methods for crowded compartments</td>
<td>–</td>
<td>200×</td>
<td>GPU</td>
</tr>
<tr>
<td>Particle-based methods for crowded compartments</td>
<td>–</td>
<td>135×</td>
<td>GPU</td>
</tr>
<tr>
<td>ABM for cellular level dynamics</td>
<td>FLAME</td>
<td>–</td>
<td>GPU</td>
</tr>
<tr>
<td>ABM for cellular level dynamics</td>
<td>–</td>
<td>100×</td>
<td>GPU</td>
</tr>
<tr>
<td>Coarse-grain deterministic simulation of blood coagulation cascade</td>
<td>coagSODA</td>
<td>181×</td>
<td>GPU</td>
</tr>
<tr>
<td>Simulation of large-scale models with LSODA</td>
<td>cupSODA*L</td>
<td>–</td>
<td>GPU</td>
</tr>
<tr>
<td>Parameter estimation with multi-swarm PSO</td>
<td>–</td>
<td>24×</td>
<td>GPU</td>
</tr>
<tr>
<td>Reverse engineering with Cartesian Genetic Programming</td>
<td>cuRE</td>
<td>–</td>
<td>GPU</td>
</tr>
<tr>
<td>Parameter estimation and model selection with approximate Bayesian computation</td>
<td>ABC-SysBio</td>
<td>–</td>
<td>GPU</td>
</tr>
</tbody>
</table>

### Simulation of spatio-temporal dynamics

The simulation of mathematical models describing complex biological systems allows to determine the quantitative variation of the molecular species in time and in space. Simulations can be performed by means of deterministic, stochastic or hybrid algorithms [98], which should be chosen according to the scale of the modeled system, the nature of its components and the possible role played by biological noise. In this section, we review GPU-powered tools for the simulation of spatio-temporal dynamics and related applications in Systems Biology (see also Table 6).

### Deterministic simulation

When the concentrations of molecular species are high and the effect of noise can be neglected, Ordinary Differential Equations (ODEs) represent the typical modeling approach for biological systems. Given a model parameterization (i.e. the initial state of the system and the set of kinetic parameters), the dynamics of the system can be obtained by solving the ODEs using some numerical integrator [118].

Ackermann et al. [99] developed a GPU-accelerated simulator to execute massively parallel simulations of biological molecular networks. This methodology automatically converts a model, described using the SBML language [119], into a specific CUDA implementation of the Euler numerical integrator. The CPU code used to test this simulator was completely identical to
the CUDA code, without any GPU-specific statements; specifically, no multi-threading or SIMD instructions were exploited. The evaluation of this implementation on a Nvidia GeForce 9800 GX2 showed a speed-up between 28× and 63×, compared with the execution on a CPU Xeon 2.66 GHz. In a similar vein, a CUDA implementation of the LSODA algorithm, named cuda-sim, was presented by Zhou et al. [101]. LSODA is a numeric integration algorithm that allows higher-quality simulations with respect to Euler’s method, and accelerates the computation also in the case of stiff systems [120]. The cuda-sim simulator performs the so-called ‘just in time’ (JIT) compilation (that is, the creation, compilation and linking at ‘runtime’ of new source code) by converting a SBML model into CUDA code. With respect to the GPU implementation of LSODA contained in the nummy library of Python, cuda-sim achieved a 47× speed-up.

Nobile et al. [100] presented another parallel simulator relying on the LSODA algorithm, named cupSODA, to speed up the simultaneous execution of a large number of deterministic simulations. Given a reaction-based mechanistic model and assuming the mass-action kinetics, cupSODA automatically determines the corresponding system of ODEs and the related Jacobian matrix. Differently from cuda-sim, cupSODA saves the terms of the Jacobian matrix. Differently from cuda-sim, cupSODA saves the time required to compute these terms on the GPU-side parser. cupSODA achieved an acceleration up to 86× with respect to COPASI [121], used as reference CPU-based LSODA simulator. This relevant acceleration was obtained, thanks to a meticulous optimization of the data structures and an intensive usage of the whole memory hierarchy on GPUs (e.g. the ODEs and the Jacobian matrix are stored in the constant memory, while the state of the system is stored in the shared memory). As an extension of cupSODA, coagSODA [114] was then designed to accelerate parallel simulations of a model of the blood coagulation cascade [122], which requires the integration of ODEs based on Hill kinetics, while cupSODA-L [115] was specifically designed to simulate large-scale models (characterized by thousands reactions), which have huge memory requirements owing to LSODA’s working data structures.

**Stochastic simulation**

When the effect of biological noise cannot be neglected, randomness can be described either by means of Stochastic Differential Equations [123] or using explicit mechanistic models, whereby the biochemical reactions that describe the physical interactions between the species occurring in the system are specified [124]. In this case, the simulation is performed by means of Monte Carlo procedures, like the stochastic simulation algorithm (SSA) [124].

A problematic issue in the execution of stochastic simulations is the availability of GPU-side high-quality random number generators (RNGs). Although the last versions of CUDA offer the CURAND library (see Supplementary File 2), early GPU implementations required the development of custom kernels for RNGs. This problem was faced for the CUDA version of SSA developed by Li and Petzold [102], who implemented the Mersenne Twister RNG [125], achieving a 50× speed-up with respect to a common single-threaded CPU implementation of SSA. Sumiyoshi et al. [103] extended this methodology by performing both coarse-grain and fine-grain parallelization: the former allows multiple simultaneous stochastic simulations of a model, while the latter is achieved by distributing over multiple threads the calculations related to the model reactions. The execution of SSA was optimized by storing both the system state and the values of propensity functions into the shared memory, and by exploiting asynchronous data transfer from the GPU to the CPU to reduce the transfer time. This version of SSA achieved a 130× speed-up with respect to the sequential simulation on the host computer.

Klinge et al. [104] investigated two different parallelization strategies for coarse-grain simulation with SSA: ‘fat’ and ‘thin’ threads, respectively. The former approach aims at maximizing the usage of shared memory and registers to reduce the data access time; the latter approach exploits lightweight kernels to maximize the number of parallel threads. By testing the two approaches on various models of increasing complexity, the authors showed that ‘fat’ threads are more convenient only in the case of small-scale models owing to the scarcity of the shared memory. Komarov and D’Souza [105] designed GPU-ODM, a fine-grain simulator of large-scale models based on SSA, which makes a clever use of CUDA warp voting functionalities (see Supplementary File 2) and special data structures to efficiently distribute the calculations over multiple threads. Thanks to these optimizations, GPU-ODM outperformed the most advanced (even multi-threaded) CPU-based implementations of SSA.

The τ-leaping algorithm allows a faster generation of the dynamics of stochastic models with respect to SSA, by properly calculating longer simulation steps [126, 127]. Komarov et al. [106] proposed a GPU-powered fine-grain τ-leaping implementation, which was shown to be efficient in the case of extremely large (synthetic) biochemical networks (i.e. characterized by >10⁶ reactions). Nobile et al. [107] then proposed cuTauLeaping, a GPU-powered coarse-grain implementation of the optimized version of τ-leaping proposed by Cao et al. [127]. Thanks to the optimization of data structures in low-latency memories, to the use of warp voting and to the splitting of the algorithm into multiple phases corresponding to lightweight CUDA kernels, cuTauLeaping was up to three orders of magnitude faster on a GeForce GTX 590 GPU than the CPU-based implementation of τ-leaping contained in COPASI, executed on a GPU Intel Core i7-2600 3.4 GHz.

**Spatial simulation**

When the spatial localization or the diffusion of chemical species has a relevant role on the emergent dynamics, biological systems should be modeled by means of Partial Differential Equations (PDEs), thus defining Reaction-Diffusion (RD) models [128]. Several GPU-powered tools for the numerical integration of PDEs have been proposed [129–131].

In the case of stochastic RD models, the simulation is generally performed by partitioning the reaction volume into a set of small sub-volumes, in which the molecular species are assumed to be well-stirred. This allows to exploit extended versions of stochastic simulation algorithms like SSA or τ-leaping, explicitly modified to consider the diffusion of species from one sub-volume toward its neighbors. Vigilius et al. [108] presented a GPU-powered simulator of RD models based on SSA. Pasquale et al. [109] proposed STAUCC (Spatial Tau-leaping in Crowded Compartment), a GPU-powered simulator of RD models based on the Ss-DPP algorithm [132], a τ-leaping variant that keeps into account the size of the macromolecules. According to published results [109], STAUCC achieves up to 24× speed-up with respect to the sequential execution.

Smoldyn proposes an alternative approach to stochastic RD models, where molecules are modeled as individual particles [133]. Although species move stochastically, reactions are fired deterministically; in the case of second-order reactions, two...
particles react when they are close enough to collide. Two GPU-accelerated versions of Smoldyn were proposed by Gladkov et al. [110] and by Dematte [111]. Although the former offers a greater acceleration (i.e. 200×), the latter shows another peculiarity of GPUs: the graphics interoperability, that is, the possibility of plotting the positions of particles in real time, by accessing the system state that resides on GPU’s global memory.

By changing the modeling paradigm, agent-based models (ABMs) explicitly represent the individual actors of a complex system (e.g. cells), tracking their information throughout a simulation. FLAME [112] is a general-purpose simulator of ABMs, which exploits GPU acceleration to strongly reduce the running time. It is worth noting that an alternative parallelization of ABMs by means of grid computing would not scale well: the running time could not be reduced below a fixed threshold—even by increasing the number of processors—because of memory bandwidth restrictions, which do not occur in the case of GPU acceleration [112]. A tailored GPU-powered simulator of ABMs was also developed by D’Souza et al. [113], to accelerate the investigation of tuberculosis.

A final issue worth to be mentioned is the multi-scale simulation of biological systems, ranging from intracellular gene regulation up to cell shaping, adhesion and movement. For instance, Christley et al. [134] proposed a method for the investigation of epidermal growth model, which fully leveraged GPU’s horsepower by breaking the simulation into smaller kernels and by adopting GPU-tailored data structures.

Applications in Systems Biology

The computational methods used in Systems Biology to perform thorough analyses of biological systems—such as sensitivity analysis, parameter estimation, parameter sweep analysis [135, 136]—generally rely on the execution of a large number of simulations to explore the high-dimensional search space of possible model parameterizations. The aforementioned GPU-accelerated simulators can be exploited to reduce the huge computational costs of these analyses.

For instance, cuTauLeaping [107] was applied to carry out a bi-dimensional parameter sweep analysis to analyze the insurgence of oscillatory regimes in a glucose-dependent signal transduction pathway in yeast. Thanks to the GPU acceleration, 216 stochastic simulations—corresponding to 216 different parameterizations of the model—were executed in parallel in just 2 h. cosgSODA [114] was exploited to execute one-dimensional and bi-dimensional parameter sweep analyses of a large mechanistic model of the blood coagulation cascade, to determine any alteration (prolongation or reduction) of the clotting time in response to perturbed values of reaction constants and of the initial concentration of some pivotal species. The comparison of the running time required to execute a parameter sweep analysis with 105 different parameterizations showed a 181× speed-up on Nvidia Tesla K20c GPU with respect to an Intel Core i5 CPU.

Nobile et al. [116] proposed a parameter estimation methodology based on a multi-swarm version of Particle Swarm Optimization (PSO) [94], which exploits a CUDA-powered version of SSA. This method, tailored for the estimation of kinetic constants in stochastic reaction-based models, achieved a 24× speed-up with respect to an equivalent CPU implementation. The tool cuRE [95] integrates this parameter estimation methodology with Cartesian Genetic Programming [137], to perform the reverse engineering of biochemical interaction networks. Liepe et al. [117] proposed ABC-SysBio, a Python-based and GPU-powered framework based on approximate Bayesian computation, able to perform both parameter estimation and model selection. ABC-SysBio also represents the foundation for SYSBIONS [138], a tool for the calculation of a model’s evidence and the generation of samples from the posterior parameter distribution.

Discussion

In this article we reviewed the recent state-of-the-art of GPU-powered tools available for applications in Bioinformatics, Computational Biology and Systems Biology. We highlight here that, although the speed-up values reported in literature confirm that GPUs represent a powerful means to strongly reduce the running times, many of the measured acceleration could be controversial, as there might be room for additional optimization of the code executed on the CPU. Indeed, according to the descriptions provided in the aforementioned papers, many performance tests were performed using CPU code that leverage neither multi-threading nor vectorial instructions (e.g. those offered by SSE [9] or AVX instruction sets [139]). However, some of the reported speed-up values are so relevant—e.g. the 180× acceleration provided by GPU-CASSERT [96], or the 50× acceleration provided by the molecular docking tool developed by Korb et al. [82]—that even an optimized CPU code could hardly outperform the CUDA code.

In addition, it is worth noting that many of the most performing tools required a tailored implementation to fully leverage the GPU architecture and its theoretical peak performance. For instance, the fine-/coarse-grain implementation of SSA presented by Sumiyoshi et al. [102] relies on the skillful usage of shared memory and asynchronous data transfers, the protein alignment tool GPU-CASSERT [96] relies on a highly optimized use of global memory and multiple streams of execution, overlapped with data transfers; the stochastic simulator cuTauLeaping [107] relies on GPU-optimized data structures, on the fragmentation of the execution into multiple ‘thin’ kernels, and on the crafty usage of both constant and shared memories. These works provide some examples of advanced strategies used in GPGPU computing, which make CUDA implementations far more complicated than classic CPU-bound implementations. In general, the most efficient GPU-powered implementations share the following characteristics: they leverage the high-performance memories, and try to reduce the accesses to the global memory by exploiting GPU-optimized data structures. These features seem to represent the key to successful CUDA implementations, along with optimized memory layouts [140] and a smart partitioning of tasks over several threads with limited branch divergence. Stated otherwise, we warn that a naive porting of an existing software to CUDA is generally doomed to failure.

As previously mentioned, CUDA is by far the most used library for GPGPU computing; anyway, alternative solutions exist. OpenCL, for instance, is an open standard suitable for parallel programming of heterogeneous systems [141]; it includes an abstract model for architecture and memory hierarchy of OpenCL-compliant computing devices, a C-like programming language for the device-side code and C API (Application Programming Interface) for the host-side. The execution and memory hierarchy models of OpenCL are similar to CUDA, as OpenCL exploits a dedicated compiler to appropriately compile kernels according to the available devices. Differently from CUDA, the kernel compilation phase of OpenCL is performed at runtime.
However, CUDA 7.0 introduced this possibility with the NVRTC library [142]. The difficulty in writing code with OpenCL led to the definition of tools as Swan [143], to facilitate the porting of existing CUDA code to OpenCL and minimizing the effort of code rewriting. The performances of CUDA code and OpenCL code converted with Swan have been compared [143], showing a 50% increment of the execution time of the OpenCL version: the CUDA compiler appeared to be more efficient in reducing registers usage, which affects the number of concurrently executed threads. In addition, the kernel launch cost of OpenCL is around nine times larger than CUDA, affecting the running time especially in the case of kernels with ‘short’ execution time.

On the contrary, an interesting feature of Swan [143] is that CUDA code ported to OpenCL was successfully executed both on Nvidia and AMD devices without any changes to the source code, making this tool an appealing alternative to full re-implementation. Hence, although CUDA-optimized code is still more efficient [140]—see e.g. the case of MaxSSmap [31], where the source code compiled with the last versions of the CUDA library largely outperforms OpenCL—the OpenCL library represents a viable alternative to CUDA, as it is hardware independent, and it can reduce the costs of porting and maintaining multi-platform support of applications.

Although the speed-up achieved with optimized CUDA code is already relevant, it is worth noting that the constant improvement in the fabrication process of GPU-enabled video cards is expected to further increase the efficiency gap with respect to CPUs. The speed-up of GPU-powered software is generally higher when running the code on more recent video cards, thanks to the larger number of cores and the increment of the available high-performance resources (e.g. registers, memory, cache), which remove the main limitations to a full occupancy of GPUs in many existing implementations. Figure 1 summarizes some general trends of CPUs (red dots) and of the GPUs (green squares) that were cited in this review and that are listed in Supplementary File 3.

However, the speed-up achieved with optimized CUDA code is already relevant, it is worth noting that the constant improvement in the fabrication process of GPU-enabled video cards is expected to further increase the efficiency gap with respect to CPUs. The speed-up of GPU-powered software is generally higher when running the code on more recent video cards, thanks to the larger number of cores and the increment of the available high-performance resources (e.g. registers, memory, cache), which remove the main limitations to a full occupancy of GPUs in many existing implementations. Figure 1 summarizes some general trends of CPUs (red dots) and of the GPUs (green squares) that were cited in this review and that are listed in Supplementary File 3.

Figure 1a compares the theoretical GFLOPS performance assuming double precision floating point calculations: even though both architectures are constantly improving, GPUs’ performances enhance at a faster rate (as shown by the regression lines), with the most recent architectures being almost two orders of magnitude more efficient than CPUs. Higher performances are directly reflected in higher energy requirements: Figure 1b compares the energy consumption of the two architectures. The GFLOPS-per-Watt ratio (GPWR, Figure 1c), however, represents a better measure of the efficiency of the devices than the mere power consumption: GPUs generally allow better theoretical performances with respect to CPUs, despite the higher energetic requirements. The higher GPWR of GPUs is the rationale behind the development of GPU-based supercomputers, which represents a ‘green’ alternative to conventional HPC infrastructures. Figure 1d shows that, nowadays, GPUs largely outnumber CPUs, considering the number of cores, thanks to their exponential increase on the most recent video cards. This characteristic is counterbalanced by the far lower working frequency of video cards (Figure 1e), although even CPUs frequency did not substantially improve in the last years. These data explain why GPU-powered software, which leverage the thousands of cores contained in a GPU, are expected to experience a relevant increment in the achievable speed-up if they are executed on newer architectures.

A potential drawback of GPUs is the availability of memory. As a matter of fact, many applications—in particular those...
processing genome-wide data—require a huge amount of memory, more than the few gigabytes contained on high-end GPUs at the time of writing. From the point of view of memory, CPUs still largely outperform GPUs. However, CUDA allows kernels to directly access CPU’s RAM by means of the so-called ‘pinned memory’ [144]. This type of memory is page-locked and can be directly read and written from the GPU, using Direct Memory Access through the PCI-express bus, without any involvement of the CPU. The drawback of this solution is represented by the bandwidth of PCI-express accesses, which provides a reduced rate with respect to device-to-device memory transfers [145].

cupSODA’s [115] is one example of computational tool following this strategy, where the pinned memory was leveraged to perform coarse-grain simulation of large-scale biochemical models achieving only a limited speed-up.

Taking all of these issues into consideration, it can be anticipated that the increasing availability of GPU-powered tools in various research areas of life sciences—as well as the creation of massive GPU-based infrastructures, providing scientists with hexa-scale performances—will finally enable the execution of fastest and thorough simulations and analyses of complex molecular structures, or pave the way to ambitious goals like genome-wide analyses and dynamical simulations of detailed mechanistic models of whole cells and organisms.

Key Points

• Computational methods and software tools developed in Bioinformatics, Computational Biology and Systems Biology can be computationally demanding when executed on Central Processing Units (CPUs), therefore limiting their applicability in many circumstances.

• General-purpose Graphics Processing Units (GPUs) are nowadays gaining an increasing attention by the scientific community, as they can considerably reduce the running time required by standard CPU-based software.

• The aim of this review is to provide an overview of recent GPU-powered tools developed in Bioinformatics, Computational Biology and Systems Biology, emphasizing their advantages (i.e. computational speed-up) as well as drawbacks (e.g. the necessity of algorithm redesign and tailored implementation to fully leverage the GPU architecture and its peak performance).

• In particular, we present recent GPU-accelerated methodologies developed for sequence alignment, molecular dynamics, molecular docking, prediction and searching of molecular structures, simulation of the spatio-temporal dynamics of cellular processes and related applications in Systems Biology.

• The main concepts related to GPUs, a collection of other applications in Bioinformatics and Computational Biology (spectral analysis, genome-wide analysis, Bayesian inference, movement tracking, quantum chemistry) and additional technical details about Nvidia GPUs are provided in the supplementary files.

Supplementary data

Supplementary data are available online at http://bib.oxfordjournals.org/.

References


101. Li H, Petzold LR. Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on...
Graphics processing units in bioinformatics


