The Journal of Supercomputing (2019) 75:7844–7856
https://doi.org/10.1007/s11227-018-2549-5

# ginSODA: massive parallel integration of stiff ODE systems on GPUs

**Marco S. Nobile[1,3] · Paolo Cazzaniga[2,3] · Daniela Besozzi[1] · Giancarlo Mauri[1,3]**

Published online: 24 August 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

## Abstract
Ordinary differential equations (ODEs) are a widespread formalism for the mathematical modeling of natural and engineering systems, whose analysis is generally performed by means of numerical integration methods. However, real-world models are often characterized by stiffness, a circumstance that can lead to prohibitive execution times. In such cases, the practical viability of many computational tools—e.g., sensitivity analysis—is hampered by the necessity to carry out a large number of simulations. In this work, we present ginSODA, a general-purpose black-box numerical integrator that distributes the calculations on graphics processing units, and allows to run massive numbers of numerical integrations of ODE systems characterized by stiffness. By leveraging symbolic differentiation, meta-programming techniques, and source code hashing, ginSODA automatically builds highly optimized binaries for the CUDA architecture, preventing code re-compilation and allowing to speed up the computation with respect to the sequential execution. ginSODA also provides a simplified Python interface, which allows to define a system of ODEs and the test to be performed in a few lines of code. According to our results, ginSODA provides up to a 25× speedup with respect to the sequential execution.

**Keywords** High-performance computing · Ordinary differential equations · Modeling and simulation · GPU computing · CUDA · Python · ginSODA

✉ Marco S. Nobile
nobile@disco.unimib.it

[1] Department of Informatics, Systems and Communication, University of Milano-Bicocca, Viale Sarca 336, 20126 Milan, Italy

[2] Department of Human and Social Sciences, University of Bergamo, Piazzale S. Agostino 2, 24129 Bergamo, Italy

[3] SYSBIO.IT Centre for Systems Biology, Piazza della Scienza 2, 20126 Milan, Italy

## 1 Introduction

Mathematical models of complex natural and engineering systems can be formalized by means of systems of coupled ordinary differential equations (ODEs). Systems of ODEs are indeed exploited in many applications, including mechanical vibrations, lasers, biological rhythms, superconducting circuits, insect outbreaks, chemical oscillators, genetic control systems, and chaotic waterwheels [18]. One notable example concerns the research area of systems biology, in which complex biological processes are typically modeled by means of ODE systems [9,10].

In these contexts, numerical integration methods play a major role, since in general no analytical solutions of such systems can be derived [2]. In addition, real-world models are often characterized by stiffness, a phenomenon that can lead to prohibitive execution times since; in this case, numerical integration methods achieve satisfactory results only when very short integration steps are taken [5]. Given these facts, the application of many computational tools, like sensitivity analysis or parameter sweep analysis, is often hampered by the necessity to carry out a massive number of simulations, which are needed to investigate the robustness of the system or to understand its behavior in different conditions.

The running time required by these computational analyses can rapidly overtake the capabilities of central processing units (CPUs). Among different parallel architectures, general-purpose graphics processing units (GPUs) can be exploited to efficiently overcome this limitation. GPUs are parallel multi-core coprocessors that give access to tera-scale performances on common workstations (and peta-scale performances on GPU-equipped supercomputers [8]), thus allowing to markedly decrease the running times required by traditional CPU-based software, still maintaining low costs and energetic efficiency.

In this work, we present ginSODA, a general-purpose, GPU-powered black-box numerical integrator for ODE systems characterized by stiffness. ginSODA is a user-friendly tool specifically developed for the creation and simulation of ODE-based models of stiff systems, which does not require any specific programming skill or GPU computing expertise. Thanks to the automatic offloading of the calculations onto the GPU, a massive number of simulations can be executed in a parallel fashion to efficiently test the effect of different model parameterizations or perturbations. ginSODA is based on LSODA [15], an ODE numerical integrator that can automatically switch between explicit and implicit methods to deal with stiffness [17]. Specifically, ginSODA uses meta-programming to create highly optimized CUDA kernels which are linked at run time to the pre-compiled LSODA simulator. This strategy was used in previous works, notably cuda-sim [21], which also supports stochastic differential equations. However, such tools are domain dependent, i.e., they can produce systems of ODEs tailored on specific applications. To the best of our knowledge, the only completely general-purpose alternative to ginSODA is the `odeint` facility provided by the *thrust* library [1], although it has three major drawbacks with respect to ginSODA: a *thrust*-based simulator must be implemented in C++, causing the code to become extremely complex and verbose; *thrust*, being similar to the popular boost library, is not a high-level library; the `odeint` facility is not adequate for the integration of complex ODE systems characterized by stiffness.

The paper is structured as follows: in Sect. 2, we recall the main numerical integration methods specifically developed for stiff systems and give a brief description of GPU computing; ginSODA's architecture is presented in Sect. 3; the API and the computational results are described in Sect. 4; in Sect. 5, we discuss some final remarks and future directions of this work.

## 2 Methods

### 2.1 Numerical integration of stiff systems

Given a model parameterization, the temporal dynamics of the system can be simulated by solving the ODEs using some numerical integrator, such as Euler or Runge–Kutta methods [2]. Unfortunately, systems of coupled ODEs can be prone to the well-known stiffness phenomenon [7], occurring when the system is characterized by two well-separated dynamical temporal scales. Stiffness causes the integration step size to reach extremely small values, therefore negatively affecting the overall running time. Advanced integration methods like LSODA [15] can tackle this issue, since they can recognize when a system becomes stiff and dynamically select the most appropriate integration algorithm: the Adams methods [2] in the absence of stiffness, and the backward differentiation formulae (BDF) [3] otherwise. LSODA was designed to solve ODE systems in the canonical form, i.e., defined as a set of equations of the form $\frac{\mathrm{d}X}{\mathrm{d}t} = f(X, t)$. The developer is supposed to specify the system of ODEs by implementing a custom function that is passed to the algorithm. Moreover, in order to efficiently tackle the integration when dealing with stiff systems, the Jacobian matrix associated with the system must be calculated and implemented as a custom function as well.

The implementation of these functions (the Jacobian matrix in particular) can be difficult, time-consuming, and error-prone. Moreover, complex analysis methods (e.g., sensitivity analysis or parameter fitting) require the systematic modification of such functions and the execution of a massive number of simulations with different parameterizations. The next sections will describe how ginSODA automatically handles the aforementioned issues by leveraging GPU computing and by using symbolic derivation, automatic CUDA code creation, and JIT compilation.

### 2.2 GPGPU computing

General-purpose GPU (GPGPU) computing provides programmers with the possibility of simultaneously leveraging classic CPUs and GPUs for scientific calculation. The idea behind GPGPU computing is to take advantage of the massive number of cores contained in a GPU to distribute the calculations needed to perform some complex computational task. Since the GPUs' cores were designed to perform real-time rendering of 3D graphics, they are very simple (with respect to modern CPUs) and supposed to perform similar tasks on different data (notably, to calculate vertex transforms and shading). This kind of paradigm is known as *same instruction multiple data* (SIMD).

**Table 1** Main CUDA memory types

| Memory name | Size | Data scope | Data persistence | Performance |
|---|---|---|---|---|
| Global memory | GBs/GPU | Readable and writable by both host and grid | Process execution | High latency |
| Local memory | GBs/GPU | Readable and writable by both host and grid | Process execution | High latency |
| Constant memory | KBs/GPU | Readable from grid, writable from host | Process execution | Low latency (cached) |
| Shared memory | KBs/STM | Readable and writable by threads in the same block | Kernel execution | Low latency |
| Registers | KBs/thread | Readable and writable by single thread | Kernel execution | No latency |

Nvidia's *Compute Unified Device Architecture* (CUDA) is a programming model and architecture for GPGPU computing, which offers native compilers and debuggers for C/C++ and FORTRAN source codes, along with specific bindings for the most widespread languages (e.g., Java, Python). CUDA is based on the concept of *kernel*, that is, a function that is executed by the cores of a device (i.e., the GPU), performing the same calculation on different data. Although this approach is the best option to achieve the highest performances, CUDA leaves the possibility of a branched execution (e.g., IF/THEN/ELSE constructs): this flexibility comes at the cost of a serialized execution that, ultimately, affects the overall performances.

Whenever the host (i.e., the CPU) runs a kernel, the latter is replicated in multiple copies named *threads*, which are organized in logical *blocks*. The blocks are queued by the CUDA scheduler and distributed for execution on the multi-core streaming multiprocessors (STMs) available in the GPU. The blocks are also organized in a further logical structure named *grid*.

This complex execution hierarchy is not the only difference with respect to classic multi-threaded environments. GPUs are also characterized by a variety of memories; the most relevant CUDA memory types are summarized in Table 1. Each memory type has a specific visibility, dimension, and it is characterized by a different performance in terms of access latencies. Low latency memories must be leveraged as much as possible, in order to achieve peak performances on the GPU.

## 3 ginSODA architecture

ginSODA is a Python package designed to provide users with a simple API for the creation and simulation of ODE-based models of complex systems. Once a model is formalized, the effect of multiple different parameterizations (i.e., initial state, model parameters) can be tested in parallel thanks to the automatic offloading of the calculations to the GPU.

ginSODA is based on the ODE integration method LSODA [15], an ODE solver that automatically switches between explicit and implicit methods to deal with stiff

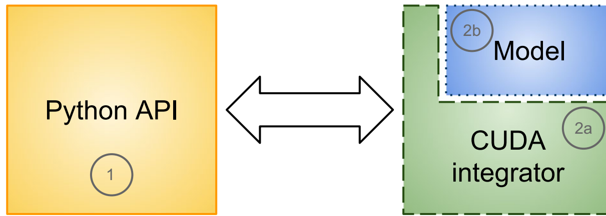


**Fig. 1** ginSODA is composed of two main components: the Python interface and the CUDA implementation. The latter can be decomposed in the (static) CUDA integrator and the model implementation produced by the user using the Python interface

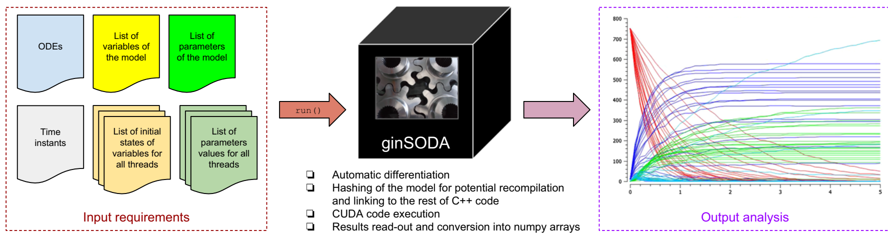


**Fig. 2** ginSODA functioning. The user creates a `ginSetup` object that is used to provide ginSODA with a model (i.e., ODEs, variables, and parameters), the list of parameters and initial conditions to be tested, and the time instants for the sampling. ginSODA then acts as a black box, performing symbolic differentiation and determining whether the model changed with respect to the last run. In the latter case, the CUDA source code, encoding the ODEs and the Jacobian matrix, is recreated by means of meta-programming, compiled and linked to the rest of the C++ integrator. The numerical integration is then performed with LSODA on the GPU, and the output is collected and converted into `numpy` arrays for further processing

systems [17]. Since LSODA leverages implicit integration (specifically, BDFs) in the presence of stiffness, it requires the calculation of a Jacobian matrix which must be provided as a C function pointer. ginSODA hides this complexity by leveraging the `sympy` module for symbolic derivation, to the aim of automatically implementing the CUDA code corresponding to the Jacobian matrix and passing the function pointer to the main static simulator.

ginSODA is architecturally composed of two main parts: (1) the python API and (2) the CUDA implementation. The latter part can be further decomposed into two elements: (2a) the static main integration part and (2b) the CUDA code implementing the model specified by the user using the API (Fig. 1).

As shown in Fig. 2 and further described in Sect. 4, the user only interacts with the API. The main integrator (2a) is fixed and not modifiable, generated during ginSODA's first execution and fully integrated with the main API. The code contained in (2b) is generated at run time by ginSODA, implementing the calculations corresponding to the model specified by the user.

Thanks to this architectural decomposition, ginSODA can leverage object-caching and recompile a small portion of the whole CUDA binary every time the model is changed. In order to recognize an actual modification of the model, ginSODA exploits a hash function (specifically, the sha512 algorithm [16]) calculated over the CUDA

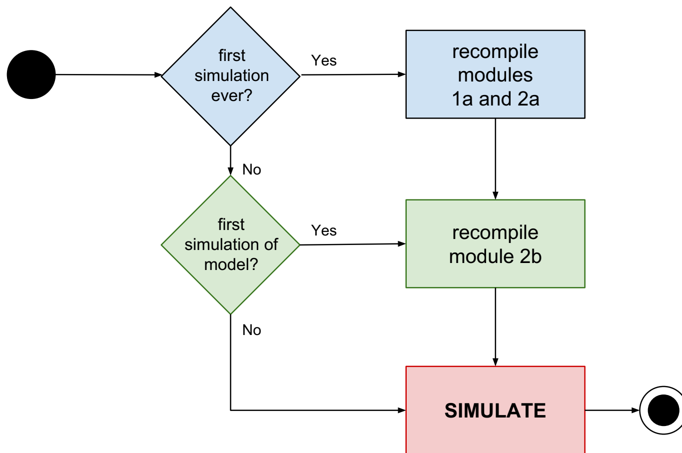


**Fig. 3** Scheme of the partial JIT compilation performed by ginSODA: only those parts that changed from the previous simulation are rebuilt and linked to the rest of the simulator

code of component (2b). If the hash value has changed (i.e., when the model being integrated is not identical to the previous one), ginSODA performs the following actions:

- the code in (2b) is recompiled;
- the object corresponding to (2b) is linked to (2a);
- the final executable binary file (2) = (2a) + (2b) is created and launched;
- at the end of the execution, the hash value associated with (2b) is stored into the model's directory.

When the hash is detected as unchanged with respect to the previous launch, the binary is immediately executed. So doing, ginSODA prevents a full JIT compilation or a GPU-side parsing of complex data structures, as in the case of cupSODA [12, 13], cuTauLeaping [14], or LASSIE [19], reducing the overall execution time (see Fig. 3).

One sensitive topic in GPGPU computing is the optimal selection of the number of blocks $\mathcal{B}$ and of threads per block $\mathcal{T}$. Given a total number of threads $N$, ginSODA tries to saturate the GPU's resources by creating warps of $\mathcal{T} = 64$ threads per block and by setting $\mathcal{B} = N/\mathcal{T}$. Moreover, if enough shared memory is available on the STMs, ginSODA accelerates the simulations by using these high-performance memory banks to store the current state of the model, strongly reducing the memory access latencies. Finally, ginSODA automatically queries the GPU to determine the compute capability, in order to create lightweight compiled binaries, saving compilation time and tailoring the executable file on the underlying architecture. Thanks to these heuristics, ginSODA automatically optimizes the kernel launches, completely hiding the architectural details and the complexity of GPUs from the user.

## 4 Results

In this section, we describe ginSODA's API and analyze its computational efficiency with respect to classic CPU-bound integrators. In order to compare ginSODA's performances with respect to traditional CPU-powered simulation, we performed multiple tests on the models described hereafter running an increasing number of parallel simulations. ginSODA's API is designed to be as simple as possible, exposing a few mandatory objects and methods, while hiding the largest part of the complexity related to sophisticated integration methods and GPU programming. We will use two example models to explain the ginSODA work flow.

The workstation used for the tests was equipped with a CPU Intel(R) Core i7-4790 K CPU (clock 4.00 GHz) and GPU GeForce GTX Titan X (clock 1075 MHz, 3072 cores, compute capability 5.2). All tests were performed on a machine with Ubuntu OS release 16.04 LTS; Python release 2.7.12; CUDA version 8.0. Python libraries used in the test: `numpy` 1.14.2; `scipy` 0.18.1; `sympy` 1.0; `pycuda` 2017.1.1. The settings used in all tests that follow are: absolute error tolerance: $1 \times 10^{-6}$; relative error tolerance: $1 \times 10^{-4}$; maximum steps: 500.

### 4.1 Competing species model

Let us assume the classic model of two species (denoted by $y_1$ and $y_2$) competing for resources:

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = y_1 \left(1 - y_1 - y_2\right),$$
$$\frac{\mathrm{d}y_2}{\mathrm{d}t} = y_2 \left(k_0 - y_2 - k_1 \cdot y_1\right).$$

If $k_0 = \frac{3}{4}, k_1 = \frac{1}{2}$, this system is characterized by four stable points: three are trivial—i.e., $(y_1, y_2) = (0, 0), (0, \frac{3}{4}), (1, 0)$—while the fourth is the solution $(y_1, y_2) = (\frac{1}{2}, \frac{1}{2})$.

The following Python code illustrates how to implement and study this model using ginSODA's API. The code is broken down into three separate listings for the sake of simplicity: Listing 1 presents the ODE model definition, Listing 2 shows an example of parameters generation, and Listing 3 describes how to launch the parallel simulations and collect the results.

The first step is to import the `ginSetup` class from `ginsoda` (Listing 1, line 1), whose constructor loads all dependencies and is responsible for the definition of all data structures needed by ginSODA. We also import `numpy` (line 2) that is used to easily generate the numeric data structures needed for the model definition. Then, the wrapper method `specify_model` loads the variables names, the parameters names, and the whole system of ODEs (lines 5–12).

**Listing 1** Example of model specification.

```
1  from ginsoda import ginSetup
2  import numpy as np
3
4  GS = ginSetup()
5  GS.specify_model(
6    variables  = ["y1", "y2"],
7    parameters = ["k0", "k1"],
8    equations  = [
9      "y1*(1.-y1-y2)",
10     "y2*(k0-y2-y1*k1)"
11   ]
12 )
```

The next step is to specify to ginSODA the actual parameterization (i.e., the initial values of the variables and the parameters) to be used in the parallel ODE integrations. One example of this task is shown in Listing 2. At line 13, the variable THREADS denotes the number of model instances that will be integrated; in this example, we will run in parallel 4096 model instances. The second step is to specify the parameters of the model (line 14): in this example, the vector parameters is duplicated THREAD times, so that all parallel instances of the model will share the same parameterization. In lines 15–19, we create and populate a vector initial_values, which contains the initial amounts of the variables in the system. In this example, each instance of the model to be integrated is characterized by a different initial condition.

**Listing 2** Specification of the model parameterizations to be tested.

```
13 THREADS         = 4096
14 parameters      = [[3./4, 1./2]]*THREADS
15 initial_values  = []
16 for x in xrange(THREADS):
17     a = 1.*x/(THREADS+1)
18     b = 1.*x/(THREADS+1)*.5
19     initial_values.append([a,b])
```

Finally, the 4096 model instances are integrated in parallel by using the run method provided by the ginSetup class (see Listing 3). The method requires three mandatory arguments: the initial state of the model (line 23); the parameters of the model (line 24); and a list of time points, in which the state of the variables is sampled (line 25). The simulation automatically terminates after the last sampling time. Once the batch of parallel integrations is completed, ginSODA stores the results in the result variable (line 22). This variable is a list of THREADS elements, one for each model instance that was simulated. Each element of this list is a numpy matrix containing the values of the variables, sampled at every time point specified in the time_instants argument. This structured output data can be directly manipulated in Python, e.g., the result variable can be plotted using matplotlib or equivalent libraries.

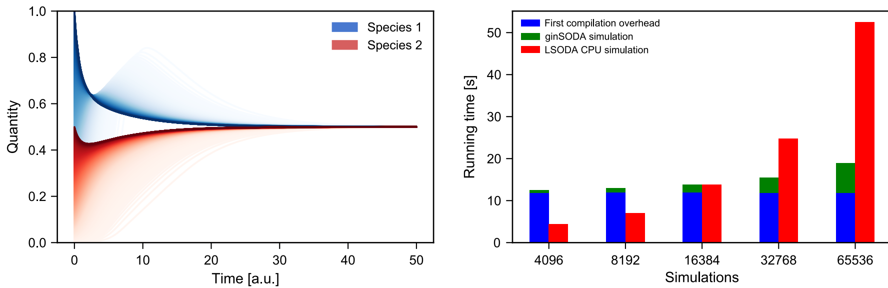


**Fig. 4** Dynamics of the competing species model simulated with ginSODA with multiple parameterizations (left panel). ginSODA strongly reduces the running time with respect to the sequential execution (right panel), although the overhead due to the first compilation is not negligible

**Listing 3** Executing the batch of parallel integrations.

```
20 time_max = 50
21 samples  = 100
22 result = GS.run(
23     initial_values=initial_values,
24     parameters=parameters,
25     time_instants=np.linspace(0, time_max, samples)
26 )
```

Figure 4 (left panel) shows the output of the 4096 parallel simulations performed with ginSODA: despite the different initial conditions, all ODEs converge to the fixed point $\left(\frac{1}{2}, \frac{1}{2}\right)$. This computational analysis can be leveraged to investigate the attractors of a system, or its robustness with respect to perturbations. Figure 4 (right panel) shows the overall running time for an increasing number of parallel simulations. In the case of ginSODA, this value can be decomposed into the compilation time (due to the first compilation when a new model is simulated, represented in blue) and the actual simulation time (represented in green). The execution time of the CPU-bound LSODA, implemented in the `scipy` library, is represented in red; since it does not require a compilation step, this running time is not further decomposed.

These results show that the time spent for the compilation of submodule (2b) (approximately 11.9 seconds) can be larger than the simulation itself (7.03 seconds in the case of 65536 parallel simulations). Hence, for the first execution of this model, the break-even with respect to the CPU was around 16384 simulations. When the module (2b) is detected as available, ginSODA directly performs the integration and largely outperforms the CPU even when a few threads are executed: in the case of 4096 simulations, ginSODA requires 0.61 seconds, while the CPU takes 4.4 seconds to complete the integration.

### 4.2 Oregonator model

As an example of stiff ODE system, we consider the Oregonator [6], a simple autocatalytic model of the oscillatory dynamics of the Belousov–Zhabotinsky chemical reaction [20]:

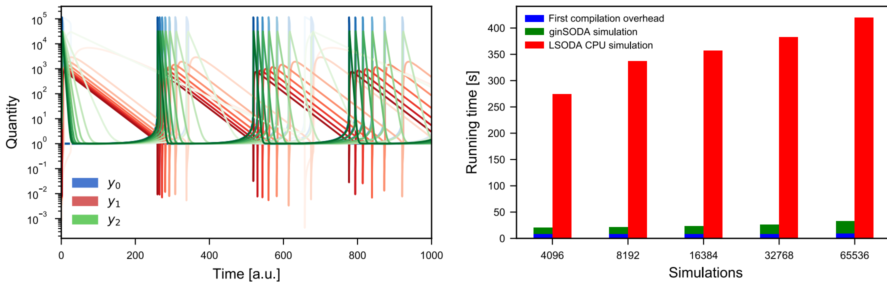


**Fig. 5** PSA of the Oregonator model (left panel): the change in the kinetic parameter $k_3$ causes a modification in the oscillations frequency. ginSODA strongly reduces the running time with respect to sequential execution (right panel)

$$\frac{dy_1}{dt} = k_0 \cdot (y_2 + y_1 \cdot (1 - k_1 \cdot y_1 - y_2)),$$
$$\frac{dy_2}{dt} = k_2 \cdot (y_3 - ((1 + y_1) \cdot y_2)),$$
$$\frac{dy_3}{dt} = k_3 \cdot (y_1 - y_3),$$

where, in the original formulation, $k_0 = 77.27$, $k_1 = 8.375 \times 10^{-6}$, $k_2 = 0.013$, and $k_3 = 0.161$. The stiffness of the Oregonator is due to the fast exchange of species $y_0$ and $y_2$ with respect to $y_1$, a situation that leads either to wrong solutions in the case of traditional ODE solvers or to extremely small integration steps in the case of adaptive step-size methods. ginSODA's integration method (LSODA) automatically detects the stiffness and internally resorts to BDF [3] to process the stiff regions. Listing 4 shows the source code corresponding to the implementation of the Oregonator.

**Listing 4** Implementation of the Oregonator model with ginSODA.

```
from ginsoda import ginSetup
import numpy as np

GS = ginSetup()
GS.add_variables(["y1", "y2", "y3"])
GS.add_parameters(["k0", "k1", "k2", "k3"])
GS.add_equations([
   "k0*(y2+y1*(1.-k1*y1-y2))",
   "k2*(y3-((1+y1)*y2))",
   "k3*(y1-y3)"
])
```

As an example of investigation performed with ginSODA, we now consider the case of parameter sweep analysis (PSA), in which the parameter $k_3$ is systematically perturbed to observe the change in the overall behavior of the system. Listing 5 explains how to perform the PSA: the initial state is identical for all threads (line 13), while the kinetic parameter $k_3$ is systematically varied (lines 14–16).

**Listing 5** PSA on parameter $k_3$ of the Oregonator model.

```
12 THREADS = 32
13 initial_y = [[4., 1.1, 4.]]*THREADS
14 parameters = []
15 for x in xrange(THREADS):
16     parameters.append([77.27, 8.375e-6, 0.013, 0.161*(0.1*x+.1)])
17 result = GS.run(
18   initial_values=initial_y,
19   parameters=parameters,
20   time_instants=linspace(0, 1000, 10000)
21 )
```

Figure 5 (left panel) shows the results of the PSA, highlighting how the variation of the parameter $k_3$ affects the period of the oscillations (for the sake of clarity, we report in the figure only a subset of the output dynamics). ginSODA performed the integration efficiently, leveraging the BDF during the "spikes" of the dynamics across all integrations. The switching to BDF was performed automatically and in a transparent way for the user, thanks to ginSODA's automatic code generation facilities.

In the case of a model characterized by a complex emergent dynamics, like the Oregonator, according to our tests ginSODA largely outperforms the CPU independently from the number of parallel simulations executed, with a speedup that reaches 25× (Fig. 5, right panel). It is worth noting that LSODA, like many similar complex integration methods, is not re-entrant and only a single instance of the algorithm can run on the CPU at the same time. Thus, the gap of performances with respect to ginSODA cannot be mitigated using a multi-threaded approach.

## 5 Conclusion

In this paper, we presented ginSODA, a Python module for the automatic offload to the GPU of multiple simulations of ODE models characterized by stiffness. ginSODA is composed of a simplified API for the definition of the model, as well as for the variables or parameters to be tested. Thanks to a set of heuristics, ginSODA can be used as a black-box tool without any previous knowledge of numerical integration or GPU programming. ginSODA is based on the LSODA algorithm, an adaptive integration method which automatically switches between explicit and implicit integration methods according to the stiffness of the system.

To provide LSODA with the necessary functions implementing the ODEs and the associated Jacobian matrix, ginSODA automatically performs symbolic derivation and partial JIT compilation. A caching system, based on a hash function, prevents unnecessary re-compilation of already simulated models.

According to our tests, ginSODA allows a speedup up to 25× with respect to the CPU-based LSODA algorithm, although the overhead due to JIT compilation can affect the performances during the first execution of a number of parallel simulations of small models.

Currently, ginSODA is implemented as a mixture of Python (i.e., module ①) and C++/CUDA code (i.e., module (2a)). As future development, we plan to port the whole code of ginSODA to PyCUDA [11] in order to provide a simpler and monolithic Python

implementation. This solution would also prevent the need for intermediate files, thus increasing the performances.

The current implementation of ginSODA leverages a single GPU, although the API provides a set of optional methods to target alternative GPUs contained in the host machine. As future development, we will extend ginSODA to automatically identify multiple GPUs, create tailored and optimized binary CUDA executable files, and distribute the simulations over the CUDA-enabled available GPUs in a transparent way with respect to the user.

ginSODA will be integrated in the COSYS infrastructure for systems biology [4], providing a means for GPU-accelerated deterministic biochemical simulation supporting arbitrary kinetics, a characteristic that is still missing in simulators developed for mass action-based models, like cupSODA [12,13].

ginSODA is open-source and cross-platform (provided that a Nvidia GPU is available on the machine) and can be freely downloaded from GitHub at the following address: https://github.com/aresio/ginSODA.

## References

1. Bell N, Hoberock J (2011) Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems Jade Edition, pp 359–371. Elsevier
2. Butcher JC (2008) Numerical methods for ordinary differential equations. Wiley, Chichester
3. Cash JR (2000) Modified extended backward differentiation formulae for the numerical solution of stiff initial value problems in ODEs and DAEs. J Comput Appl Math 125(1–2):117–130
4. Cumbo F, Nobile MS, Damiani C, Colombo R, Mauri G, Cazzaniga P (2017) COSYS: a computational infrastructure for systems biology. In: Bracciali A, Caravagna G, Gilbert D, Tagliaferri R (eds) Computational intelligence methods for bioinformatics and biostatistics. Lecture Notes in Bioinformatics, vol 10477. Springer, Berlin, pp 82–92
5. Curtiss CF, Hirschfelder JO (1952) Integration of stiff equations. Proc Natl Acad Sci 38(3):235–243
6. Field RJ, Noyes RM (1974) Oscillations in chemical systems. IV. Limit cycle behavior in a model of a real chemical reaction. J Chem Phys 60(5):1877–1884
7. Higham DJ, Trefethen LN (1993) Stiffness of ODEs. BIT Numer Math 33(2):285–303
8. Joubert W, Archibald R, Berrill M, Brown WM, Eisenbach M, Grout R, Larkin J, Levesque J, Messer B, Norman M (2015) Accelerated application development: the ORNL Titan experience. Comput Electr Eng 46:123–138
9. Kitano H (2001) Foundations of systems biology. The MIT Press, Cambridge
10. Kitano H (2002) Computational systems biology. Nature 420(6912):206
11. Klöckner A, Pinto N, Lee Y, Catanzaro B, Ivanov P, Fasih A (2012) PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. Parallel Comput 38(3):157–174
12. Nobile MS, Besozzi D, Cazzaniga P, Mauri G (2014) GPU-accelerated simulations of mass-action kinetics models with cupSODA. J Supercomput 69(1):17–24
13. Nobile MS, Besozzi D, Cazzaniga P, Mauri G, Pescini D (2013) cupSODA: a CUDA-powered simulator of mass-action kinetics. In: Malyshkin V (ed) Parallel computing technologies. Lecture Notes in Computer Science, vol 7979. Springer, Berlin, pp 344–357
14. Nobile MS, Cazzaniga P, Besozzi D, Pescini D, Mauri G (2014) cuTauLeaping: A GPU-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems. PLoS ONE 9(3):e91963
15. Petzold L (1983) Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. SIAM J Sci Stat Comput 4:136–148
16. Secure Hash Standard (SHS) (2015) Federal Information Processing Standards Publication. https://csrc.nist.gov/publications/detail/fips/180/4/final
17. Söderlind G, Jay L, Calvo M (2015) Stiffness 1952–2012: sixty years in search of a definition. BIT Numer Math 55(2):531–558

18. Strogatz SH (2018) Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering. CRC Press, Boca Raton
19. Tangherloni A, Nobile MS, Besozzi D, Mauri G, Cazzaniga P (2017) LASSIE: simulating large-scale models of biochemical systems on GPUs. BMC Bioinform 18(1):246
20. Zhabotinsky AM (1991) A history of chemical oscillations and waves. Chaos: an interdisciplinary. J Nonlinear Sci 1(4):379–386
21. Zhou Y, Liepe J, Sheng X, Stumpf MPH, Barnes C (2011) GPU accelerated biochemical network simulation. Bioinformatics 27(6):874–876