Communications in Nonlinear Science and Numerical Simulation

Program package MPGOS: challenges and solutions during the integration of a large number of independent ODE systems using GPUs

--Manuscript Draft--

Manuscript Number:	CNSNS-D-19-01951R1
Article Type:	Research Paper
Section/Category:	Numerical methods for ordinary and partial differential equations(Feng)
Keywords:	ordinary differential equations; non-linear systems; GPU programming; massively parallel architecture
Corresponding Author:	Ferenc Hegedus, Ph.D. Budapest, HUNGARY
First Author:	Ferenc Hegedus, Ph.D.
Order of Authors:	Ferenc Hegedus, Ph.D.
Abstract:	Challenges and efficient solution techniques during the integration of a large number of independent ordinary differential equations (ODEs) using the massively parallel architecture of graphics processing units (GPUs) are presented. One of the main difficulties is the minimisation of the memory transactions through the PCI-E bus between the host (CPU) and the device (GPU) required frequently, for instance, during the calculation of the Lyapunov exponent, winding number or maximum response diagram. The second difficulty is the minimisation of the slow global memory transactions \Red{and memory usage} by exploiting the memory hierarchy of the GPU architecture. \Red{Finally, a good GPU solver has to treat the possible asynchronous features of the ODE systems efficiently; for instance, event detection occurring at distinct time instances or handling the orders of magnitude difference in the required number of time steps of the different ODE systems.} The program package MPGOS (written in C++ and CUDA C software environments) can address the aforementioned issues easily via the addition of user-defined functions that must be implemented similarly to the right-hand side of the system; via the possibility of the definition of shared parameters common to all instances of the independent ODE systems; \Red{via user-programmable parameters to store only the desired properties of the trajectories; and via an easy was to overlap GPU and CPU computations}. This paper focuses on the detailed description of the implementation strategies of the program package.

Cover Letter

Ferenc Hegedűs
associate professor
Budapest University of Technology and Economics
1111 Hungary
Budapest, Műegyetem rkp. 3
+36 1 463-1680
fhegedus@hds.bme.hu

Dear Editor,

In the submitted manuscript, I discuss the possible issues and challenges that can appear during the integration of a large number of independent ODE systems on GPUs. From application point of view, the manipulation of the trajectory of a non-linear system is usually necessary between several integration phases (e.g. during the computations of Lyapunov exponent). Such manipulation is usually required even during in the middle of an integration process in case of non-smooth dynamical system (e.g. in case of impact dynamics). Moreover, there can be many other problems where the performance of a massively parallel GPU code can be degraded significantly if the user has to do these manipulations on the CPU side.

The recently developed program package MPGOS addresses these issues. In the present paper I intend to introduce the implementation strategies to minimise the PCI-E and global memory transactions, to avoid the necessity of dense output and to exploit the memory hierarchy of the GPUs. These are mandatory to avoid performance degradation using GPUs.

Since the integration of a large number of independent ODE systems is important in many non-linear applications (detailed parameter studies or the investigation of multi-stability), I think that this paper suits to the profile of “Communications in Nonlinear Science and Numerical Simulation” very well.

Thank you for your consideration.

Yours sincerely,
Ferenc Hegedűs

*Declaration of Interest Statement

Ferenc Hegedűs
associate professor
Budapest University of Technology and Economics
1111 Hungary
Budapest, Műegyetem rkp. 3
+36 1 463-1680
fhegedus@hds.bme.hu

Dear Editor,

The author(s) declare that they have no conflict of interest.

Yours sincerely,
Ferenc Hegedűs

Credit Author Statement

Ferenc Hegedűs
associate professor
Budapest University of Technology and Economics
1111 Hungary
Budapest, Műegyetem rkp. 3
+36 1 463-1680
fhegedus@hds.bme.hu

Dear Editor,

Since this paper is a single-author publication. All of the presented work is my contribution alone.

Yours sincerely,
Ferenc Hegedűs

Highlights:

- GPU accelerated integration of ODE systems.
- Minimisation of PCI-E and global memory transactions (including the dense output).
- Efficient treatment of non-smooth, impacting dynamics.
- Exploitation of shared memory in the semi-discretisation of PDEs.

Reviewer #1:

This paper describes a package, developed by the author in C++ and CUDA, that addresses the massive numerical integration of ODEs exploiting the parallel architecture of graphic processing units. The paper describes the strategies to optimise the computations as well as its implementation. The paper is interesting and well-written and it deserves to be published. However, some clarifications are needed.

Dear Reviewer, I would like to thank you for the effort put into the manuscript, for the very kind review and for finding the content interesting. I extended the manuscript according to your suggestions. In addition, I included further details and reorganisation/renaming of the sections. This paper was submitted almost a year ago. Since then, I have more experience (also with other program packages). Thus, there are issues I forget to include (e.g., overlapping CPU and GPU computations), or I thought that some technique should be trivial but testing other program packages it turned out not to be general (e.g., using a single monolithic kernel or asynchronous solution of the ODE systems). I hope that you will find the extended content appropriate and interesting (marked by red colour in the manuscript). I summarized my detailed answer for the raised issues below.

1) There is little mention to other libraries/packages for the numerical integration of ODEs in GPUs. The text should mention the advantages/disadvantages of MPGOS when compared with them.

Thank you for your remark. Now throughout the manuscript, comparison with the program packages ODEINT (C++) and DifferentialEquations.jl (Julia) are continuously done (where it was appropriate). In addition, in section Summary, comparison with 2 more program packages were made.

2) I also miss some concrete examples showing the effectiveness of the package. For instance, at the end of Section 4.1 is it claimed that "the technique described above can have a significant positive effect on the performance." An example would be great to show it. A similar comment can be done about Sections 4.2 and 4.3.

I was thinking a lot how to address this issue efficiently. With some of my colleagues we already submitted a follow-up paper where such performance comparisons were done in great details. We used different program packages with different mathematical models as test cases. And we discussed the efficiencies employing both CPUs and GPUs. The present manuscript focuses more on the theoretical issues and solution techniques using a GPU as hardware. Performance characteristics is huge topic, and I did not wanted to include it to remain focused and avoid self-plagiarism, and the details of the performance comparisons can already be found elsewhere. Although, the follow-up paper is submitted recently (thus it is not available for the general readers yet), at least the performance curves are already published for instance in the official website of MPGOS: www.gpuode.com (page CAPABILITIES AND REFERENCES). It is clear that my GPU solver is superior in many ways.

Moreover, the examples are useful to clarify what your package does and what does not.

Thank you very much for this remark. It suggest that the main features of the program packages is poorly explained. Therefore, I extended the end of Sec. 2 to explain

the storage of the different parameters; included a completely new section (Sec. 3) to describe the workload distribution. And Sec. 4 (old Sec. 3) is extended considerably to explain the workflow of the solver better. I felt that it was much more feasible to make these explanations in Sec. 2-4, rather than in Sec. 5 (old Sec. 4).

3) It is not specified which time steppers are included in MPGOS, and they should be specified. Is it possible/difficult for the user to use their own time-steppers? What about implicit time steppers? Some of them require to solve linear systems (this is specially true for ODEs coming from discretization of PDEs),

Unfortunately, only non-stiff Runge—Kutta solvers are available in MPGOS: a fourth order classic Runge—Kutta with fixed time stepping, and the adaptive Runge—Kutta—Cash—Karp method (order five with fourth order embedded error estimation). However, MPGOS is superior when these solvers are suitable (see again www.gpuode.com). The code is fully open source; thus, anybody can modify the code and write an own stepper. However, it is not supported officially with proper interface in the present version.

so I understand that the solver runs on the GPU, right? Could you explain this a bit? Which solvers are used?

I included a completely new section (Sec.3) and extended Sec. 4 (old Sec. 3) considerably to explain in more details how the solver works.

Sometimes these matrices are sparse, how is this handled?

I explained this at the end of Sec. 2. Right now the parameters and also the shared parameters are stored as 1D linear vector. Therefore, it is the responsibility of the user for bookkeeping of the physical content of the vector elements, and use them accordingly inside ODE functions.

4) In Section 6, some issues regarding parallelism are discussed. It is not clear which strategy is used to distribute tasks when their number is larger than the number of GPUs. It is a static distribution? a dynamic distribution? This should be discussed a bit.

The aforementioned completely new section (Sec. 3) serves to address this issue and provide more detailed explanation. The parallelisation strategy is actually very simple, a single GPU thread is assigned to a single instance of the ODE system. MPGOS also uses a so-called solver object to manage the workflow and distribute tasks to many GPUs or overlap CPU and GPU computations, see also Sec. 5.4.

Minor points:

p2, l22: "are" -> "is"

p3, l82: "ordinary differential equation systems" -> "ordinary differential equations"

p3, l85: "have" -> "has"

Thank you very much for finding these typos. They are corrected.

Program package MPGOS: challenges and solutions during the integration of a large number of independent ODE systems using GPUs

Ferenc Hegedűs[a,*]

[a] *Department of Hydrodynamic Systems, Faculty of Mechanical Engineering, Budapest University of Technology and Economics, Budapest, Hungary*

Abstract

Challenges and efficient solution techniques during the integration of a large number of independent ordinary differential equations (ODEs) using the massively parallel architecture of graphics processing units (GPUs) are presented. One of the main difficulties is the minimisation of the memory transactions through the PCI-E bus between the host (CPU) and the device (GPU) required frequently, for instance, during the calculation of the Lyapunov exponent, winding number or maximum response diagram. The second difficulty is the minimisation of the slow global memory transactions and memory usage by exploiting the memory hierarchy of the GPU architecture. Finally, a good GPU solver has to treat the possible asynchronous features of the ODE systems efficiently; for instance, event detection occurring at distinct time instances or handling the orders of magnitude difference in the required number of time steps of the different ODE systems. The program package MPGOS (written in C++ and CUDA C software environments) can address the aforementioned issues easily via the addition of user-defined functions that must be implemented similarly to the right-hand side of the system; via the possibility of the definition of shared parameters common to all instances of the independent ODE systems; via user-programmable parameters to store only the desired properties of the trajectories; and via an easy was to overlap GPU and CPU computations. This paper focuses on the detailed description of the implementation strategies of the program package.

Key words: ordinary differential equations, non-linear systems, GPU programming, massively parallel architecture

1. Introduction

For many physical problems, the governing equations describing the dynamics are initial value problems of first-order ordinary differential equation (ODE) systems. Even

*Corresponding author
Email address: fhegedus@hds.bme.hu (Ferenc Hegedűs)

partial differential equations can be decomposed into a large system of ODEs via suitable spatial discretization [1, 2]. Among the simplest models, one can find the classic Duffing [3–9], Morse [10, 11], Toda [12–15], Lorenz [16–19] and van der Pol [20] equations which are extensively studied for many decades from the non-linear dynamical point of view. Examples for more complex systems can be found in a variety of scientific fields: sonochemistry and bubble dynamics [21–33], engineering [34–40], social science [41–44], neuroscience [45–47] or a large number of coupled systems to study excitable media [48–50] or synchronisation and chimera states [51–53].

The aforementioned dynamical models are non-linear systems and the majority of the studies focuses on the analysis of bifurcation structures and basins of attraction of co-existing attractors [54–67]. In these cases, a large number of independent instances of the same ODE system have to be solved with different parameter combinations and/or initial conditions. The number of the instances can be in the order of hundreds of millions [68] or even billions [69]. Therefore, such problems are well-suited for the massively parallel architecture of graphics cards (GPUs) using a simple per-thread parallelisation strategy [70–74] (one instance of an ODE system is associated to a GPU thread). The program package MPGOS is designed to efficiently perform detailed numerical analysis of dynamical systems discussed above by exploiting the high processing power of GPUs.

The investigation of non-linear systems is usually carried out via many integration phases. At the end of each phase, the manipulation/investigation of the trajectories is usually necessary. For instance, computing the Lyapunov exponent [75–77] or the winding number [78–81], a new set of initial conditions needs to be prepared. In bubble dynamics, the ratio of the maximum and minimum values (compression ratio) is often a paramount property of the solutions. In systems that can exhibit impact dynamics [34–36, 58], the detection of impacts and the manipulation of the trajectories according to an impact law is mandatory.

The aforementioned tasks can easily be done using CPUs only. Even if the employed code is capable for parallel computations, a CPU operates with only a small number of threads each having access to a large amount of fast cache. Thus, the overhead to stop the integration process, manipulate the trajectories and restart the integration has a minor impact on the overall performance as all the work is done by the CPU on data that already reside in the cache. The workflow can be separated into two parts inside a loop: integration part and manipulation part. The integration part is usually done by an external library that is responsible only to integrate the system over a specified time domain. Well-tested and highly optimised codes are available for a large variety of programming languages [82]. The code for the manipulation part has to be written by the user since it depends on the specific problem and objectives. Thus, developers of integration packages usually pay little attention to provide a means for trajectory manipulation. However, there are exceptions, e.g., the program package DifferentialEquations.jl [83] where trajectory manipulation can be done via *CallbackFunctions* without stopping the integration process.

Transferring the most resource demanding integration part of the whole computation to GPUs can cause difficulties during the trajectory manipulation/investigation. Doing so on the CPU side requires quite slow memory transactions via the PCI-E bus that might have a serious impact on the performance, especially, if the complete dense output of all the trajectories has to be copied to the CPU side for further evaluation. If the program package used do not offer the possibility to do user specified work during or after

the integration process, the user needs to write its own GPU code to minimise the PCI-E transactions. In this case, the "decryption" of the data structure of the integrator is also necessary, which can be a harder task than using the program itself (in particular if object-oriented programming is involved by the package for data hiding). The trajectory manipulation (and the related event detection procedure) usually needs at different time instances of the different ODE systems. Thus, even if a program package offers a means to manipulate trajectories on the GPU side, this asynchronous behaviour needs to be handled efficiently and avoid serialisation of the code. Furthermore, GPUs operate with a massive number of threads (even hundreds of thousands); consequently, a single thread responsible to integrate a single instance of an ODE system can operate with limited amount of fast memories (registers and caches). In this sense, another difficulty is how to provide simple means to the user (without the requirement of GPU programming) to be able to exploit the memory hierarchy of the GPUs and minimise slow global memory transactions. In addition, a single monolithic kernel to perform an integration phase is also mandatory for satisfactory performance. Invoking a kernel for every time steps causing the reload of the state and other variables again from the slow global memory, and the compiler has little opportunity to optimise register usage. For detailed discussion and introduction to GPU programming, the interested reader is referred to the available handbooks [84–86].

All the challenges described above can be addressed by the program package MPGOS via the addition of a handful of user defined functions that have to be implemented similarly to the right-hand side of the ODE system, for details see Sec. 4. Also, it is a fully asynchronous solver with respect to the independent ODE systems and to the CPU. Therefore, MPGOS is not merely another software package — written in C++ and CUDA C languages — to integrate ODEs on GPU but an efficient tool to analyse dynamical/non-linear systems. It provides an easy-to-use object-oriented framework to minimise slow PCI-E and global memory operations, to distribute the workload to many GPUs and to overlap CPU and GPU computations. An efficient event detection algorithm is implemented and the user has the possibility to exploit the shared memory capacity by defining parameters common to all instances of the ODE system. Still, the package is a modular and general-purpose solver.

The main aim of the present paper is to provide a thorough description of the solution techniques employed in MPGOS to address the challenges discussed in this section. This study does not intend to give details of the implementation and program usage, it is already written in the software manual [87]. Moreover, performance measurements of different test cases and comparison with other libraries are also omitted to remain focused. It is a fairly broad subject, and our performance results are published in a separate paper [88]. The provided performance characteristics and runtime comparisons with other program packages (including solvers using CPUs) can also be found in the official website of MPGOS [89] and in its manual [87]. These numerical experiments will be continuously updated as newer versions of the investigated ODE suits are released. Thus, the interested user should check these sources regularly. Comparison with other program packages is done throughout the paper and in Sec. 8.

2. How MPGOS treats an ODE

The program solves (integrates) a large number of independent ordinary differential equations of the following form:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, t; \boldsymbol{p_{cp}}, \boldsymbol{p_{sp}}; \boldsymbol{p_{ac}}), \tag{1}$$

where $\boldsymbol{x}$ is the vector of the state variables, t is the time and $\boldsymbol{p_{cp}}$, $\boldsymbol{p_{sp}}$ and $\boldsymbol{p_{ac}}$ are the vectors of different types of parameters. The dot stands for the derivative with respect to time. The vector function $\boldsymbol{f}$ has to be the same for all systems; that is, the code solves many instances of the same system simultaneously described by Eq. (1) but with different parameter sets and/or initial conditions.

The parameters are divided into three subcategories called *control parameters* $\boldsymbol{p_{cp}}$, *shared parameters* $\boldsymbol{p_{sp}}$ and user programmable parameters called *accessories* $\boldsymbol{p_{ac}}$. The sets of control parameters are different from instance to instance. The shared parameters are common for all instances of the investigated ODE system. They are parameters of Eq. (1); however, their values do not change from instance to instance. This is the reason they are called shared parameters (shared among all the instances). Shared parameters are automatically loaded into the fast shared memory of the streaming multiprocessors of the GPU reducing the required number of slow global memory transactions, see also Sec. 6. This can be extremely important in memory bandwidth limited applications. The accessories are multi-purpose (user programmable) parameters. Strictly speaking, they are not parameters of Eq. (1) rather than storages that can be updated after every successful time step or after every successful event detection. In this regard, the number of the accessories are independent of Eq. (1), and they are absolutely under the control of the user. Accessory variables are very efficient tools to continuously calculate, monitor and store special properties of the trajectories without the requirement for storing the dense output, for details see Sec. 4.

In the present version of MPGOS, $\boldsymbol{p_{cp}}$, $\boldsymbol{p_{sp}}$ and $\boldsymbol{p_{ac}}$ are stored as a linear, one-dimensional vectors. It is the responsibility of the user to track the physical content of each vector elements. For instance, if a matrix has to be specified in $\boldsymbol{p_{sp}}$ that is shared among all the instances of the ODE system, it is up to the user how the matrix is actually stored (row-major, column-major, diagonal or only a few elements if it is sparse), and build-up the right-hand side accordingly.

3. Parallelisation strategy and distribution of the workload

The main purpose of using GPUs is to perform large parameter studies. Typically millions of instances of a system at different parameter sets or initial conditions need to be solved. In practice, these instances are not solved one at a time on a GPU. Instead, one creates smaller chunks of problems and integrate only a moderate number of instances on a single GPU. The reasons are the limited amount of global memory or the efficient usage of other GPU resources, the distribution of the workload to different GPUs or the overlap of CPU and GPU computations. Still, this moderate number can be in the order of tenth or hundreds of thousands.

A C++ object called *solver object* is responsible for performing integration phases on a chunk of problems. Upon defining a solver object, memory allocations immediately

take place on both the host (CPU) and device (GPU) side. To be able to do this, a few template parameters and the serial number of the GPU have to be specified. A solver object has a proper interface to set-up the parameters, initial conditions and other properties of an instance of the ODE system, to synchronise data between the host and the device, and to perform an integration phase; for the technical details see the manual of MPGOS [87]. In an application, multiple solver objects can be defined each assigned to a single GPU. For a single GPU, however, multiple solver objects can be associated, see Fig. 1. Since every GPU related operations of the solver object are asynchronous with respect to the CPU, the control flow immediately returns back to the CPU after initiating a kernel launch for integration. Therefore, CPU and GPU computations can easily be overlapped by assigning at least two solver objects to a GPU. Meanwhile, a solver object performs the integration; the CPU can fill-up (or modify) the other with proper data and perform the corresponding data transfer between the CPU and GPU. MPGOS provides tutorial examples for such computations.

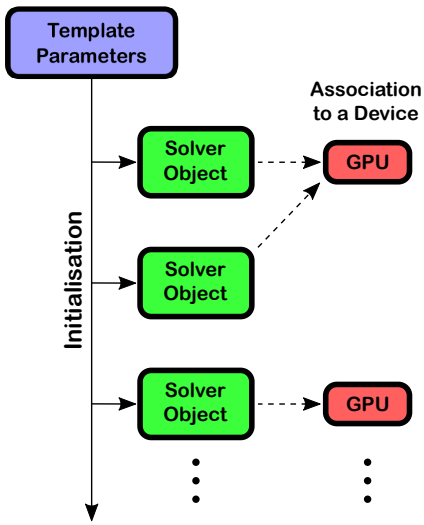

Figure 1: Initialisation of solver objects on a CPU. The collection of template parameters and the serial number of the employed GPU are necessary information for the memory allocations.

The parallelisation technique is straightforward. A single instance of the ODE system is assigned to a single GPU thread. That is, the different threads work on a different set of data (parameters, initial conditions, time domains or accessories). It is called per-thread approach [70–72]. The technique is depicted in Fig. 2. The thread hierarchy is a simple 1D organisation of block of threads. Thus, no special care has to be taken for workload distribution; only the block size must be specified as an integer multiple of the warp size (32 in the present CUDA architectures), and to launch a sufficiently large number of threads to utilise the GPU fully.

It is to be stressed that each instance of an ODE system is solved asynchronously. That is, they have their own integration time domain, internal time stepping and error control. They can even be stopped at different time instances by a user-defined termination evaluated during runtime. Therefore, the different instances can have a different

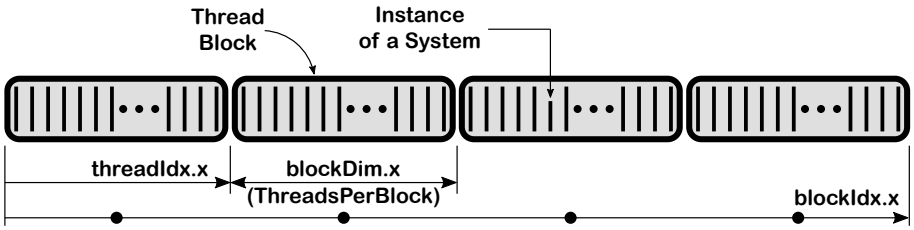

Figure 2: Parallelisation strategy inside a solver object.

number of total time steps; however, instructions during a single step are performed in a lock-step way inside a warp. Naturally, this approach causes some amount of thread divergence as the performance in a warp is determined by its slowest thread (the thread needs to carry out the largest number of time steps). If a thread finished its integration in a warp, it becomes idle. In spite of the presence of thread divergence, this technique still outperforms any alternative solutions (e.g., packing many instances into a single monolithic ODE system), see Sec. 7 for details.

4. Control flow of the solver

In order to thoroughly understand the techniques of MPGOS to handle the challenges discussed in Sec. 1, further details of the control flow of the integration procedure need to be introduced. The main difficulty is to work out a framework in which the user can easily manipulate the trajectories and extract information from the solutions continuously "on the fly" during the integration. This can be achieved via a handful of pre-declared user-defined device functions (including the right-hand side of the ODE) that are called at specific points of the solver work flow. Inside the function bodies, the user has access to every variable. In fact, any kind of control flow can be implemented according to the requirements. The detailed discussion of their implementations and their restrictions are beyond the scope of the present paper, but can be found in the manual of the package [87]. Nevertheless, the function body of the user-defined device functions can be given as simply as in case of a right-hand side function in MATLAB, and the user does not need confident knowledge of GPU programming or how an instance of the ODE system is associated to a GPU thread.

The simplified block diagram of the control flow is presented in Fig. 3 and can be summarised as follows. Via the built-in C++ object (solver object), a member function is called to perform an integration phase in a time domain spanned between t_0 and t_1. Every thread has its own instance of the ODE system, time domain t_0 and t_1, initial conditions, control parameter set, shared parameters and accessories. *All these variables are loaded into the fastest register memory of the GPU.* Thus, if the ODE system is simple enough and fits completely into the registers, the application is extremely fast as global memory operations are necessary only at the beginning and at the end of each integration phases. In case of complex or high dimensional ODE systems, register spill can occur. However, without prefetching data into the registers, these variables have to be loaded from the global memory anyway. Naturally, parameters defined as shared $\boldsymbol{p_{sp}}$ are loaded into the shared memory. Next, during an initialisation procedure, the

special user-defined device function is called labelled as `Initialisation()` in Fig. 3. Its purpose is to initialise the integration process. This can be regarded as pre-processing; for a possible application, see Sec. 5.1.

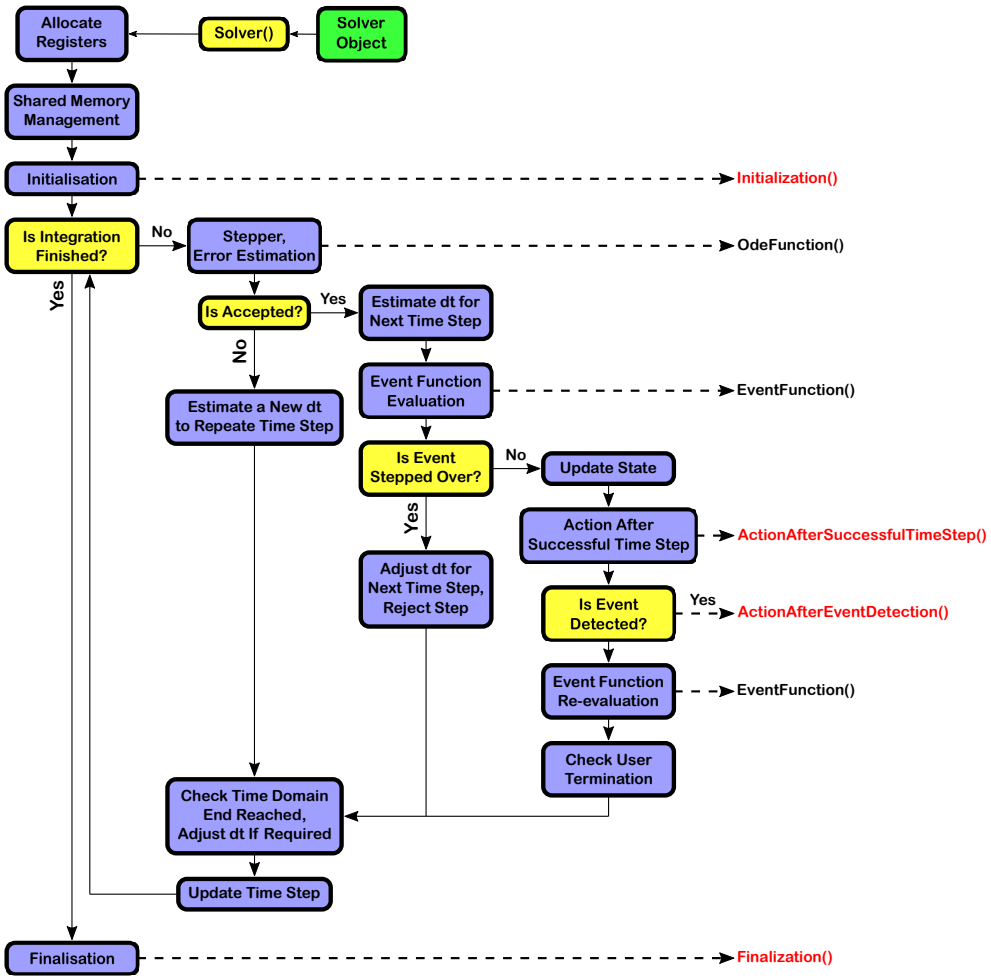

Figure 3: The control flow of the integration procedure and the list of the user-defined functions called at specific points of the integration phase. The functions marked by black colour code are the conventional functions usually used by existing libraries (e.g., MATLAB). The program package MPGOS uses the additional four (red) functions as well.

In case of a false stop condition (end of the time domain or stoped by the user), the solver performs an integration step, estimates the local error of the solution, accepts/rejects the current step and estimates a new time step. Using algorithms with fixed time stepping, the error and time step estimations are omitted. The stepper — as in case of any other libraries — successively calls the right-hand side of the ODE system named as `OdeFunction()` in Fig. 3. As of the present version of MPGOS, only the following steppers are available: the classic fourth-order Runge–Kutta scheme with

fixed time stepping; and the adaptive, fifth-order Runge–Kutta–Cash–Karp method with fourth-order embedded error estimation.

In case of an accepted step, an additional series of instructions are performed. For instance, the detection of events take place if the number of the user-specified event functions are higher then zero. An event occurs at the zero value of the corresponding event function. First, the values of the event functions are retrieved by calling the user-defined device function named as `EventFunction()` (it is called by every thread). There is a built-in root-finding algorithm based on the bisection method to find the requested special points within the prescribed tolerance. Based on the values of the event functions, a modified (reduced) time step is calculated if the current time step is too large to detect the event within the tolerance. In this case, the current step is considered as rejected. If an event is successfully detected, the user-defined device function `ActionAfterEventDetection()` is called. In this way, the trajectories can be manipulated at specific conditions defined by the event functions. This "inter-processing" feature of the program package MPGOS is important to handle, e.g., non-smooth dynamical systems efficiently, see Sec. 5.2.

If the time step is accepted and none of the events is stepped over, the current state is updated and stored (if the dense output is enabled) and the user-defined function `ActionAfterSuccessfulTimeStep()` is called (e.g., to manipulate the trajectories). This is another possibility for "inter-processing". The two types of inter-processing features (action after a detected event or action after a successful time step) serves not only to manipulate the trajectories but also to collect specific information about the solutions without storing the dense output; an application is provided in Sec. 5.3. The overuse of global memory via the dense output can also lead to the underutilisation of the GPU due to the reduced number of residing threads, see Sec. 6.2 for more details.

Inside both the user-defined device functions `ActionAfterEventDetection()` and `ActionAfterSuccessfulTimeStep()`, the user can initiate the stopping of the integration procedure by setting the value of a specific function argument to 1. This is called user-termination and provides a great flexibility for the extent of the integration as any kind of control logic can be implemented inside the above mention device functions.

Eventually, after the final successful time step, the function `Finalisation()` is called. It has a very similar purpose as the device function `Initialisation()`; namely, "post-processing", see Sec. 5.1 for a possible application.

Inspecting Fig. 3, one might conclude that a large amount of thread divergence is presented during the integration due to a large number of conditional statements in the general workflow. However, the program package is highly templatised; thus, if certain features are not necessary (e.g., event detection), the corresponding portion of the code is eliminated at compile time. In addition, the computation of a single step (always done by every thread) is much resource-intensive compared to any task in the complex control flow procedure afterwards. For example, if a single thread in a warp steps over an event, the time step adjustment procedure is invoked only for that specific thread while the rest of the threads are idle. Then, all the threads in the warp perform another time step that needs an order of magnitude large number of instructions. Because of the asynchronous treatment of the instances of the ODE systems, during the precise location of the event of a specific thread, the rest of the threads can proceed further with the integration.

It must be stressed that most of the available libraries to integrate ODE systems operate only with user-defined functions highlighted by the black colour code in Fig. 3.

In contrast, MPGOS provides much more flexibility to the user to intervene into the integration process and avoid computations on the CPU side (avoid extremely slow PCI-E memory transactions and/or reduce the global memory usage). The interventions can be done via the four additional user-defined function marked by red in Fig. 3.

There are also exceptions in the literature. The program package ODEINT [90, 91], for example, uses *observers* called after every successful time step. It is a similar feature than the function `ActionAfterSuccessfulTimeStep()` of MPGOS. However, no root-finding is implemented in ODEINT to locate events precisely. In the ODE suite DifferentialEquations.jl [83], via *CallbackFunctions*, the same features can be achieved as in the case of MPGOS; however, in its present version, the code packs the large number of instances of the ODE system into a single monolithic function that makes the event detection on the GPUs extremely inefficient, see the corresponding results in the follow-up publication [88] or the website of the program package [89]. Parenthetically, ODEINT also builds-up a monolithic ODE system.

5. Resolving the issues with the PCI-E memory transactions

This section collects a few "real-life" examples to demonstrate how the extension of the user-defined device functions can help to avoid expensive PCI-E memory transactions; thus, to prevent performance degradation. Keep in mind that these examples are not exhaustive, there is a large variety of other problems the user-defined functions can address efficiently depending on the tasks and requirements. Moreover, it must be emphasised again that this section discusses the proposed issues theoretically. For implementation considerations and runtimes, see references [87–89]. As an alternative approach to hide the large latency of PCI-E memory transactions, the role of overlapping CPU and GPU computations is also discussed.

5.1. Pre or post-processing

As a first example, let us consider the computation of the largest Lyapunov exponent of second order non-linear oscillators [3, 11, 12]. Their general form written into a first order system is

$$\dot{x}_1 = x_2, \tag{2}$$

$$\dot{x}_2 = f(x_1, x_2) + g(t), \tag{3}$$

where the function $g(t+T) = g(t)$ is periodic in time with period T. For further calculations, the extension with the linearised equations is necessary [81]:

$$\dot{x}_3 = x_4, \tag{4}$$

$$\dot{x}_4 = \frac{\partial f}{\partial x_1} x_3 + \frac{\partial f}{\partial x_2} x_4. \tag{5}$$

Observe that Eqs. (2)-(3) are independent from Eqs. (4)-(5); that is, there is a one-directional coupling. Integrating Eqs. (2)-(5) over the time domain $(0, T)$ several times, the largest Lyapunov exponent can be calculated as [75]

$$\lambda_{max} \approx \frac{1}{NT} \sum_{i=1}^{N} \ln\left(\frac{d_i}{d_{i-1}}\right), \tag{6}$$

where N is the number of the integration phases of length T and

$$d_i = \sqrt{x_{3,i}^2(T) + x_{4,i}^2(T)} \tag{7}$$

is the distance of the trajectory in the linearised subspace from the origin at $t = T$ and at integration phase i. For $i = 0$, the distance d_0 can be calculated from the initial conditions. As it is arbitrary, the initial distance is usually chosen to be unity ($d_0 = 1$). Observe that due to the periodicity of the vector field, the time domain $(0, T)$ can be fixed for every integration phase. In Eq. (6), the meaning of d_{i-1} and d_i are the initial and the final distances of a single integration phase, respectively.

In practice, the numerical behaviour of the above proposed problem is ill-conditioned. In long term, for chaotic and periodic attractors, the distance d_i tends to infinity and zero, respectively. This phenomenon introduces spurious numerical errors. Therefore, after every integration phase, the trajectory corresponding to the linear subspace of the system is normalised onto the unit circle:

$$x_{3,i}(0) = \frac{x_{3,i-1}(T)}{d_{i-1}}, \tag{8}$$

$$x_{4,i}(0) = \frac{x_{4,i-1}(T)}{d_{i-1}}. \tag{9}$$

This means that the initial distance of each integration phase is always unity, and the calculation of the largest Lyapunov exponent can be simplified to

$$\lambda_{max} \approx \frac{1}{NT} \sum_{i=1}^{N} \ln(d_i). \tag{10}$$

There are two kinds of optimisation possible via the user-defined device functions. First, the normalisation procedure by Eqs. (8)-(9) can be implemented inside the `Initialisation()` or `Finalisation()` functions, see again Fig. 3. Second, the sum of the distances in Eq. 10 can be accumulated into an accessory parameter $\boldsymbol{p_{ac}}$ inside the function `Finalisation()`. In this way, PCI-E memory transactions are necessary only two times: before the first and after the last integration phase. For chaotic attractors, in order to obtain λ_{max} within a few percent error, thousand or even tens of thousands of integration phases are usually necessary. Thus, the technique described above can have a significant positive effect on the performance.

5.2. Inter-processing

A complex control flow problem occurs when a massively parallel hardware architecture is used for mechanical systems exhibiting non-smooth impact dynamics [34–36, 58]. For simplicity, consider only autonomous systems of the form

$$\dot{x}_i = f_i(x_i; p_j), \tag{11}$$

where $i = 1 \cdots n$ and $j = 1 \cdots k$. Here n is the dimension of the state space and k is the number of parameters. Moreover, let us consider a hypersurface in the state space with dimension $n - 1$ given by an implicit form as

$$F_{im}(x_i) = 0, \tag{12}$$

on which the mechanical system exhibits an impact. That is, if a trajectory reaches this surface at a time instance t_{im}, an impact law

$$x_i(t_{im}^+) = r_i \, x_i(t_{im}^-) \tag{13}$$

has to be applied, where r_i is a factor for the individual components. In most of the cases, $r_i = 1$ but only for the velocity components: the absolute velocity vector perpendicular to the surface needs to be reversed with a certain kinetic energy loss factor described by r_i. Therefore, r_i is often called the Newtonian coefficient of restitution. From a technical point of view, an event detection algorithm is necessary to locate the intersection of the trajectory with the hypersurface defined by Eq. (12); next, the trajectory has to be "thrown away" to another point in the state space according to the impact law given by Eq. (13).

In parameter studies, during the integration of a large number of instances of Eqs. (11) over the time domain (t_0, t_1) having different parameter sets or initial conditions, the trajectories corresponding to the different instances can have different numbers of impacts (including no impact) at different time instances. In a conventional single threaded CPU approach, this phenomenon causes no major problem. One can process the instances one-by-one and handle the impacts as follows: stop the integration, apply the impact law on the trajectory separately from the integrator and restart the integration with the new initial conditions.

Using the massively parallel architecture of GPUs, where the large number of instances are integrated in parallel, the handling of the impacts is not straightforward. It must be stressed that it is very likely that the instances are *not impacting exactly at the same time*. Keep in mind again that the parallelisation strategy is very simple; namely, a single instance is associated to a single thread. What the program should do if a thread is impacting? Stop the whole integration only for a single thread? Or treat the impacting thread as idle and continue the integration with the rest of threads until all the threads are stopped at an impact or at the end of the time domain? In this case, the thread divergence can be overwhelming. In addition, where should the impact law be applied (CPU or GPU side)? Doing it on the CPU side, the required PCI-E memory transactions can further degrade the performance.

The program package MPGOS offers a quite simple solution for the aforementioned problem. The impact law can be implemented via the user-defined function `ActionAfterEventDetection()` called after a successful impact detection. Naturally, the function is called only for the thread that is impacting causing some amount of thread divergence in a warp (smallest execution unit, a collection of 32 threads). However, the evaluation of the impact law a few times in a warp requires far less computations compared to the one of the total integration process. More importantly, the computation does not have to be stopped as the impact law is applied automatically via the user-defined function `ActionAfterEventDetection()` on each thread individually. The strength of this approach is demonstrated on the model of a pressure relief valve throughout the second tutorial example of the manual [87]. The model is adopted from [36]. According to the detailed discussion in Sec. 7, in this specific example, the asynchronous integration technique of the different instances of the ODE systems are also mandatory.

5.3. Elimination of the need of dense output

Special properties of the computed trajectories of a dynamical system are often the keen interest of their analysis. For instance, to create amplification or resonance diagrams, the global maximum of a specific component of the solution is needed. An integral quantity (e.g. work done) can be approximated from the dense output by summarising the product of the values of the solution at the time instances with the corresponding time steps:

$$W \propto \sum_{i=1}^{N} x_j(t_i)\Delta t_i. \tag{14}$$

As a final example, in the field of bubble dynamics, both the minimum and the maximum values of a computed bubble radius are important to obtain the compression ratio [92, 93]. In many cases, even the elapsed time between the maximum and minimum radii is also interesting [94] (how fast the compression is taking place).

In a traditional approach, the dense output of the trajectories are required to extract the aforementioned quantities. Even using CPUs, this means a large amount of extra work as all the data have to be processed again and loaded from the system memory to compute units of the CPU core(s). In case of GPUs, the situation is much worse due to the large amount of data transfer through the PCI-E bus. Keep in mind that the complete dense output has to be copied to the CPU side for a massive number of threads (not only the initial conditions and parameters). Otherwise, the user has to write his/her own GPU code to do the job on the GPU side.

Combining the usage of the user-programmable parameters (accessories $\boldsymbol{p_{ac}}$) and the user-defined device function `ActionAfterSuccessfulTimeStep()`, the computation of the dense output and its copy to the system memory of the CPU can be completely "by-passed". Inside this function, all the important variables can be accessed by the user (state variables, parameters, accessories, the actual time instance or the actual time step). Therefore, the only task is to allocate enough user-programmable accessories for each quantity and to write a suitable control flow that updates them. For example, the control logics

$$p_{ac,1} = \max(\, p_{ac,1},\, x_i\,), \tag{15}$$

$$p_{ac,2} = \min(\, p_{ac,2},\, x_i\,), \text{and} \tag{16}$$

$$p_{ac,3} = p_{ac,3} + x_i\,\Delta t \tag{17}$$

accumulate the maximum, minimum and the approximated integral of the i^{th} component of the solution into three accessories. Here, the notations x_i and Δt means the actual state of the i^{th} component and the time step during the call of the function *ActionAfterSuccessfulTimeStep()*, respectively. The main strength of the proposed approach is that during the update of the accessories, the computed states at the current time step are still reside in the fast register memory of the GPU; thus, the computations presented via Eqs. (15)-(17) are fast. Moreover, only the values of some accessory variables need to be transferred to the CPU side as a final result instead of the whole dense output.

5.4. Overlapping CPU and GPU computations

Another approach to hide the very low latency of the PCI-E memory transactions is to use multiple solver objects and overlap CPU and GPU computations and PCI-E memory transactions. The simplest workflow is depicted in Fig. 4, where two solver

objects are defined (*Solver Object-1* and *Solver Object-2*) and associated to the same GPU. Accordingly, only half of the total number of the instances are handled by a single solver object. Inside the constructor, MPGOS assigns different CUDA streams to each solver objects. Since every GPU related task initiated by the CPU (including memory transactions via the PCI-E bus) are asynchronous with respect to the CPU, after dispatching a task to a GPU, the control immediately returns back to the CPU. Therefore, the CPU can do useful work on a solver object (one half of the instances) and initiate the synchronisation of the data between the Host and the Device meanwhile the other solver object performs the integration on the other half of the instances. In Fig. 4, the term "Working on SO-1,2" also includes the synchronisation of the data between the Host and the Device. The horizontal dashed lines represent synchronisation points between the CPU and the GPU. MPGOS offers a very easy way to define multiple solver objects and such synchronisation possibility, see also the related tutorial examples in the manual [87].

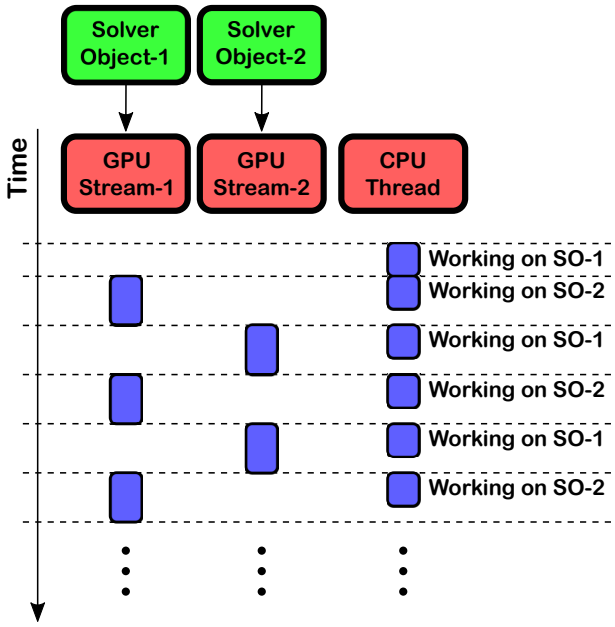

Figure 4: The simplest workflow of overlapping CPU and GPU computations and PCI-E memory transactions.

The techniques described in this section and in Sec. 5.1 can be equally efficient. The former one put more computations to the GPU side, and the latter one makes the workload between the CPU and the GPU more evenly distributed. The most efficient option depends on the problem and on the ratio of the peak processing power of the employed hardware. Although the PCI-E memory transactions can be easily overlapped with useful GPU computation with the technique introduced in Fig. 4, the storage of the dense output is still not advisable due to the possible underutilisation of the GPU, see Sec. 6.2 for more details.

6. Minimisation of the global memory transactions and usage

In this section, the global memory-related issues are discussed; that is, the high latency and possible negative effect on the utilisation of the GPU. Guidelines are provided to overcome these difficulties, minimise the negative effects and avoid performance bottlenecks due to memory bandwidth limitation.

6.1. The importance of monolithic kernels

Loading/writing into the global memory during a kernel launch is the second most expensive memory operation in GPU programming (the slowest is the PCI-E memory transaction). Although the access time of a variable in the global memory is high (the latency is approximately 600 clock cycles), this is the only memory type present in a device in large amount (few to tens of GBs). Therefore, its usage is usually necessary. In order to bridge the huge performance gap between the global memory bandwidth and the throughput of the computational units — as in case of all the high-performance computing devices — a multi-level cache hierarchy is implemented into the architecture of GPUs. In addition, GPUs has a large register file (fastest memory type). In this way, the frequently used data can be loaded into the fast on-chip cache or into the registers of the GPU and can be *kept there* to minimise the number of the global memory transactions. Without such intensive data reuse, it is impossible to "feed" the compute units of the GPU with enough data.

Consequently, any performant solver has to build-up a single monolithic kernel that performs an integration phase in a single launch. Thus, during the computation of a time step, all (or most of) the data from the previous computations are still residing in the fast registers or shared memory for reuse. Dispatching only the stepper and the error control part of the computations to the GPU, an individual kernel function have to be called for every time step. Therefore, at the beginning of every step, all the required variables have to be loaded from the global memory again, preventing any data reuse. The present version of the already mentioned ODE suits ODEINT [90, 91] and DifferentialEquations.jl [83] do *not* use monolithic kernels. This is one of the main reasons for the orders of magnitude larger runtimes compared to MPGOS, see the references [87, 89] (results may change as newer versions of the ODE suits are released).

6.2. Underutilisation of the GPU due to dense output

To hide any kind of latency (both memory request or execution dependency), GPUs follow the memory throughput-oriented design. As mentioned before, GPUs use a huge register file per streaming multiprocessor, e.g., 65536 entries of 32 bit registers (Kepler or newer architectures), which is considered significant even though a streaming multiprocessor handles a massive number of threads (e.g., a maximum of 2048 in Kepler architecture). In contrast, an equivalent number for the Ivy Bridge CPU architecture (single core) is $144 \times 8 = 1152$ entries of 32 bit registers. Thus, the strategy of the GPU to hide latency is to use a massive number of threads with a large register file. The context switch between the warps has practically no delay; consequently, the streaming multiprocessor has high flexibility to select some eligible warps for execution, while some others are waiting for data arriving from the global memory or from a previous computation. Because of the large number of registers, a single warp can usually perform many instructions before stalling due to a data request. As the decrease of the latency

is not a high priority, the throughput of the global memory bandwidth can be increased drastically.

The above-mentioned design works well only if enough threads are residing on a GPU and on a streaming multiprocessor. When a large number of time steps has to be stored during an integration phase, the amount of the global memory can be the major limiting factor of the number of the simultaneously launched GPU threads. This might result in a significant decrease of the occupancy and the ability of the warp scheduler to select an eligible warp for execution and thus hide latency. This is another reason why the storage of the dense output has to be avoided if feasible, for some specific examples see Sec. 5.3.

6.3. Exploitation of the shared memory

In contrast to CPUs, where the cache is hardware managed, in GPUs, a major part of the cache is user-programmable and it is called shared memory. It is shared as all the threads in the same block of threads can see its content. Therefore, shared memory is also used for data exchange between the threads, and quantities needed for every thread can be loaded from the global memory only once and broadcasted to all threads.

From the program package MPGOS point of view, shared memory is a perfect place to store parameters common to all instances of the ODE system studied. Such parameters are called shared parameters $\boldsymbol{p_{sp}}$ introduced in Sec. 2 and are automatically loaded into the shared memory of the GPU, see again Sec. 4. For single variables, the usage of shared memory has minor significance as a common parameter can be "hard-code" into the right-hand side implementation of the ODE system. However, there are complex situations where the hard-coding is inconvenient. Let us consider, for instance, the partial differential equation (PDE) of heat transfer in a liquid domain in spherical coordinates (only in the radial direction):

$$\frac{\partial T}{\partial t} + u(r,t)\frac{\partial T}{\partial r} = \frac{1}{\rho C_p}\frac{\partial}{\partial r}\left(\lambda r^2 \frac{\partial T}{\partial r}\right), \tag{18}$$

where $T(r,t)$ and $u(r,t)$ are the temperature and velocity fields as a function of time t and the spatial (spherical) co-ordinate r, respectively. The material properties are the density ρ, specific heat at constant pressure C_P and the thermal conductivity λ. The velocity field can be obtained by solving hydrodynamic equations. For example, in case of a radially pulsating gas bubble placed in the centre of the radial coordinate and assuming incompressibility, this velocity field can even be approximated by analytical means as a function of the radius of the pulsating bubble and its derivative [95]. The exact computation of $u(r,t)$ and the underlying governing equations, and the formulation of the boundary conditions are out of the scope of the present discussion.

The main focus here is the semi-discretisation of Eq. 18 in the spatial co-ordinate r to decompose it into a system ODEs of initial value problem, and the possible exploitation of the shared memory of the hardware (GPU). For the semi-discretisation, many techniques exist in the literature [1, 2]. In most of the cases, the discretised system can be written into a matrix form as

$$\dot{\boldsymbol{T}} = -\boldsymbol{u} \otimes (\boldsymbol{D}\boldsymbol{T}) + \frac{1}{\rho C_p}\boldsymbol{D}\left(\lambda\left(\boldsymbol{r}\otimes\boldsymbol{r}\right)\otimes(\boldsymbol{D}\boldsymbol{T})\right), \tag{19}$$

where the vectors $\boldsymbol{T}$ and $\boldsymbol{u}$ are the discretised temperature and velocity fields evaluated at the grid points of a spatial discretisation scheme:

$$\boldsymbol{r}^T = (r_1, r_2, \cdots r_n)\,. \tag{20}$$

Matrix $\boldsymbol{D}$ is the differentiation matrix whose structure and values depend on the employed semi-discretisation technique and mesh. The notation $\otimes$ stands for the element wise multiplication. The advantage of the above formalism is that a derivative with respect to r is transformed into a multiplication with $\boldsymbol{D}$; thus, the implementation of Eq. (19) is straightforward.

Now let us assume that a large parameter study has to be performed with different material properties and/or boundary conditions. If the equations are discretised with the same algorithm and grid, the differentiation matrix $\boldsymbol{D}$ becomes identical between all the instances of Eq. (19). Therefore, it is a perfect candidate to store its values in the shared memory of the GPU. It is highly unlikely that the compiler will recognise that the values of $\boldsymbol{D}$ are used by all the threads and that it will be kept in the fast on-chip cache (that part of the shared memory which is handled automatically by the hardware). Defining $\boldsymbol{D}$ as a shared parameter $\boldsymbol{p_{sp}}$, MPGOS can ensure that the matrix values will be kept in the fast shared memory. Similarly, as the quantity $\boldsymbol{r} \otimes \boldsymbol{r}$ depends only on the grid of the discretisation, its values can be precomputed on the CPU side and stored also in the shared memory of the GPU as shared parameter $\boldsymbol{p_{sp}}$. Keep in mind that in MPGOS, all the shared parameters are packed into the linearly stored vector $\boldsymbol{p_{sp}}$ that is passed to the user-defined device functions summarised in Fig. 3. Therefore, it is the responsibility of the user for the "bookkeeping" of the physical content of the vector of shared parameters.

Observe that in this way, the major part of the variables in Eq. (19) can be kept in the fast on-chip memory of the hardware. This makes the computations fast and efficient. Observe also that hardcoding the values of $\boldsymbol{D}$ and $\boldsymbol{r} \otimes \boldsymbol{r}$ is a cumbersome task and far from flexible. Even in older GPU architectures, the available amount of shared memory is at least 48 KB. That is, altogether 6144 number of double-precision floating point numbers can be stored. Therefore, in the limit case (a single block resides in a streaming multiprocessor), the aforementioned technique works well up to a resolution (grid size) of $n = 77$. For a detailed example, the interested reader is referred to one of our previous publications [96], where millions of instances of a coupled ODE-PDE system is solved in a large dimensional parameter space in order to calculate the spherical stability of a single gas bubble placed in a highly viscous liquid.

7. The importance of the asynchronous solution technique

According to the discussion in Sec. 3, the program package MPGOS employs a simple per-thread parallelisation strategy: one GPU thread is responsible to integrate one instance of the ODE system. It has to be stressed again that every instance has its own time domain and time stepping; that is, they are not shared between the threads and every instance can progress with its own "rhythm" asynchronously. This means that MPGOS handles a multitude of lightweight instances of an ODE systems, which are completely "separated" (besides the shared parameters). Therefore, in case of adaptive algorithms, this can cause a certain level of thread divergence as the different instances might need

different number of time steps to complete. The execution time of the smallest execution unit — a warp that is a block of 32 number of threads — is determined by its slowest thread. It is hard to avoid such an overhead [97] and its amount is problem dependent.

Another commonly employed approach is when every instance of the ODE shares the same time domain and time stepping. It allows one to implement the multiple instance into a single monolithic ODE system. Although the system to be solved is monolithic, it is composed by the collection of replicated and independent sets of equations (sub-systems). In this way, the right-hand side can be written in a vectorised from and can be evaluated on a GPU [83, 90, 98]. The advantage of this technique is that the high processing power of GPUs can be exploited without the necessity to "reinvent the wheel" and reimplement existing algorithms/libraries.

The common time stepping, however, might have a severe consequence. Even if only one sub-system has to slow down (decrease the time step) due to a rapidly varying state variable, the complete monolithic ODE system is slowed down. Since the collection of the independent sets of equations has different but usually close parameter/initial condition combinations, it is very likely that the sub-systems in the monolithic ODE has similar trajectories. This implies slow down not only once but as many times as the number of combined sub-systems. In a worst-case scenario, the runtime can be even higher than the one of solving the independent sub-systems serially using a CPU. This phenomenon can be clearly seen in the runtimes of the program packages ODEINT [90, 91] and DifferentialEquations.jl [83] for certain problems [87–89].

Another issue with the monolithic ODE system and the common time stepping is the impossibility to stop the integration of the distinct instances at different values of the time. That is, all the instances have to advance in time and stop the integration synchronously. This loss of flexibility makes the efficient treatment of autonomous ODE systems almost impossible. For instance, during the analysis of periodic orbits, a suitable Poincaré section has to be defined. Since the system is autonomous, the time domain of a single integration phase (integration from Poincaré section to Poincaré section) can be different from instance to instance of the ODE. In case of a single monolithic system, the proper handling of such situation is not trivial and needs a complex control logic (e.g., storing the point of the Poincaré section, and discard the rest of the results if further integration is needed due to another instance).

8. Summary

The conventional way of thinking of an initial value problem is a task where a system of ordinary differential equations has to be integrated between the time instances t_0 and t_1, store the dense output (if required) and detect special states of the trajectory called events (again if necessary). In many integrator packages, a little or no opportunity is usually offered to ease the post-processing of the trajectories, their manipulation during the integration phase, not to mention overlapping CPU and GPU computations or explicit exploitation of shared memory.

The lack of the aforementioned features of the integration packages become severe if the massively parallel architecture of GPUs is employed. For the post-processing of the results between a large number of integration phases, expensive PCI-E memory transactions are necessary; unless, the overlapping of CPU and GPU computations are supported. This can have a significant negative impact of the overall performance of the

application (even if the GPU integrator itself is highly efficient). Otherwise, the user has to write its own kernel function to do the job on the GPU side or reimplement the numerical schemes to fit the problem to their needs [70–74, 99–105]. The program package MPGOS offers a set of functionalities described throughout this paper to minimise the slow PCI-E memory operations and avoid the related performance degradation. In addition, it also offers the possibility to define shared parameters to minimise the second slowest global memory transactions. The package builds up a monolithic kernel to ensure data reuse and solves the instances of the ODE system asynchronously. Still, it is a modular and general-purpose program package.

Throughout the paper, comparisons of the functionality of MPGOS were continuously done with ODEINT and DifferentialEquations.jl since we have direct experience with these ODE suits. But, there are other few general-purpose, GPU capable packages in the "market". For example, ginSODA is suitable for solving stiff ODE systems; however, according to its publication [106], it lacks the aforementioned special features of MPGOS (e.g., event detection, pre or post-processing). The ODE suit SUNDIALS developed at the Lawrence Livermore National Laboratory (LLNL) [107, 108] also has support for dispatching workload to GPUs. However, the provided examples for parallel execution use spatially discretised PDEs. This suggests that the only way for parameter studies is to use a monolithic ODE function, which has severe drawbacks, see again Sec. 7.

Acknowledgement

This paper was supported by the Alexander von Humboldt Foundation, by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences, and by the Higher Education Excellence Program of the Ministry of Human Capacities in the frame of Water science & Disaster Prevention research area of Budapest University of Technology and Economics (BME FIKP-VÍZ).

The author also wishes to acknowledge the kind help and valuable remarks of Werner Lauterborn, Ulrich Parlitz and Robert Mettin.

References

[1] J. P. Boyd, Chebyshev and Fourier Spectral Methods, Dover Publications, New York, Mineola, 2001.

[2] C. Canuto, M. Y. Hussaini, A. Quarteroni, T. A. Zang, Spectral Methods, Springer-Verlag, Berlin, Heidelberg, 2006.

[3] C. Bonatto, J. A. C. Gallas, Y. Ueda, Chaotic phase similarities and recurrences in a damped-driven Duffing oscillator, Phys. Rev. E 77 (2) (2008) 026217.

[4] V. Englisch, W. Lauterborn, Regular window structure of a double-well Duffing oscillator, Phys. Rev. A 44 (2) (1991) 916–924.

[5] R. Gilmore, J. W. L. McCallum, Structure in the bifurcation diagram of the Duffing oscillator, Phys. Rev. E 51 (1995) 935–956.

[6] Y. H. Kao, J. C. Huang, Y. S. Gou, Persistent properties of crises in a Duffing oscillator, Phys. Rev. A 35 (12) (1987) 5228–5232.

[7] J. Kozłowski, U. Parlitz, W. Lauterborn, Bifurcation analysis of two coupled periodically driven Duffing oscillators, Phys. Rev. E 51 (3) (1995) 1861–1867.

[8] U. Parlitz, W. Lauterborn, Superstructure in the bifurcation set of the Duffing equation $\ddot{x} + d\dot{x} + x + x^3 = fcos(\omega t)$, Phys. Lett. A 107 (8) (1985) 351–355.

[9] C. S. Wang, Y. H. Kao, J. C. Huang, Y. S. Gou, Potential dependence of the bifurcation structure in generalized Duffing oscillators, Phys. Rev. A 45 (6) (1992) 3471–3485.

[10] W. Knop, W. Lauterborn, Bifurcation structure of the classical Morse oscillator, J. Chem. Phys. 93 (6) (1990) 3950–3957.
[11] C. Scheffczyk, U. Parlitz, T. Kurz, W. Knop, W. Lauterborn, Comparison of bifurcation structures of driven dissipative nonlinear oscillators, Phys. Rev. A 43 (12) (1991) 6495–6502.
[12] T. Kurz, W. Lauterborn, Bifurcation structure of the Toda oscillator, Phys. Rev. A 37 (1988) 1029–1031.
[13] B. K. Goswami, The interaction between period 1 and period 2 branches and the recurrence of the bifurcation structures in the periodically forced laser rate equations, Opt. Commun. 122 (4) (1996) 189–199.
[14] B. K. Goswami, Self-similarity in the bifurcation structure involving period tripling, and a suggested generalization to period n-tupling, Phys. Lett. A 245 (1-2) (1998) 97–109.
[15] B. K. Goswami, Flip-flop between soft-spring and hard-spring bistabilities in the approximated Toda oscillator analysis, Pramana 77 (5) (2011) 987–1005.
[16] R. Meucci, F. Salvadori, K. A. Naimee, S. Brugioni, B. K. Goswami, S. Boccaletti, F. T. Arecchi, Attractor selection in a modulated laser and in the Lorenz circuit, Phil. Trans. R. Soc. A 366 (2008) 475–486.
[17] B. K. Goswami, Control of multistate hopping intermittency, Phys. Rev. E 78 (2008) 066208.
[18] B. K. Goswami, Controlled destruction of chaos in the multistable regime, Phys. Rev. E 76 (2007) 016219.
[19] E. N. Lorenz, Deterministic nonperiodic flow, J. Atmos. Sci. 20 (2) (1963) 130–141.
[20] R. Mettin, U. Parlitz, W. Lauterborn, Bifurcation structure of the driven van der Pol oscillator, Int. J. Bifurcat. Chaos 03 (06) (1993) 1529–1555.
[21] Y. Zhang, Y. Zhang, Chaotic oscillations of gas bubbles under dual-frequency acoustic excitation, Ultrason. Sonochem. 40 (2018) 151–157.
[22] Y. Zhang, Y. Zhang, S. Li, Combination and simultaneous resonances of gas bubbles oscillating in liquids under dual-frequency acoustic excitation, Ultrason. Sonochem. 35 (2017) 431–439.
[23] Y. Zhang, Y. Zhang, S. Li, The secondary Bjerknes force between two gas bubbles under dual-frequency acoustic excitation, Ultrason. Sonochem. 29 (2016) 129–145.
[24] Y. Zhang, D. Billson, S. Li, Influences of pressure amplitudes and frequencies of dual-frequency acoustic excitation on the mass transfer across interfaces of gas bubbles, Int. J. Heat Mass Transf. 66 (2015) 16–20.
[25] Y. Zhang, X. Du, H. Xian, Y. Wu, Instability of interfaces of gas bubbles in liquids under acoustic excitation with dual frequency, Ultrason. Sonochem. 23 (2015) 16–20.
[26] K. Yasui, T. Tuziuti, J. Lee, T. Kozuka, A. Towata, Y. Iida, The range of ambient radius for an active bubble in sonoluminescence and sonochemical reactions, J. Chem. Phys. 128 (18) (2008) 184705.
[27] K. Yasui, T. Tuziuti, T. Kozuka, A. Towata, Y. Iida, Relationship between the bubble temperature and main oxidant created inside an air bubble under ultrasound, J. Chem. Phys. 127 (15) (2007) 154502.
[28] K. Yasui, T. Tuziuti, Y. Iida, Optimum bubble temperature for the sonochemical production of oxidants, Ultrasonics 42 (1) (2004) 579–584.
[29] H. Haghi, A. J. Sojahrood, M. C. Kolios, On amplification of radial oscillations of microbubbles due to bubble-bubble interaction in polydisperse microbubble clusters under ultrasound excitation, J. Acoust. Soc. Am. 143 (3) (2018) 1862–1862.
[30] H. Haghi, A. J. Sojahrood, A. C. De Leon, A. Agata Exner, M. C. Kolios, Experimental and numerical investigation of backscattered signal strength from different concentrations of nanobubble and microbubble clusters, J. Acoust. Soc. Am. 144 (3) (2018) 1888–1888.
[31] H. Haghi, A. J. Sojahrood, R. Karshafian, M. C. Kolios, Numerical investigation of the subharmonic response of a cloud of interacting microbubbles, J. Acoust. Soc. Am. 141 (5) (2017) 3493–3493.
[32] A. J. Sojahrood, D. Wegierak, H. Haghi, R. Karshfian, M. C. Kolios, A simple method to analyze the super-harmonic and ultra-harmonic behavior of the acoustically excited bubble oscillator, Ultrason. Sonochem. 54 (2019) 99–109.
[33] A. J. Sojahrood, Q. Li, H. Haghi, R. Karshafian, T. M. Porter, M. C. Kolios, Towards the accurate characterization of the shell parameters of microbubbles based on attenuation and sound speed measurements, J. Acoust. Soc. Am. 141 (5) (2017) 3493–3493.
[34] C. J. Hős, A. R. Champneys, K. Paul, M. McNeely, Dynamic behaviour of direct spring loaded pressure relief valves in gas service: II reduced order modelling, J. Loss Prevent. Proc. 36 (2015) 1–12.

[35] C. J. Hős, A. R. Champneys, K. Paul, M. McNeely, Dynamic behavior of direct spring loaded pressure relief valves in gas service: Model development, measurements and instability mechanisms, J. Loss Prevent. Proc. 31 (2014) 70–81.
[36] C. Hős, A. R. Champneys, Grazing bifurcations and chatter in a pressure relief valve model, Physica D 241 (22) (2012) 2068–2076.
[37] C. Hős, A. R. Champneys, L. Kullmann, Bifurcation analysis of surge and rotating stall in the moore–greitzer compression system, IMA J. Appl. Math. 68 (2) (2003) 205–228.
[38] Y. Altintas, G. Stepan, D. Merdol, Z. Dombovari, Chatter stability of milling in frequency and discrete time domain, CIRP J. Manuf. Sci. Tech. 1 (1) (2008) 35–44.
[39] T. G. Molnar, Z. Dombovari, T. Insperger, G. Stepan, Bifurcation analysis of nonlinear time-periodic time-delay systems via semidiscretization, Int. J. Numer. Meth. Eng. 115 (1) (2018) 57–74.
[40] A. K. Kiss, S. S. Avedisov, D. Bachrathy, G. Orosz, On the global dynamics of connected vehicle systems, Nonlinear Dyn. 96 (3) (2019) 1865–1877.
[41] S. Rinaldi, P. Landi, F. D. Rossa, Temporary bluffing can be rewarding in social systems: The case of romantic relationships, J. Math. Sociol. 39 (3) (2015) 203–220.
[42] J.-M. Rey, A mathematical model of sentimental dynamics accounting for marital dissolution, PLoS One 5 (3) (2010) 1–8.
[43] P. Landi, F. Dercole, The social diversification of fashion, J. Math. Sociol. 40 (3) (2016) 185–205.
[44] E. Ferrer, J. L. Helm, Dynamical systems modeling of physiological coregulation in dyadic interactions, Int. J. Psychophysiol. 88 (3) (2013) 296–308.
[45] A. V. Andreev, A. N. Pisarchik, Mathematical simulation of coherent resonance phenomenon in a network of Hodgkin-Huxley biological neurons, in: D. E. Postnov (Ed.), Saratov Fall Meeting 2018: Computations and Data Analysis: from Nanoscale Tools to Brain Functions, Vol. 11067, International Society for Optics and Photonics, SPIE, 2019, pp. 36–41.
[46] A. V. Andreev, N. S. Frolov, A. N. Pisarchik, A. E. Hramov, Chimera state in complex networks of bistable Hodgkin-Huxley neurons, Phys. Rev. E 100 (2) (2019) 022224.
[47] V. A. Maksimenko, V. V. Makarov, B. K. Bera, D. Ghosh, S. K. Dana, M. V. Goremyko, N. S. Frolov, A. A. Koronovskii, A. E. Hramov, Excitation and suppression of chimera states by multiplexing, Phys. Rev. E 94 (5) (2016) 052205.
[48] R. S. Zimmermann, U. Parlitz, Observing spatio-temporal dynamics of excitable media using reservoir computing, Chaos 28 (4) (2018) 043118.
[49] A. Schlemmer, S. Berg, T. Lilienkamp, S. Luther, U. Parlitz, Spatiotemporal permutation entropy as a measure for complexity of cardiac arrhythmia, Front. Phys. 6 (2018) 39.
[50] P. Bittihn, S. Berg, U. Parlitz, S. Luther, Emergent dynamics of spatio-temporal chaos in a heterogeneous excitable medium, Chaos 27 (9) (2017) 093931.
[51] V. K. Chandrasekar, R. Gopal, A. Venkatesan, M. Lakshmanan, Mechanism for intensity-induced chimera states in globally coupled oscillators, Phys. Rev. E 90 (6) (2014) 062913.
[52] T. A. Glaze, S. Lewis, S. Bahar, Chimera states in a Hodgkin-Huxley model of thermally sensitive neurons, Chaos 26 (8) (2016) 083119.
[53] A. Yeldesbay, A. Pikovsky, M. Rosenblum, Chimeralike states in an ensemble of globally coupled oscillators, Phys. Rev. Lett. 112 (14) (2014) 144103.
[54] D. R. da Costa, M. Hansen, G. Guarise, R. O. Medrano-T, E. D. Leonel, The role of extreme orbits in the global organization of periodic regions in parameter space for one dimensional maps, Phys. Lett. A 380 (18) (2016) 1610–1614.
[55] S. L. T. de Souza, A. A. Lima, I. L. Caldas, R. O. Medrano-T, Z. O. Guimarães-Filho, Self-similarities of periodic structures for a discrete model of a two-gene system, Phys. Lett. A 376 (15) (2012) 1290–1294.
[56] E. S. Medeiros, R. O. Medrano-T, I. L. Caldas, S. L. T. de Souza, Torsion-adding and asymptotic winding number for periodic window sequences, Phys. Lett. A 377 (8) (2013) 628–631.
[57] E. S. Medeiros, S. L. T. de Souza, R. O. Medrano-T, I. L. Caldas, Replicate periodic windows in the parameter space of driven oscillators, Chaos Solitons Fract. 44 (11) (2011) 982–989.
[58] E. S. Medeiros, S. L. T. de Souza, R. O. Medrano-T, I. L. Caldas, Periodic window arising in the parameter space of an impact oscillator, Phys. Lett. A 374 (26) (2010) 2628–2635.
[59] R. O. Medrano-T, R. Rocha, The negative side of Chua's circuit parameter space: stability analysis, period-adding, basin of attraction metamorphoses, and experimental investigation, Int. J. Bifurcat. Chaos 24 (09) (2014) 1430025.
[60] J. A. de Oliveira, L. T. Montero, D. R. da Costa, J. A. Méndez-Bermúdez, R. O. Medrano-T, E. D. Leonel, An investigation of the parameter space for a family of dissipative mappings, Chaos

29 (5) (2019) 053114.
[61] A. Celestino, C. Manchein, H. A. Albuquerque, M. W. Beims, Stable structures in parameter space and optimal ratchet transport, Commun. Nonlinear Sci. Numer. Simul. 19 (1) (2014) 139–149.
[62] A. C. C. Horstmann, H. A. Albuquerque, C. Manchein, The effect of temperature on generic stable periodic structures in the parameter space of dissipative relativistic standard map, Eur. Phys. J. B 90 (5) (2017) 96.
[63] N. S. Nicolau, T. M. Oliveira, A. Hoff, H. A. Albuquerque, C. Manchein, Tracking multistability in the parameter space of a Chua's circuit model, Eur. Phys. J. B 92 (5) (2019) 106.
[64] D. W. C. Marcondes, G. F. Comassetto, B. G. Pedro, J. C. C. Vieira, A. Hoff, F. Prebianca, C. Manchein, H. A. Albuquerque, Extensive numerical study and circuitry implementation of the watt governor model, Int. J. Bifurcat. Chaos 27 (11) (2017) 1750175.
[65] A. Celestino, C. Manchein, H. A. Albuquerque, M. W. Beims, Ratchet transport and periodic structures in parameter space, Phys. Rev. Lett. 106 (23) (2011) 234101.
[66] C. A. C. Jousseph, S. A. Abdulack, C. Manchein, M. W. Beims, Hierarchical collapse of regular islands via dissipation, J. Phys. A 51 (10) (2018) 105101.
[67] C. A. Jousseph, T. S. Kruger, C. Manchein, S. R. Lopes, M. W. Beims, Weak dissipative effects on trajectories from the edge of basins of attraction, Physica A 456 (2016) 68–74.
[68] F. Hegedűs, W. Lauterborn, U. Parlitz, R. Mettin, Non-feedback technique to directly control multistability in nonlinear oscillators by dual-frequency driving, Nonlinear Dyn. 94 (1) (2018) 273–293.
[69] F. Hegedűs, K. Klapcsik, W. Lauterborn, U. Parlitz, R. Mettin, GPU accelerated study of a dual-frequency driven single bubble in a 6-dimensional parameter space: The active cavitation threshold, Ultrason. Sonochem. 67 (2020) 105067.
[70] C. P. Stone, A. T. Alferman, K. E. Niemeyer, Accelerating finite-rate chemical kinetics with coprocessors: Comparing vectorization methods on GPUs, MICs, and CPUs, Comput. Phys. Commun. 226 (2018) 18–29.
[71] N. J. Curtis, K. E. Niemeyer, C. J. Sung, An investigation of GPU-based stiff chemical kinetics integration methods, Combust. Flame 179 (2017) 312–324.
[72] K. E. Niemeyer, C.-J. Sung, Recent progress and challenges in exploiting graphics processors in computational fluid dynamics, J. Supercomput. 67 (2) (2014) 528–564.
[73] C. Stone, R. Davis, Techniques for solving stiff chemical kinetics on GPUs, in: 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, Grapevine (Dallas/Ft. Worth Region), Texas, 2013, p. 17.
[74] F. Sewerin, S. Rigopoulos, A methodology for the integration of stiff chemical kinetics on GPUs, Combust. Flame 162 (4) (2015) 1375–1394.
[75] K. Geist, U. Parlitz, W. Lauterborn, Comparison of different methods for computing Lyapunov exponents, Prog. Theor. Phys. 83 (5) (1990) 875–893.
[76] U. Parlitz, Identification of true and spurious Lyapunov exponents from time series, Int. J. Bifurcat. Chaos 02 (01) (1992) 155–165.
[77] C. Skokos, The Lyapunov Characteristic Exponents and Their Computation, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 63–135.
[78] V. Englisch, U. Parlitz, W. Lauterborn, Comparison of winding-number sequences for symmetric and asymmetric oscillatory systems, Phys. Rev. E 92 (2) (2015) 022907.
[79] V. Englisch, W. Lauterborn, The winding-number limit of period-doubling cascades derived as Farey-fraction, Int. J. Bifurcat. Chaos 4 (4) (1994) 999–1002.
[80] U. Parlitz, W. Lauterborn, Period-doubling cascades and devil's staircases of the driven van der Pol oscillator, Phys. Rev. A 36 (3) (1987) 1428–1434.
[81] U. Parlitz, W. Lauterborn, Resonances and torsion numbers of driven dissipative nonlinear oscillators, Z. Naturforsch. A 41 (4) (1986) 605–614.
[82] C. Rackauckas, A comparison between differential equation solver suites in MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran, The Winnower 6 (2018) e153459.98975.
[83] C. Rackauckas, Q. Nie, DifferentialEquations.jl - A performant and feature-rich ecosystem for solving differential equations in Julia, J. Open Res. Softw. 5 (1) (2017) 15.
[84] T. Soyata, GPU Parallel Program Development Using CUDA, CRC Press, Boca Raton, Florida, 2018.
[85] D. B. Kirk, W.-m. W. Hwu, Programming Massively Parallel Processors, Elsevier Inc., Cambridge, United States, 2017.
[86] J. Cheng, M. Grossman, T. McKercher, Professional CUDA C Programming, John Wiley & Sons, Inc., Indianapolis, Indiana, 2014.

[87] F. Hegedűs, MPGOS: GPU accelerated integrator for large number of independent ordinary differential equation systems, Budapest Univesity of Technology and Economics, Budapest, Hungary (2019).
[88] D. Nagy, L. Plavecz, F. Hegedűs, Solving large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs: performance comparisons of MPGOS, ODEINT and DifferentialEquations.jl, submitted.
[89] www.gpuode.com.
[90] K. Ahnert, D. Demidov, M. Mulansky, Solving Ordinary Differential Equations on GPUs, Springer International Publishing, 2014, pp. 125–157.
[91] http://headmyshoulder.github.io/odeint-v2/.
[92] R. Mettin, C. Cairós, A. Troia, Sonochemistry and bubble dynamics, Ultrason. Sonochem. 25 (2015) 24–30.
[93] W. Lauterborn, T. Kurz, Physics of bubble oscillations, Rep. Prog. Phys. 73 (10) (2010) 106501.
[94] P. A. Tatake, A. B. Pandit, Modelling and experimental investigation into cavity dynamics and cavitational yield: influence of dual frequency ultrasound sources, Chem. Eng. Sci. 57 (22) (2002) 4987–4995.
[95] F. Hegedűs, K. Klapcsik, The effect of high viscosity on the collapse-like chaotic and regular periodic oscillations of a harmonically excited gas bubble, Ultrason. Sonochem. 27 (2015) 153–164.
[96] K. Klapcsik, F. Hegedűs, Study of non-spherical bubble oscillations under acoustic irradiation in viscous liquid, Ultrason. Sonochem. 54 (2019) 256–273.
[97] L. Murray, GPU acceleration of Runge-Kutta integrators, IEEE T. Parall. Distr. 23 (1) (2012) 94–101.
[98] D. Demidov, K. Ahnert, K. Rupp, P. Gottschling, Programming CUDA and OpenCL: A case study using modern C++ libraries, SIAM J. Sci. Comput. 35 (5) (2013) C453–C472.
[99] Y. Shi, W. H. Green, H.-W. Wong, O. O. Oluwole, Accelerating multi-dimensional combustion simulations using GPU and hybrid explicit/implicit ODE integration, Combust. Flame 159 (7) (2012) 2388–2397.
[100] H. P. Le, J.-L. Cambier, L. K. Cole, GPU-based flow simulation with detailed chemical kinetics, Comput. Phys. Commun. 184 (3) (2013) 596–606.
[101] A. Al-Omari, J. Arnold, T. Taha, H. B. Schüttler, Solving large nonlinear systems of first-order ordinary differential equations with hierarchical structure using multi-GPGPUs and an adaptive Runge Kutta ODE solver, IEEE Access 1 (2013) 770–777.
[102] B. Brock, A. Belt, J. J. Billings, M. Guidry, Explicit integration with GPU acceleration for large kinetic networks, J. Comput. Phys. 302 (2015) 591–602.
[103] F. I. Fazanaro, D. C. Soriano, R. Suyama, M. K. Madrid, J. R. Oliveira, I. B. Muñoz, R. Attux, Numerical characterization of nonlinear dynamical systems using parallel computing: The role of GPUs approach, Commun. Nonlinear Sci. Numer. Simul. 37 (2016) 143–162.
[104] A. Imren, D. C. Haworth, On the merits of extrapolation-based stiff ODE solvers for combustion CFD, Combust. Flame 174 (2016) 1–15.
[105] T. Kovac, T. Haber, F. V. Reeth, N. Hens, Heterogeneous computing for epidemiological model fitting and simulation, BMC Bioinform. 19 (1) (2018) 101.
[106] S. M. Nobile, P. Cazzaniga, D. Besozzi, G. Mauri, ginSODA: massive parallel integration of stiff ODE systems on GPUs, J. Supercomput. 75 (12) (2018) 1–12.
[107] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, C. S. Woodward, SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers, ACM Trans. Math. Softw. 31 (3) (2005) 363–396.
[108] https://computing.llnl.gov/projects/sundials.