Contents lists available at ScienceDirect

Communications in Nonlinear Science and Numerical Simulation

journal homepage: www.elsevier.com/locate/cnsns

The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs

Dániel Nagy, Lambert Plavecz, Ferenc Hegedűs [*]

Department of Hydrodynamic Systems, Faculty of Mechanical Engineering, Budapest University of Technology and Economics, Budapest, Hungary

A R T I C L E   I N F O

Article history:
Received 10 December 2020
Received in revised form 1 March 2022
Accepted 9 April 2022
Available online 19 April 2022

Keywords:
Ordinary differential equations
Non-stiff problems
GPU programming
CPU programming

A B S T R A C T

This paper discusses the main performance barriers for solving a large number of independent ordinary differential equation systems on processors (CPU) and graphics cards (GPU). With a naïve approach, for instance, the utilisation of a CPU can be as low as 4% of its theoretical peak processing power. The main barriers identified by the detailed analysing of the hardware architectures and profiling using hardware performance monitoring units are as follows. First, exploitation of the SIMD capabilities of the CPU via vector registers. The solution is to implement/enforce explicit vectorisation. Second, hiding instruction latencies on both CPUs and GPUs that can be achieved with increasing (instruction-level) parallelism. Third, the efficient handling of large timescale differences or event handling using the massively parallel architecture of GPUs. A viable option to overcome this difficulty is asynchronous time stepping. The above optimisation techniques and their implementation possibilities are discussed and tested on three program packages: MPGOS written in C++ and specialised only for GPUs; ODEINT implemented in C++, which supports execution on both CPUs and GPUs; finally, DifferentialEquations.jl written in Julia that also supports execution on both CPUs and GPUs. The tested systems (Lorenz equation, Keller–Miksis equation and a pressure relief valve model) are non-stiff and have low dimension. Thus, the performance of the codes are not limited by memory bandwidth, and Runge–Kutta type solvers are efficient and suitable choices. The employed hardware are an Intel Core i7-4820K CPU with 30.4 GFLOPS peak double-precision performance per cores and an Nvidia GeForce Titan Black GPU that has a total of 1707 GFLOPS peak double-precision performance.

1. Introduction

Low dimensional ordinary differential equation (ODE) systems are still widely used in physics and non-linear dynamical analysis. Among the simplest models (second or third-order systems), one can still find many publications employing the classic Duffing [1–6], Morse [7–10], Toda [11–14], Brusselator [15,16], van der Pol [17–19] and Lorenz [20–23] equations. From time to time, the results obtained in these simple test cases provide new insights into the complex dynamics of chaotic behaviour or into the bifurcation structure in multi-dimensional parameter spaces. There is an exhaustive literature of low dimensional systems in many other branches of physics; for instance, single bubble models in acoustic cavitation and sonochemistry [24–31], multi-species models in population dynamics [32–34] or the problem of pressure relief valves [35–38], machine tools [39,40] or wheel shimmy [41,42] in engineering. Naturally, it is hard to give an exact

[*] Corresponding author.
E-mail addresses: n.nagyd@gmail.com (D. Nagy), plaveczlambert@gmail.com (L. Plavecz), fhegedus@hds.bme.hu (F. Hegedűs).

https://doi.org/10.1016/j.cnsns.2022.106521

definition for low dimension. In this paper, we follow a computational approach: a system is considered low dimensional if the application (solver) is not limited by memory bandwidth. Roughly speaking, this means that the entire problem (including the variables required by the solver) fits into the L1 cache of the CPU, or a large portion of the entire problem fits into the registers of the GPU. For the details of the different architectures, see Appendices A.1.2 and A.2.1.

Although the precise description of the majority of the physical problems usually requires complex modelling (e.g., involving partial differential equations), the application of reduced-order models still play a significant role in many scientific fields. The obvious reason is that with increasing model complexity, the required computational capacity increases as well. Therefore, there is always a compromise between the size of the investigated system and the number of the parameter combinations that can be examined with a given computational capacity (within a reasonable time). Good examples are the computations of the escape rates in transient chaos [43] that need long transient simulations with millions of initial conditions; or parameter studies in high-dimensional parameter spaces, see a recent publication of a scan of nearly 2 billion parameter combinations of an acoustic bubble [44]. In addition, in many cases, a low-order system is already an acceptable approximation of the physical process: the Keller–Miksis equation in bubble dynamics agrees very well with measurements [24], or moderately complex pressure relief valve models are able to predict the stability limit of the device [35]. *The present paper focuses on the solution techniques of large parameter studies of such low-order systems.*

For CPUs, suits for solving ordinary differential equations have existed for decades. The codes of Harier [45–47] and ODEPACK [48,49] written in Fortran are the ones having the longest history, and they are well-known packages in the community. The ODE suit SUNDIALS developed at the Lawrence Livermore National Laboratory (LLNL) also has a long history in the research of ODE solvers [50,51]. Besides the original Fortran 77 codes, the newest developments are done in ANSI C and C++. Another promising candidate for the solution of ODEs is ODEINT [52,53] written in C++ to fully exploit its object-oriented capabilities. As final examples, all the major high-level programming languages, like Python (e.g., SciPy) [54], Matlab [55] or Julia (DifferentialEquations.jl) [56] provide a tool for ODEs. The interested reader is referred to one of the most exhaustive overviews of the available ODE suits in the market [57]. It is expected that low-level, compiler-based solvers have excellent performance, while high-level languages are good for making clear and well-structured code, and excellent in fast and easy development. Some solutions (e.g., Julia) try to combine both and compile the code before execution (just-in-time compilation) providing a means for both fast and efficient code development. One of the many important questions is *how effectively these program packages exploit the modern architectural features of CPUs for large parameter studies of low-dimensional ODEs; for instance, the SIMD capabilities via the vector registers.*

In contrast to CPUs, the general-purpose utilisation of GPUs for scientific computing is relatively new. It also needs a different way of thinking due to its massively parallel architecture. In case of large parameter studies where each instance of the ODE system is independent, the parallelisation strategy is simple; namely, a single GPU thread solves a single instance of an ODE system having different initial conditions or parameter sets. Even though the parallelisation strategy is trivial, many problems can still arise degrading the performance significantly. For example, GPU is a co-processor, and before an integration phase, data transfer via the extremely slow PCI-E bus is mandatory. If trajectory manipulations can be done only by the CPU between successive integration phases (e.g., the regular normalisation during the computation of the Lyapunov spectra), the performance can decrease. Unless, the program package provides the means to overlap CPU and GPU computations by asynchronous dispatch of GPU tasks (integration). Another example of an issue is the handling of events and non-smooth dynamical systems that can be a real challenge. For a detailed introduction to the possible issues, the interested reader is referred to our preliminary publication [58].

Due to the aforementioned reasons, for GPUs, reimplementations of even the simplest Runge–Kutta solvers exist in the literature for many specialised problems: solving chemical kinetics [59–62], simulation in astrophysics [63,64], epidemiological model fitting [65,66] or non-linear dynamical analysis [67,68], to name a few. Some general-purpose solvers, e.g., ODEINT or DifferentialEquations.jl (written in Julia) offer the possibility to transfer the integration procedure to the GPU. However, the problem formulation must fit into a framework suitable for both CPU and GPU computations, which might result in suboptimal exploitation of the processing power of GPUs, as we shall see in Sections 3.2 and 3.3. To the best knowledge of the authors, only a few projects exist for developing a general-purpose solver tuned for GPU hardware. One is the program package MPGOS [69–71] written in C++ that has many built-in features: event handling, efficient trajectory manipulation during the integration, distribution of the workload to many GPUs and an easy way to overlap GPU and CPU computations. Its drawback is that only Runge–Kutta type solvers are available in its present version. Another package is ginSODA suitable for solving stiff ODE systems; however, according to its publication [72], it lacks the aforementioned special features of MPGOS.

One of the main aims of the present paper is to identify the most severe performance bottlenecks using both CPUs and GPUs for performing massive parameter studies and provide efficient solution techniques to overcome these difficulties. The major performance barriers are the lack of the exploitation of SIMD instruction sets of CPUs, the instruction latency on both CPUs and GPUs, and the inefficiency introduced by solving a large number of independent ODE systems with synchronous time stepping. To achieve our goal, confident knowledge of the hardware architectures is mandatory. This study also provides performance comparisons of three program packages by examining three different test cases using both CPUs and GPUs. It must be stressed that *even the sophisticated software packages do not solve the issues mentioned above automatically, the user usually has to be aware of the performance barriers and their possible "remedies" to carry out a highly efficient parameter scan*. With the help of the introduced techniques and ideas, the reader can easily select the best implementation strategy and software package for a given problem. The introduced ideas can also help during the development of a new and efficient solver.

In the following, the three program packages are introduced with preliminary personal opinions in terms of user-friendliness and performance based on the experience of the authors. The first solver tested is MPGOS that is a natural choice as it is developed by Hegedűs, F., who is one of the co-authors of the present paper. It only supports integration on GPUs. Its main features have already been introduced in the previous section. Although MPGOS is written in C++, its interface is user-friendly, and the user needs no knowledge about GPU programming for good performance. The second choice is ODEINT written in C++, that is a general-purpose solver having many features including different types of solvers. However, it is hard to use and not user-friendly. A good performance is expected at least for CPUs. Using the appropriate data structure (e.g., Thrust) integration can also be transferred to the GPU. ODEINT is part of the boost library collection [73]. The third candidate is the program package DifferentialEquations.jl written in Julia and capable of performing integration on both CPUs and GPUs. It has many built-in features: stiff problems via automatic differentiation, delay differential and algebraic differential equations, stochastic differential equations, symplectic solvers, boundary value problems and many more. Julia is a high-level language offering an easy-to-use development environment like Matlab or Python, but it compiles the code before the first run. Therefore, good performance is expected combined with user-friendliness. In the present paper, only the above-described three solvers are tested to remain focused and to avoid an overwhelming flow of data.

Altogether three systems are examined. The first test case performs 1000 steps with the classic fourth-order Runge–Kutta scheme with fixed time steps on the classic Lorenz system [23]. This is a standard test case provided in the majority of the program packages as tutorial or reference example. The amount of work that needs to be done is well defined as it does not depend on the initial conditions or on the parameter sets (fixed time steps, no error control). Thus, there is no thread divergence using GPUs. Second, an amplification diagram is simulated employing the Keller–Miksis equation [24,74] describing the radial pulsation of a spherical gas bubble placed in an infinite domain of liquid. This model is a perfect example to test the codes when even a single trajectory has orders of magnitude differences in the time scale during an integration process. On GPUs, this feature of the system can result in a large amount of thread divergence, or a tremendous amount of overhead of the computation in case the time stepping cannot be separated between the systems. The third model describes the dynamics of a pressure relief valve [37] that can exhibit impact dynamics. In this example, the performance of the packages can be tested when multiple events have to be detected. The events occur at different time instances corresponding to the different systems. Moreover, upon the detection of an impact as an event, the impact law has to be applied immediately on the specific system. It must be stressed that the impacts are not "synchronised", each system has its own unique history of impacts that makes this problem a challenge for GPU codes.

2. Limiting factors of a computation

The performance of an application running on either CPU, GPU or both can be limited by memory bandwidth, by saturation of the compute units or by latency. In case of memory bandwidth limited applications, the bandwidth (usually measured in Gb/s) of the global memory (GPU) or system memory (CPU) bus system is not enough to feed the processing units with enough data. Thus, these units are idle in some portion of the total runtime. A perfect example is a matrix–matrix multiplication with matrices so large that they do not even fit into the last level (L3) cache of the CPU or shared memory of the GPU. In this case, with a naïve approach, the elements of an operation have to be loaded from the global or system memory (again) even if that particular data was already loaded before. The reason is the limited amount of cache: some old data need to be evicted in order to load another portion. The optimal number of load operations is $2 \times N^2$ (every element loaded only once) in contrast to the worst-case scenario: $2 \times N^4$ (all elements are loaded again). Here, $N \times N$ is the size of the matrices. The difference between the optimal and the worst-case scenarios is huge since N is large. It can be calculated by paper and pencil that the memory bandwidth required in the worst-case scenario is usually orders of magnitude larger than the available. Many smart approaches exist to ease the pressure on the global or system memory by exploiting the memory hierarchy of the hardware. For instance, performing as much calculation as possible on submatrices loaded into the L1 cache of the CPU or into the shared memory of the GPU. This follows *the concept of data reuse*.

For low dimensional ODE systems (our case), the available memory bandwidth is usually not an issue. A large portion of the data fits into the registers, or the L1 cache or shared memory. Memory bandwidth plays an important role for very large problems when the ODE system comes, e.g., from the semi-discretisation of one or more partial differential equations. In this case, usually many matrix–vector multiplications are involved during the evaluation of the right-hand side of the ODE function.

The second type of limiting factor of an application is when the compute units of the hardware are fully utilised. This is a desirable operation condition as the main aim is to harness the peak processing power of the hardware. It must be stressed, however, that a compute limited application is not necessarily efficient. It is possible that a wrong choice of a numerical scheme needs orders of magnitude larger number of floating-point operations compared to an optimal algorithm.

The exploitation of the Single Instruction Multiple Data (SIMD) capability of a CPU via the Advanced Vector eXtension (AVX) instruction set is also related to compute unit limitations. In this way, the same operation is performed on different pieces of data. This instruction set uses vector registers YMM/ZMM that are capable of holding 4/8 double, or 8/16 single-precision floating-point numbers, respectively. Thus, an application using doubles can experience a 4/8 times speed-up if the vector capabilities of the CPU can be fully exploited. Conversely, an application cannot even get close to the theoretical peak performance without the use of the AVX instruction set. The AVX instruction set was introduced in the Sandy Bridge (Intel) and Bulldozer (AMD) processors (YMM registers). The AVX2 instruction set has been available since CPU generations Haswell (Intel) and Excavator (AMD). Its main novelty is the support of fused multiply-add (FMA) instructions doubling the peak theoretical performance compared to AVX discussed in detail below. The latest version of the Advanced Vector eXtension set is AVX-512 introduced in the microarchitecture Skylake (Intel). It is capable of performing instructions on 8 doubles or 16 floats, further doubling the peak theoretical performance (ZMM registers).

In terms of implementation, there are two requirements. First, the parallelism has to be increased by 4, 8 or 16 fold depending on the precision and the available instruction set of the hardware. Practically, this can be done by solving 4, 8 or 16 instances of the ODE systems in parallel. Second, employing the vector registers explicitly via their intrinsic functions is a cumbersome task as even a simple addition has to be done via the function call

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
```

for double-precision floating-point numbers. This makes the code hard to read. In addition, there are no intrinsics for trigonometric and other special functions like exponentials, logarithms or powers. Naturally, this difficulty can be overcome with operator and function overloading (this also needs the reimplementation of the special functions). For C++, there are libraries that offer full support of explicit vectorisation. Throughout this paper, the Vector Class Library (VCL) written by Fog, A. [75,76] is applied, which can be perfectly "fused" with ODEINT. In the case of DifferentialEquations.jl, the @avx macro (LoopVectorization.jl package) is an automatic tool to exploit vector register capabilities. However, in our experience, it does not always recognise the vectorisation possibilities. The other option is an own implementation by the user in Julia, which is, again, a cumbersome task.

In the case of GPUs, the program code has to be already fit to the massively parallel environment of the GPU. Therefore, increasing parallelism to exploit the hardware capabilities entirely is self-evident. Two paragraphs at the end of this section are devoted to discussing the different parallelisation possibilities of each program package (for both CPUs and GPUs). The reason is its high impact on the performance of some test cases.

It is important to note that the peak theoretical processing power of the CPUs or GPUs are computed with the assumption that every floating-point instructions are FMAs. An FMA instruction means performing an addition/subtraction and multiplication in a single clock cycle (e.g. $ax + b$). Consequently, a numerical problem consisting only of non-FMA instructions (e.g., sole additions or multiplications) can harness only half the theoretical processing power even in the best-case scenario. Since a Runge–Kutta solver itself includes some non-FMA instructions, 100% exploitation of the peak processing power cannot be expected. Another important issue one has to consider is the division operation, which has a reciprocal throughput of approximately 20–44 clock cycles per instruction. That is, it is a costly operation. Therefore, if a division of a number is required several times, compute its reciprocal in a separate variable and use it as a multiplicator. Transcendental functions are also costly operations as they are decomposed into several micro-operations. For instance, Intel uses polynomial based approximations to compute trigonometric functions. Similarly, if the value of a transcendental function is required with the same argument several times, compute it into a separate variable. Also, many programming languages and libraries provide functions that compute, e.g., both the sine and cosine values approximately within the same time as a single trigonometric function evaluation. Moreover, if possible, replace a general power function with a rational exponent with a combination of square and reciprocal square functions and multiplications/divisions.

If both the utilisation of the memory bus system and the arithmetic processing units are low (the code is neither limited by memory bandwidth nor by the utilisation of the compute units), the limiting factor is the latency. Performing an operation (memory request or instruction), the results are available only after a certain amount of time (measured in clock cycles). This delay is called latency (memory latency or instruction latency). Keep in mind that latency has nothing to do with memory bandwidth (peak data transfer in Gb/s) or arithmetic throughput (peak processing power in FLOPS). Even in case of a request of a single floating-point number from the global/system memory (the memory bandwidth is definitely not saturated), the data is available for computations only after approximately 600 clock cycles. Similarly, even if an addition or a multiplication can be performed within a single clock cycle, the computed results are available for further use only after 3–5 clock cycles (architecture and instruction dependent). Keep in mind that the division operation has a high latency of approximately 21–45 cycles. The transcendental functions are decomposed into additions, multiplications and divisions.

The main strategies of CPUs and GPUs for hiding latency are different. CPUs follow the latency-oriented design; that is, a single thread (or two in case of hyper-threading) in a core uses a small number of registers but a large amount of low-latency L1 and L2 caches (the last level L3 cache is shared among the cores). As long as the whole computational problem fits into the L1 and/or L2 caches, the performance is usually not limited by memory bandwidth or memory latency (this is the case for all the test problems examined in the present paper). Nevertheless, instruction latencies (and vectorisation via the AVX instruction set) still need to be handled by increasing parallelism.

In contrast to CPUs, GPUs follow the memory throughput-oriented design. On the one hand, it uses a huge register file per streaming multiprocessor: 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). An equivalent number for the Ivy Bridge CPU architecture (single core) is $144 \cdot 8 = 1152$ entries of 32 bit registers. On the other hand, a GPU has a small amount of L1 cache or shared memory (user-programmable

L1 cache); their total size for hundreds or thousands of threads is in the order of the L1 and L2 caches of the CPU (for a maximum of 2 threads). In the case of the Kepler architecture, their total size is 64 Kb (GK110) or 128 Kb (GK210). In proportion, the amount of its last-level L2 cache is even smaller: 1536 Kb for tens or hundreds of thousands of threads (the unified last-level L3 cache size of Ivy Bride CPU architecture is 2–8 Mb for 2–8 threads). 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 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 (memory latency) or from a previous computation (instruction latency). 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 latency is not a high priority, the throughput of the global memory bandwidth can be increased drastically. In the case of a simple low order system, like the Lorenz equation, the complete problem fits into the registers, and the application becomes free of global memory transactions.

Finally, let us summarise the parallelisation possibilities. One way to increase parallelism is to replicate the low-dimensional ODE systems and build up a monolithic right-hand side function. Thus, a single ODE function is composed of a multitude of independent sub-ODEs corresponding to different parameter combinations or initial conditions. That is, the evaluation of the right-hand side contains independent instructions suitable for the out-of-order execution of CPUs to hide latency. The explicit vectorisation by the VCL library packs 4 or 8 sub-ODEs into Vec4d or Vec8d data types. In the case of GPUs, each sub-ODE can be assigned to a single GPU thread for evaluation. The user can easily implement this technique without modifying the underlying solver algorithm. Due to the usage of a single monolithic right-hand side function, each sub-ODE must proceed with the same (shared) time step. Therefore, this approach is called *synchronous* parallelisation technique. In ODEINT and DifferentialEquations.jl, this is the only way for parallelisation. For algorithms employing fixed time-stepping, this strategy has no drawbacks; however, for adaptive solvers and in the case of event handling, the synchronous time stepping can have a serious (negative) performance impact, see Sections 3.2 and 3.3 for details. The total number of the packed ODE systems are called *unroll factor* throughout the paper. For CPUs, its value is $n_{avx} \cdot m$ (controlled by the user), where n_{avx} is the unroll factor via explicit vectorisation and m is an additional unroll factor to hide instruction or L1 cache latencies. In the case of GPUs, the unroll factor is always identical to the number of the launched threads.

The program package MPGOS follows a different approach called *asynchronous* time stepping. Instead of building up a single monolithic ODE function, each GPU thread acts as an independent solver of a single instance of the ODE system (having different parameter combinations or initial conditions). That is, for adaptive algorithms, each thread (i.e., ODE system) has its own error control and time stepping. Inevitably, this technique introduces some amount of thread divergences, as the execution time of a warp is determined by the thread having the largest number of time steps; still, in many cases, this is the only way to make GPUs a viable alternative to CPUs. The interested reader is again referred to Sections 3.2 and 3.3 for details.

It is to be stressed, and it is evident from the discussion of this section that maximising the performance requires a proper implementation of the right-hand side of the ODE system. Even a highly optimised solver cannot do optimisations (e.g., explicit vectorisation) without the help of the user. The above-introduced techniques, however, needs knowledge of hardware architectures. Thus, for non-expert users, implementing a highly efficient code is far from trivial. To help "digesting" the principles introduced in this section, the interested reader finds a step-by-step optimisation of a simple test case in Appendix A with code snippets, a minimalist description of hardware architectures and with profiling data on how much the peak processing power of each hardware (CPU or GPU) is exploited. Also, a comprehensive overview of the differences between CPU and GPU architectures and computational strategies are introduced in the textbook [77].

3. Performance characteristics of the test cases

The introductory example in Appendix A was ideal for demonstrating possible performance issues, and investigating basic optimisation techniques. In this section, the performance characteristics of more complex non-linear models are presented, where the runtimes are plotted as a function of the number of the instances of the solved ODE systems. Detailed profiling is omitted here, and the runtimes provide the bases for the comparison of the different program packages. Nevertheless, whenever it is feasible, the techniques revealed by the detailed profiling in Appendix A are employed to maximise performance. For CPU solvers (ODEINT, DifferentialEquations.jl), this means explicit vectorisation and synchronous parallelisation. Even though all the studied program packages are able to shuttle computations to GPUs, only in the case of MPGOS has the user the option to tune the register usage and the size of the thread blocks. In the case of DifferentialEquations.jl, they are completely hidden by the underlying abstraction. In our ODEINT versions, the ODE functions are implemented in a CUDA file using the Thrust library that must be compiled with **nvcc**. Thus, at least the maximum register usage can be managed. Similarly to the Standard Template Library (STL) in C++, Thrust provides many containers and algorithms that can be run on CUDA supported GPUs. Thrust is part of the NVIDIA CUDA framework [78]. In the subsequent sections, three different models are examined with large alteration in complexity and features that need to be handled. During the simulations, only *double-precision* floating-point arithmetics is used.

All of our source codes can be found in a GitHub repository [79]. The applications are tested under Ubuntu 20 LTS operating system. The C++ compiler is gcc 7.5.0, and the CUDA Toolkit version is 10.0. The versions of the program packages is as follows; ODEINT: v2, MPGOS: v3.1, Julia: v1.5.0 and DifferentialEquations.jl: v6.15.0. In case of Julia, many other modules are used: DiffEqGPU v1.6.0, SimpleDiffEq v1.2.1, LoopVectorization v0.8.24.

The CPU codes are tested on a four-core Intel Core i7-4820K processor. During the simulations, only a single core is used with a processing power of 30.4 GFLOPS (double-precision). The length of the vector registers is $n_{avx} = 4$ (AVX instruction set, no FMA is supported). The GPU kernels are tested on an Nvidia GeForce Titan Black card having 1707 GFLOPS peak double-precision performance. As the test cases fit into the L1 cache of the CPU core, there would be no L3 cache contention if all the four cores were exploited. Therefore, employing only a single core is reasonable; the runtime differences and the theoretical peak performance differences of the hardware provide a complete picture of program package performances.

3.1. The Lorenz system

The first test model is the classic Lorenz system written as

$$\dot{x}_1 = 10.0\,(x_2 - x_1), \tag{1}$$

$$\dot{x}_2 = p\,x_1 - x_2 - x_1\,x_3, \tag{2}$$

$$\dot{x}_3 = x_1\,x_2 - 2.666\,x_3. \tag{3}$$

It is composed of three first-order equations, where x_i are the components of the state space, p is a parameter, and the dot stands for the time derivative. The system was first studied by Edward Lorenz for simplified modelling of the atmospheric convection [23]. The system is famous for its chaotic solution, the Lorenz attractor. It has been the subject of hundreds of research articles [20–22,80–82]; also, a complete book is devoted to the study of the Lorenz equations and their non-linear dynamics [83].

During the computations, the control parameter p is varied between 0.0 and 21.0 with a resolution N distributed uniformly. That is, during a single run, N instances of Eqs. (1)–(3) are solved each having a different parameter value. The main objective is to calculate the runtime of 1000 time steps with the classic 4th order Runge–Kutta method (fixed time steps) as a function of N. These performance characteristic curves obtained by using the aforementioned program packages (using both CPUs and GPUs) are compared and examined.

Due to its simplicity, the Lorenz system is a standard test case for many program packages. Note that the right-hand side involves only additions/subtractions and multiplications. In addition, because of the fixed time steps, there is no thread divergence (every instance needs exactly the same number of steps). Therefore, the total amount of work is well-defined, which makes this system a perfect example to test the vectorisation capabilities of the program codes (AVX in CPUs) and the exploitation of the raw peak processing power of the hardware.

3.1.1. Performance curves on CPUs

The performance curves obtained on CPUs are summarised in Fig. 1. The blue and black curves are computations with ODEINT and DifferentialEquations.jl (Julia for short), respectively. The brown curve is our "hand-tuned" version written in C++. The green performance characteristic curve is digitalised from the publication written by the developers of ODEINT [52] (Karsten Ahnert and his co-workers). For the computations by Ahnert et al. an Intel Core i7-920 CPU is employed using all of the 4 cores; the total peak double-precision performance is 42.56 GFLOPS. Code snippets are omitted during the discussion as all the source codes can be found in the GitHub repository [84].

The fastest solver is our own "hand-tuned" implementation in C++ (brown curve), which follows the parallelisation technique described by Lst. 4 in Appendix A.1.4 (the optimal unroll factor is $n_{avx}{\cdot}2$), and it exploits the explicit vectorisation possibility via the VCL library, see Appendix A.1.3. This code is specialised only for the Lorenz system; thus, it is free of any overhead that stems from constraints of general-purpose solvers. This case serves as a reference for comparison with other program packages, and it has approximately a 70% floating-point efficiency.

ODEINT provides the fastest general-purpose alternatives (blue and green curves). Employing only the unroll technique (without vectorisation) to increase parallelism (optimal unroll factor is 8, upper blue curve), the code is approximately 4 times slower than the reference computation. It is very likely that the compiler cannot generate a code to exploit the vectorisation capabilities of the CPU automatically. Therefore, the results of the brown and the upper blue curves are consistent as with full vectorisation, the maximum speed-up is exactly a factor of 4. Interestingly, for the ODEINT implementation, a larger unroll factor was necessary to hide the latency as much as possible. The results taken from [52] indicate a somewhat faster implementation (green curve). However, the total processing power of the used CPU is 42.56 GFLOPS instead of 30.4 GFLOPS. Assuming a linear correlation between the runtimes and the peak processing power and employing a correction factor of $42.56/30.4 = 1.4$ on the green curve, the runtimes of the upper blue and the corrected green curves are within 10% difference. These results also imply consistency. One of the main advantages of ODEINT is its compatibility with different data structures. Therefore, the explicit vectorisation by the VCL library can easily be implemented by replacing the state type from double to Vec4d. In this case, the optimal unroll factor is increased to $n_{avx} \cdot 16$, see the lower blue curve. Including the explicit vectorisation, the speed-up is only $\times 1.75$ between the two blue curves in the asymptotic regime ($N > 40000$), which indicates a suboptimal usage of the vector registers. In summary, our personal view is that ODEINT is a fast solver with CPUs; its only drawback is that the vectorisation capabilities cannot be fully exploited.

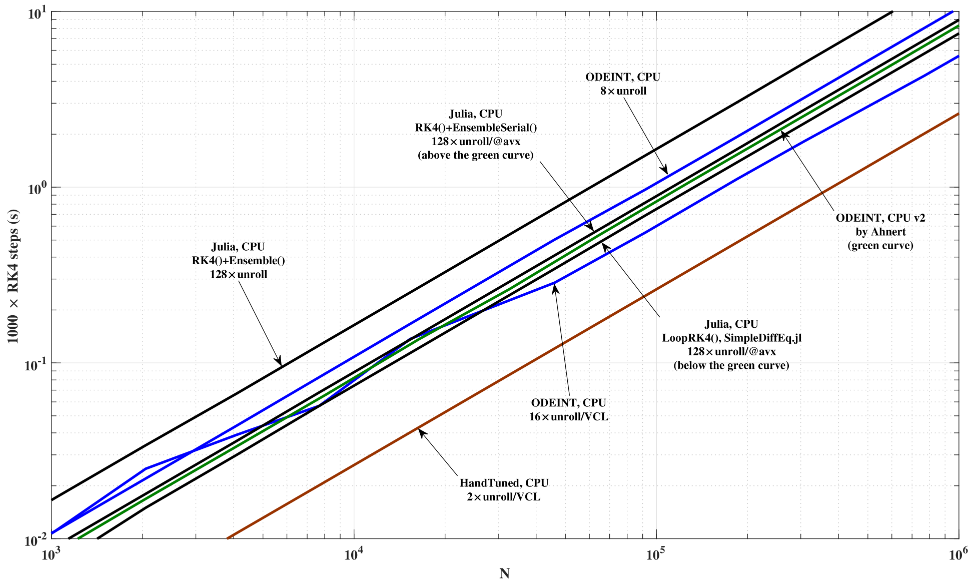

Fig. 1. Performance curves of the Lorenz system; that is, the runtime of 1000 steps with the classic 4th order Runge–Kutta method as a function of the ensemble size N. Brown curve: our "hand-tuned" version written in C++; blue curves: ODEINT; green curve: ODEINT by Ahnert et al. [52]; black curves: Julia (DifferentialEquations.jl).

In general, Julia provides a somewhat slower option to the Lorenz problem. By employing the unroll technique alone (without vectorisation), the code is approximately 6.3 times slower (uppermost black cure) than the baseline "hand-tuned" C++ version (brown curve). This is larger than the optimal 4 times slow down ("hand-tuned" without vectorisation). This implies the presence of some amount of overhead compared to ODEINT, which is also confirmed by the much larger required unroll factor (128) to hide latency. The vectorisation can be employed during the evaluation of the right-hand side by using the @avx macro (LoopVectorization.jl package) on the loop inside the ODE function. The speed-up is $\times 1.85$ (middle black curve) that is lower than the optimal ($\times 4$), but is still larger than the $\times 1.75$ speed-up produced by ODEINT. The runtime can further be decreased by using the LoopRK4() solver from the package SimpleDiffEq.jl (lowermost black curve). This version of the code is still 2.9 times slower than the baseline, and 1.3 times slower than the fastest ODEINT code (lower blue curve). *Although Julia is somewhat slower than ODEINT, it is a high-level programming language in which the code development is extremely easy and fast compared to ODEINT.*

3.1.2. Performance curves on GPUs

The performance curves obtained on GPUs are summarised in Fig. 2 together with the "hand-tuned" baseline CPU solver. The colour code is the same as in the case of Fig. 1. Namely, the blue and black curves are computations with ODEINT and Julia, respectively. The brown curves are our "hand-tuned" versions written in C++ (CPU, dashed) and CUDA C (GPU, solid). Finally, the green performance characteristic curves are again digitalised from the publication written by the developers of ODEINT [52]. During the computations by Ahnert et al. an Nvidia Tesla K20c GPU was employed, which has a peak double-precision performance of 1175 GFLOPS. Code snippets are omitted during the discussion as all the source codes can be found in the GitHub repository [84].

The fastest baseline simulation is shown by the solid brown curve that is free of any overhead. Profiling with **nvprof**, both the optimal block size and maximum register usage (compiler option) are 64. The actual register usage is 40; thus, the 50% theoretical occupancy can easily be achieved. The floating-point efficiency and also the "Issue Stall Reasons (No. Selected)" are above 80% in the asymptotic regime (approximately above $N = 40000$). Therefore, the code is considered to be highly optimised. The folder "ProfileData" of the related GitHub repository contains detailed profiling results for $N = 1536$, 15360 and 768000. This solution is 50 times faster than the baseline CPU solver (dashed brown curve). The GPU, however, has 56 times higher double-precision peak performance than the CPU. This small discrepancy can come from the technique the floating-point efficiency is obtained for CPUs. It must be calculated indirectly from the number of arithmetic instructions and the runtime. In addition, the very low runtime of the GPU computations (e.g., 40 ms at $N = 768000$) can lead to high relative uncertainties, as a small change in the runtime can cause a large speed-up difference.

The performance characteristic curve of the program package MPGOS is close to the baseline simulations in the asymptotic regime where the arithmetic units of the GPU are saturated with enough instructions, and the GPU is fully utilised. The slow down is approximately 1.3 times. This 30% increase in runtime is the price of being general purpose and modular. For low values of N, the difference is much larger (close to a 4 times slow down). However, this range of N represents suboptimal, underutilised usage of the GPUs. Therefore, it has less practical relevance.

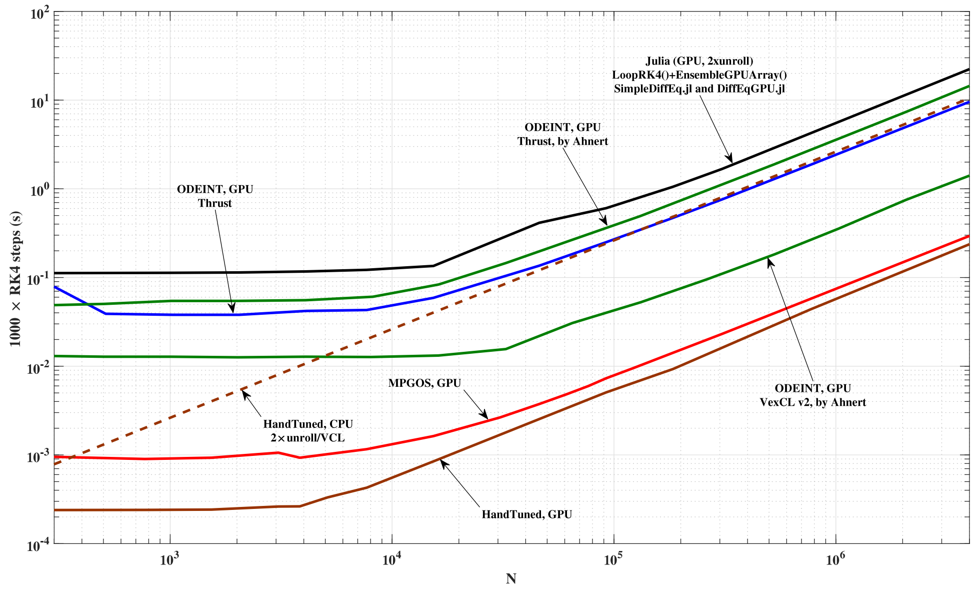

Fig. 2. Performance curves of the Lorenz system; that is, the runtime of 1000 steps with the classic 4th order Runge–Kutta method as a function of the ensemble size N. Brown curves: our “hand-tuned” versions written in C++ or CUDA C; blue curve: ODEINT; green curves: ODEINT by Ahnert et al. [52]; black curve: Julia (DifferentialEquations.jl).

The Julia GPU solver (black curve) and the ODEINT implementations with the Thrust library (blue and upper green curves) have similar performance characteristics. All have poor performance: they have higher or have nearly equal runtimes compared to our baseline CPU solver, although the peak performance of the underlying hardware is more than an order of magnitude higher. The common approach of these implementations is the separate invocation of the kernel function(s) at each time step. This is due to the synchronous parallelisation approach: employing a single monolithic ODE function. Therefore, at the beginning of every time step, all the variables have to be loaded from the slow global memory of the GPU to the registers again. Therefore, there is no chance for data reuse and latency hiding via the fastest register memory, and the applications are memory bandwidth limited. Keep in mind that the latency of the global memory is as high as approximately 600 clock cycles.

Ahnert and his co-workers in their publication [52] provide an alternative solution that builds-up a single monolithic kernel using the library VexCL [85]. The VexCL library is similar to Thrust, but it supports a variety of hardware (Nvidia and AMD/ATI GPUs, multi-core CPUs). The corresponding digitalised runtimes are shown by the lower green curve in Fig. 2. The performance of this case is significantly improved by eliminating the memory bandwidth limiting bottleneck. The runtime difference between this ODEINT version and MPGOS is reduced to ×4.6. However, according to the authors, this ODEINT approach has severe restrictions: *“it only supports embarrassingly parallel problems (no data dependencies between threads of execution), and it does not allow conditional statements or loops with non-constant number of iterations”*. Therefore, the flexibility of the code has been lost, which is important in many situations, see Section 3.2 or Section 3.3.

3.2. The Keller–Miksis equation

The second test model is the Keller–Miksis equation describing the evolution of the radius of a spherical gas bubble placed in a liquid domain and subjected to external excitation [24]. It is a non-linear second-order ordinary differential equation. In order to remain consistent with our previous publications [44,86,87], the dual-frequency version is employed. The system reads as

$$\dot{y}_1 = y_2, \tag{4}$$

$$\dot{y}_2 = \frac{N_{\mathrm{KM}}}{D_{\mathrm{KM}}}, \tag{5}$$

where the numerator, N_{KM}, and the denominator, D_{KM}, are

$$N_{\mathrm{KM}} = (C_0 + C_1 y_2)\left(\frac{1}{y_1}\right)^{C_{10}} - C_2\left(1 + C_9 y_2\right) - C_3\frac{1}{y_1} - C_4\frac{y_2}{y_1} - \left(1 - C_9\frac{y_2}{3}\right)\frac{3}{2}y_2^2 - \left(C_5\sin(2\pi\tau) + C_6\sin(2\pi C_{11}\tau + C_{12})\right)\left(1 + C_9 y_2\right)$$

$$-y_1\left(C_7\cos(2\pi\tau)+C_8\cos(2\pi C_{11}\tau+C_{12})\right), \tag{6}$$

and

$$D_{\mathrm{KM}} = y_1 - C_9 y_1 y_2 + C_4 C_9, \tag{7}$$

respectively. The dimensionless bubble radius is y_1, and the dimensionless bubble wall velocity is y_2. The dot stands for the derivative with respect to the dimensionless time defined as

$$\tau = \frac{\omega_1}{2\pi}t. \tag{8}$$

The system coefficients can be computed in advance of the integration on the CPU side. They are summarised as follows.

$$C_0 = \frac{1}{\rho_L}\left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{9}$$

$$C_1 = \frac{1-3\gamma}{\rho_L c_L}\left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\frac{2\pi}{R_E\omega_1}, \tag{10}$$

$$C_2 = \frac{P_\infty - p_V}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{11}$$

$$C_3 = \frac{2\sigma}{\rho_L R_E}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{12}$$

$$C_4 = \frac{4\mu_L}{\rho_L R_E^2}\frac{2\pi}{\omega_1}, \tag{13}$$

$$C_5 = \frac{P_{A1}}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{14}$$

$$C_6 = \frac{P_{A2}}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{15}$$

$$C_7 = R_E\frac{\omega_1 P_{A1}}{\rho_L c_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{16}$$

$$C_8 = R_E\frac{\omega_1 P_{A2}}{\rho_L c_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{17}$$

$$C_9 = \frac{R_E\omega_1}{2\pi c_L}, \tag{18}$$

$$C_{10} = 3\gamma, \tag{19}$$

$$C_{11} = \frac{\omega_2}{\omega_1}, \tag{20}$$

$$C_{12} = \theta. \tag{21}$$

In the above defined system, $c_L = 1497.3\,\mathrm{m/s}$ and $\rho_L = 997.1\,\mathrm{kg/m^3}$ are the sound speed and density of the liquid domain, respectively. The vapour pressure is $p_V = 3166.8\,\mathrm{Pa}$. The surface tension is $\sigma = 0.072\,\mathrm{N/m}$ and the liquid kinematic viscosity is $\mu_L = 8.902^{-4}\,\mathrm{Pa\,s}$. The ambient pressure is $P_\infty = 1\,\mathrm{bar}$. The parameters of the periodic external forcing are P_{A1} and P_{A2} (pressure amplitudes), $\omega_1 = 2\pi f_1$ and $\omega_2 = 2\pi f_2$ (driving frequencies), and θ is the phase shift between the component of the dual-frequency driving. The polytropic exponent γ for air is chosen as $\gamma = 1.4$ (adiabatic behaviour). Finally, the equilibrium bubble radius is R_E.

The main aim of this study is examining the numerical behaviour of the Keller–Miksis equation instead of performing detailed parameter studies with exhaustive physical interpretation. Thus, only the first frequency component f_1 of the external driving is chosen as a control parameter and the rest is kept fixed ($P_{A1} = 1.5\,\mathrm{bar}$, $P_{A2} = 0\,\mathrm{bar}$, $f_2 = 0\,\mathrm{kHz}$, $\theta = 0$ and $R_E = 10\,\mu\mathrm{m}$). The first frequency component f_1 is varied between 20 kHz and 1 MHz with a resolution N distributed logarithmically. Since the amplitude of the second frequency component is zero, the system is driven only by a single frequency. According to the dimensionless form of the time coordinate given by Eq. (8), the state space is periodic in time with period $\tau_p = 1$. At each frequency value, the maximum value of the dimensionless bubble radius y_1^{max} is extracted from the converged solution. The first 1024 integration phases are regarded as transients and discarded, and the values of y_1^{max} is determined from the subsequent 64 integration phases. A single integration phase means solving the Keller–Miksis equation over the time domain $\tau = [0, 1]$. The plot of y_1^{max} as a function of f_1 with a resolution of $N = 46080$ is depicted in Fig. 3. Such a function is called amplification diagram or frequency response curve. The objective of this section is to compare the runtimes of the computations of such curves with different resolutions N.

The source of the challenge of this problem is the qualitatively different dynamics of the bubbles at different frequency values. This is demonstrated in Fig. 4 where the time series of the dimensionless bubble radius of 2 integration phases

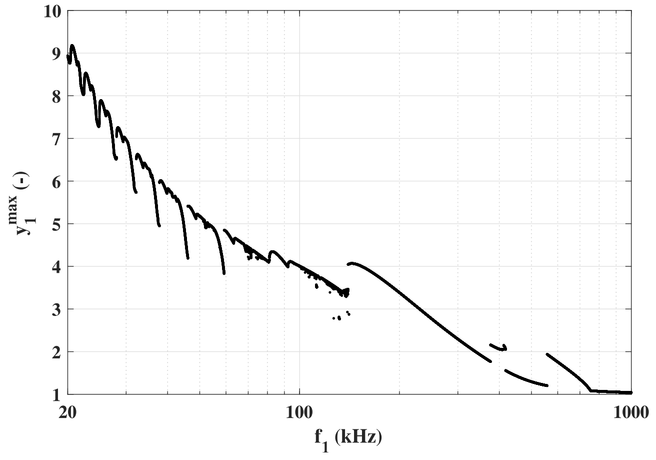

Fig. 3. Frequency response diagram of the Keller–Miksis equation where the maximum value of the dimensionless bubble radius y_1^{max} is plotted as a function of the frequency f_1. The value of f_1 is varied between 20 kHz and 1 MHz with a resolution N distributed logarithmically.

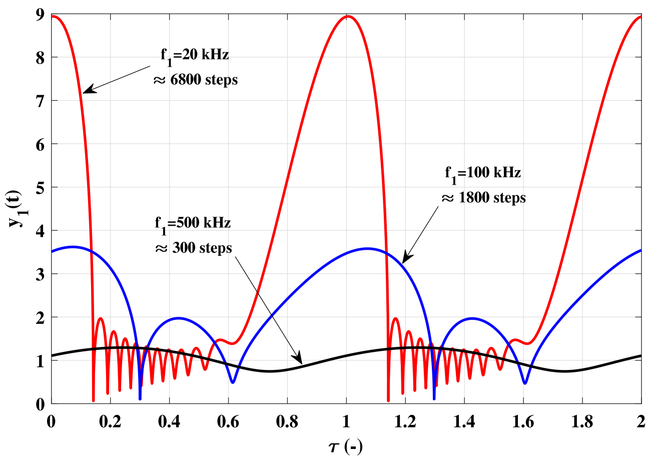

Fig. 4. Typical time series of bubble radii at frequency values $f_1 = 20$ kHz (red), 100 kHz (blue) and 500 kHz (black). With the Runge–Kutta–Dormand–Prince solver with 10^{-10} absolute and relative tolerances, the required time steps are approximately 6800, 1800 and 300.

(after the transients) are plotted for $f_1 = 20$ kHz (red), 100 kHz (blue) and 500 kHz (black). The bubble dynamics at 20 kHz has a slow expansion phase followed by a very rapid contraction (bubble collapse) with some "afterbounces". Clearly, the solution has orders of magnitude difference in its time scales; thus, the application of adaptive solvers is mandatory. With the Runge–Kutta–Dormand–Prince scheme with 10^{-10} absolute and relative tolerances, the maximum and the minimum time steps are $1.2 \cdot 10^{-3}$ and $2.2 \cdot 10^{-9}$, respectively. With increasing frequency, the dynamics become much smoother. The required time steps to keep the prescribed tolerance (given above) are approximately 6800 (20 kHz), 1800 (100 kHz) and 300 (500 kHz).

In case of such systems, it is extremely important to be able to separate the time coordinate of each instance and solve them asynchronously. Otherwise, in packed systems, the synchronous time step will always be determined by the smallest required time step in an ensemble. For instance, solving the three Keller–Miksis equations, presented in Fig. 4, synchronously, the overall number of time steps will be larger than 6800 (worst case individual scenario). The reason is that there are time intervals where the blue curve has a larger error than the red curve. Therefore, the required time step to keep the tolerance at a prescribed level is triggered alternately between the red and blue curves, although the red curve will dominate the time step selection. When the error of the blue curve determines the time step selection, the red curve has to perform a step with a smaller size than necessary. Here we assume that the black curve is so smooth that it never triggers a time stepping. For CPUs, this is a minor issue as with VCL library without further unrolling, only a maximum of 4 Keller–Miksis equations are packed together. However, program packages other than MPGOS can handle ensemble simulations on GPUs only by packing the whole ensemble of oscillators into a single monolithic ODE function. The thousands of these instances might slow each other down resulting even in orders of magnitude larger number of total steps than required. In these cases, one can expect quite poor performance demonstrated clearly in the next section.

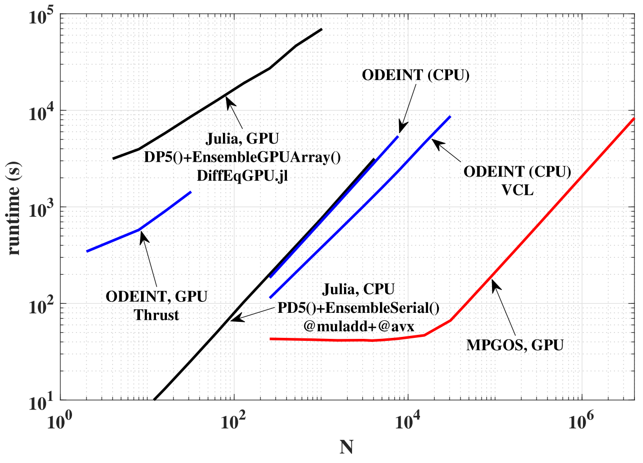

Fig. 5. Performance curves of the Keller–Miksis equation; that is, the runtime of the problem defined in this section as a function of the ensemble size N. Red curve: MPGOS; blue curves: ODEINT; black curves: Julia (DifferentialEquations.jl).

3.2.1. Performance curves on CPUs and GPUs

The performance characteristics are summarised in Fig. 5, where the total runtime of the complete problem is plotted as a function of the resolution N of the frequency range. The runtime involves the 1024 transient integration phases and an additional 64 to determine y_1^{max}. The colour coding is the same as in the case of the previous figures: the red, blue and black curves are related to MPGOS, ODEINT and Julia, respectively. To simplify the discussion, hand-tuned versions and results taken from other publications are omitted here; and curves corresponding to both the CPU and GPU computations are examined together. As usual, the implementations and the source codes can be found in the GitHub repository [88].

In general, the precise location of y_1^{max} is no goal, it is determined as the maximum of the series of points of y_1 given at the locations of the natural time steps of the adaptive solvers. Therefore, the overhead of the root finding of an event handling algorithm is excluded. In addition, whenever it was feasible, the determination of y_1^{max} is done without the production of dense output. This can be done via special functions called after every successful time step, in which only a single parameter is continuously updated. These functions are called *ActionAfterSuccessfulTimeStep* (MPGOS), *observer* (ODEINT) and *CallbackFunctions* (Julia). In the case of using GPUs, it is extremely important to avoid large data transfer through the slow PCI-E bus. In addition, large global memory requirements per thread can significantly reduce the number of the residing threads in a single run, which might decrease the utilisation of the GPU and the latency hiding capabilities, thus performance. In MPGOS and ODEINT, the Runge–Kutta–Cash–Karp algorithm, while in Julia, the Runge–Kutta–Dormand–Prince scheme is applied. Both are 5th order solvers with 4th order embedded error estimation. In all the cases, both the relative and absolute tolerances of the integrators are 10^{-10}.

The fastest implementation is provided by MPGOS (GPU hardware). The feature of the performance curve is very similar to the one of the Lorenz system. There is an initial plateau in the runtimes where the GPU is not fully utilised. This phase is followed by the linear characteristics (doubling the work doubles the runtime). MPGOS solves the ensemble of the Keller–Miksis equations asynchronously; therefore, they do not slow each other down at the collapses, see the related discussion in Section 3.2. Naturally, thread divergence is presented due to the adaptive solvers and the possible large differences in the required number of time steps at different parameter values. However, the asynchronous approach is a viable option here. Compare for example with the GPU versions of ODEINT and Julia, where the ensemble of Keller–Miksis equations can only be solved as a single large ODE system. These implementations are orders of magnitude slower compared to their CPU versions. Thus they are not discussed further.

MPGOS is approximately 130 times faster than the CPU-ODEINT code used with VCL vector class library (lower blue curve in Fig. 5), where the unroll technique is employed only for vectorisation to minimise the slowdown. Keep in mind that the ratio of the peak double-precision floating-point performance between the hardware is 56 indicating that MPGOS harnesses the processing power of the hardware better. One reason is the underutilisation of the vector capabilities of the CPU by ODEINT. The VCL version is only 2.3 times faster than the “ordinary” ODEINT code. The implementation in Julia (CPU version) is only 1.09 times slower compared to the non-VCL ODEINT solver. That is, Julia provides a highly optimised code apart from the exploitation of vectorisation. In the case of Julia, the @muladd (for FMA) and the @avx macros are used with an unroll factor of 4 (this is the minimum required systems for the optimal exploitation of vector registers). However, the @avx macro has no measurable effect on the runtimes, in contrast to the case of the Lorenz system where the speed-up via the @avx macro was approximately a factor 2. Unfortunately, we could not find out the reason of this non-robust behaviour. The speed-up factors η between the different program packages discussed above are summarised in Table 1.

Table 1
Summary of the speed-up factors between the different program packages.

	MPGOS Julia	MPGOS ODEINT	ODEINT Julia	MPGOS ODEINT VCL
η	325	299	1.09	130

3.3. *A system exhibiting impact dynamics*

The last test case is a model that describes the dynamics of a pressure relief valve. The numerical difficulty of this problem is the possible non-smooth impacting behaviour. The dimensionless governing equations are taken from [37] and can be written as

$$\dot{y}_1 = y_2, \tag{22}$$

$$\dot{y}_2 = -\kappa y_2 - (y_1 + \delta) + y_3, \tag{23}$$

$$\dot{y}_3 = \beta(q - y_1\sqrt{y_3}), \tag{24}$$

where y_1 and y_2 are the displacement and the velocity of the valve body, respectively. The pressure relief valve is attached to a reservoir chamber in which the dimensionless pressure is y_3. The control parameter of the system is the dimensionless flow rate q varied between 0.2 and 10 with a resolution of N (uniform distribution). The rest of the parameters are kept constant: $\kappa = 1.25$ is the damping coefficient, $\delta = 10$ is the precompression parameter, $\beta = 20$ is the compressibility parameter.

In Eqs. (22)–(24), the zero value of the displacement ($y_1 = 0$) means that the valve body is in contact with the seat of the valve. If the velocity of the valve body y_2 has a non-zero, negative value at this point, the following impact law is applied:

$$y_1^+ = y_1^- = 0, \tag{25}$$

$$y_2^+ = -r y_2^-, \tag{26}$$

$$y_3^+ = y_3^- \tag{27}$$

That is, the velocity of the valve body is reversed by the Newtonian coefficient of restitution $r = 0.8$ that approximates the loss of energy of the impact.

Fig. 6 shows two examples of impacting solutions at different initial conditions; the control parameter is $q = 0.3$ in both cases. The first numerical challenge is that the different instances of the ODE system can have an impact at different time instances. It is impossible to handle this situation properly when a large number of instances are packed into a single monolithic system as the precise detection of the location of the impact has to be serialised. That is, if a single instance is impacting, all the other instances are blocked from progressing further during the precise impact detection. The necessary control flow operations must be serialised as well. The main reason is that the SIMD lanes in case of vector registers (CPUs) and the threads in a warp (GPU) have to perform the same instruction but on multiple data. For CPUs, it is a minor drawback as usually only a few instances are packed together (4 is enough for Ivy Bridge). Thus, the performance drop can be acceptable. However, in the case of GPUs, where thousands of threads have to work together and block each other, the performance loss can be very severe.

Another issue comes from the autonomous nature of the system. For a feasible investigation of the periodic orbits, a suitable definition of the Poincaré section is mandatory. In case of the pressure relief valve model introduced here, a possible option is the local maximum of the displacement. Therefore, a single integration phase means the integration from a local maximum $y_{1,n}^{max}$ to the next local maximum $y_{1,n+1}^{max}$, see again Fig. 6. This technique needs another event detection procedure for the local maxima (besides the detection of the impact). The consequence of such a definition of the Poincaré section is a difference between the integration time domains of each ODE system instances. Thus, in the case of packed ODEs of synchronous parallelisation, it is *impossible* to stop the integration phases properly. Simply, the final time instances are different, and all the independent systems have to march simultaneously with the same time step. It is an insoluble problem. This is the reason why performance curves are not presented for Julia and ODEINT using GPUs in Section 3.3.1.

Since MPGOS solves all the instances of the ODE system asynchronously, these problems are not serious issues. Naturally, a small amount of thread divergence occurs due to the employed adaptive solver, the detection of the events and the application of the impact law for individual threads. However, the divergent part in an event detection is restricted only to the determination of a corrected time step and the application of the very simple impact law. These operations need a far less number of instructions compared to a complete time step. For the details, the interested reader is referred to our preliminary publication [58] or to the manual of the program package [69].

The objective of this example is the study of the numerical behaviour of the different program packages instead of providing detailed physical investigation and interpretation of the results. In order to achieve this, a bifurcation diagram is generated, where the maximum and minimum displacement of the valve body (y_1^{max} and y_1^{min}) through 32 integration

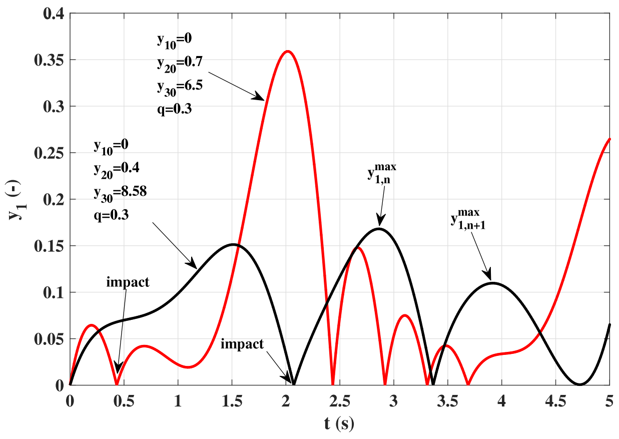

Fig. 6. Typical impacting solutions of the pressure relief valve model described by Eqs. (25)–(27) employing different initial conditions.

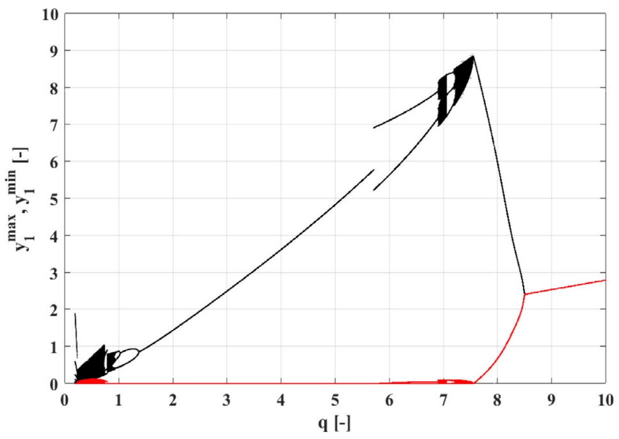

Fig. 7. Bifurcation diagram where the maximum (black) and the minimum (red) values of the valve position y_1 are plotted as a function of the dimensionless flow rate q.

phases are plotted as a function of the control parameter q (dimensionless flow rate). Note that the values of y_1^{max} are the points of the Poincaré section. The values of y_1^{min} are registered only to recognise impacting solutions ($y_1^{min} = 0$). Again, the control parameter q is varied between 0.2 and 10, and the resolution N is the parameter for the performance characteristic curve. In each simulation, the first 1024 iterations are regarded as initial transients and discarded. The data of the next 32 iterations are recorded and written into a text file. A bifurcation diagram with $N = 30720$ is shown in Fig. 7. The black and red dots are the values of y_1^{max} and y_1^{min}, respectively. It is clear that in a wide range of the control parameter (approximately between 0.2 and 7.5), the solutions are impacting. This puts the program packages to the test.

3.3.1. Performance curves on CPUs and GPUs

In this section, the performance characteristic curves are discussed shortly, as all the basics are already examined with great detail in the previous sections. They are shown in Fig. 8, where the total runtime of the complete problem is plotted as a function of the resolution N of the control parameter q. The runtime involves the 1024 transient integration phases and an additional 32 to determine the values of y_1^{max} and y_1^{min}. The colour coding is the same as in the previous cases: the red, blue and black curves are related to MPGOS, ODEINT and Julia, respectively. Due to the aforementioned issues (event handling, impact law and different time domains), only the results of MPGOS are depicted as a sole GPU performance curve. Due to the same complications, the CPU versions do not use the vector registers via the AVX instruction set. In our experience, the vectorised versions have no observable benefit in this problem. Similarly to the example of the Keller–Miksis equation, in MPGOS and ODEINT, the Runge–Kutta–Cash–Karp algorithm, while in Julia, the Runge–Kutta–Dormand–Prince scheme is applied. Both the relative and absolute tolerances of the integrators are 10^{-10}. The absolute tolerance of the impact detection is set to 10^{-6}.

It should be noted that ODEINT does not support automatic root-finding of special points (events). When a possible event is detected via the *observer*, the user has to implement an own algorithm (we used the bisection method) to find the

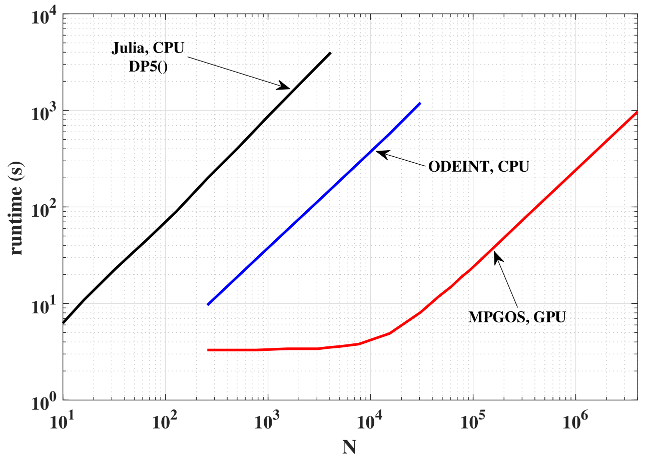

Fig. 8. Performance curves of the pressure relief valve model; that is, the runtime of the problem defined in this section as a function of the ensemble size N. Red curve: MPGOS; blue curve: ODEINT; black curve: Julia (DifferentialEquations.jl).

precise location of the events within a prescribed tolerance. It is important in this case as the global error of the solution is sensitive to the proper application of the impact law.

In the asymptotic (linear) regime, MPGOS is approximately 148 times faster than ODEINT, which is much higher than the ratio of the peak performance difference of the employed hardwares ($\times 57$). This indicates a very efficient implementation of the problem in MPGOS. Comparing the CPU versions, the ODEINT solver is about 26 times faster than Julia. This result is surprising as Julia has approximately the same performance as ODEINT in the case of the Keller–Miksis equation (Section 3.2) and only has slightly larger runtimes for the Lorenz system (Section 3.1). It implies a suboptimal event handling by DifferentialEquations.jl. It is beyond the scope of the present paper to identify the main issue. For further details of the source codes and implementations, see the GitHub repository of the problem [89].

4. Summary and conclusion

The main aim of the present study was to provide a detailed performance comparison of different program packages to solve a large number of independent, low-order, non-stiff ordinary differential equation (ODE) systems. Both the CPU and GPU capabilities of the ODE suits were tested. The three candidates were MPGOS, designed for using only GPUs, ODEINT, written in C++ and DifferentialEquations.jl, implemented in Julia. Three models were tested, each having different features and numerical challenges. First, the Lorenz system was investigated employing the classic fourth-order Runge–Kutta solver (fixed time step). This is a standard example in many program packages and textbooks for measuring the performance of a specific solver. Second, the Keller–Miksis equation, known in sonochemistry, was tested. Here, the large time-scale differences presented in a solution pose a real challenge when exploiting the SIMD units of a hardware, or if a single monolithic ODE system is built-up from a multitude of Keller–Miksis equations. The last example was a model describing the dynamics of a pressure relief valve that can exhibit impacting behaviour. In this case, the solvers needed to handle multiple events and the non-smooth nature of the system (impact). In the last two models, adaptive Runge–Kutta algorithms were used (Cash–Karp or Dormand–Prince).

As a general conclusion, MPGOS is superior using GPUs. In many cases, it has orders of magnitude lower runtimes compared to the other program packages. In addition, it is the only package that could handle the non-smooth dynamical system (pressure relief valve). Therefore, it is a perfect tool for parameter studies and non-linear analysis in high dimensional parameter space [16,90–102]. For CPUs, in many cases, ODEINT and Julia have approximately the same performance. Therefore, Julia is a viable option compared to other program packages written in low-level languages. However, Julia has some overhead in case of event handling, and ODEINT is superior in these problems. It must also be noted that ODEINT lacks root finding, and for precise event detection, the users have to write their own algorithm.

Our personal impression about the program packages is as follows. Julia is a high-level programming language and thus specifically designed to be user-friendly allowing quick code development with minimal learning time. Its CPU performance can be considered excellent (except for cases with event handling), but for GPUs, the runtimes show poor efficiency. Nevertheless, it must be stressed that the package has many other features not shown here; for instance, automatic differentiation to calculate the Jacobi for stiff problems, using arbitrary-precision numbers or the support of delay differential equations, to name a few. Moreover, it has excellent support, and the newer versions of the code are expected with performance improvements.

ODEINT is written in the low-level C++ programming style. Due to the exhaustive usage of template metaprogramming, the CPU related codes have excellent performance, and it provides a relatively easy means to fuse the application with other libraries. For instance, with the Vector Class Library for explicit vectorisation. It provides multistep and symplectic integrators, as well as stiff solvers, too. However, the package is far from user-friendly. One needs a very long learning period to be able to build-up applications dissimilar to the provided tutorial examples. The performance in terms of GPU usage is poor.

MPGOS fuses the user-friendliness and high-performance into a single ODE suite. Its interface is easy to use, and the right-hand side of the ODE function can be implemented in a similar way as in the case of Julia or MATLAB. In addition, it also provides low-level means for fine-tuning and handling specialised problems. The main drawback of MPGOS is the lack of additional features as those of Julia and ODEINT; it supports only Runge–Kutta type solvers. However, MPGOS is continuously under development, and the interested user should regularly check its new features in the official website [70].

CRediT authorship contribution statement

Dániel Nagy: Software, Investigation. **Lambert Plavecz:** Software, Investigation. **Ferenc Hegedűs:** Writing – original draft, Writing – review & editing, Visualisation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The research reported in this paper is part of project no. BME-NVA-02, implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021 funding scheme. This paper was also supported by the Alexander von Humboldt Foundation, Germany (HUN 1162727 HFST-E), by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences (Ferenc Hegedűs), and by the New National Excellence Program of the Ministry of Human Capacities, Hungary, project no. ÚNKP-20-1-I-BME-168 (Dániel Nagy), ÚNKP-20-1-I-BME-186 (Lambert Plavecz) and ÚNKP-21-5-BME-369 (Ferenc Hegedűs).

Appendix A. Detailed demonstration of the programming techniques, brief introduction of the hardware architectures and profiling

In this section, the techniques to harness the peak processing power of CPUs and GPUs are demonstrated via the simple one-dimensional ODE system written as

$$\dot{x} = x^2 - p, \tag{A.1}$$

where x is the sole state variable and p is a parameter. Since this paper focuses on extensive parameter studies, many instances of Eq. (A.1) are solved each having a different parameter value. The dot stands for the derivative with respect to time. The numerical algorithm is the classic fourth-order Runge–Kutta scheme with fixed time steps. Thus, the required number of arithmetic operations is independent of the initial condition and the parameter value. The reason for the choice of Eq. (A.1) is the structure of its right-hand side; namely, it consists of exactly one addition and one multiplication (a single FMA instruction).

A.1. Maximising the performance on CPU

A.1.1. The naïve approach

The simplest (naïve) approach to solve a large ensemble of Eq. (A.1) is simply to loop through the parameter values and perform the integration one after another. The corresponding simplified code snippet is listed in Lst. 1. The control parameter p is varied between 0.1 and 1.0 with a resolution of $N = 2^{16} = 65536$ on an equidistant grid. Altogether $n = 1000$ fourth-order Runge–Kutta steps are computed. The dense output is not stored; thus, a single Runge–Kutta step updates the state variable x. As the system is autonomous, it is unnecessary to register the actual time t. The final time instances can be obtained by $T = n \cdot dt$.

The runtime of the code was 2.383 s on the Intel Core i7-4820K CPU (Ivy Bridge architecture) using a single core. Although the runtime is an important measure of code performance, it does not provide any information on how much of the peak theoretical performance of the CPU core is harnessed. For this, a specialised profiling technique has to be used in order to monitor the total number of floating-point instructions, see Appendix A.1.2 for more details. For code snippet Lst. 1, it turned out that the achieved GFLOPS (Giga floating-point operations per second) is 1.155, whereas the peak theoretical power of the core is 30.4 GFLOPS. This means a mere 3.9% FLOPS efficiency. This value is surprisingly low and to improve the code, a deeper understanding of the hardware architecture and the profiling technique is required, see Appendix A.1.2.

Listing 1: Simplified code snipped of the naïve approach for solving Eq. (A.1).

```
const int N = 1<<16;
const int n = 1000;
const double dt = 0.01;

double* P = linspace(0.1, 1.0, N);
double x;
double p;

for (int i=0; i<N; i++)
{
    p = P[i];
    x = -0.5;

    for (int j=0; j<n; j++)
        OneStep(x,p,dt);
}
```

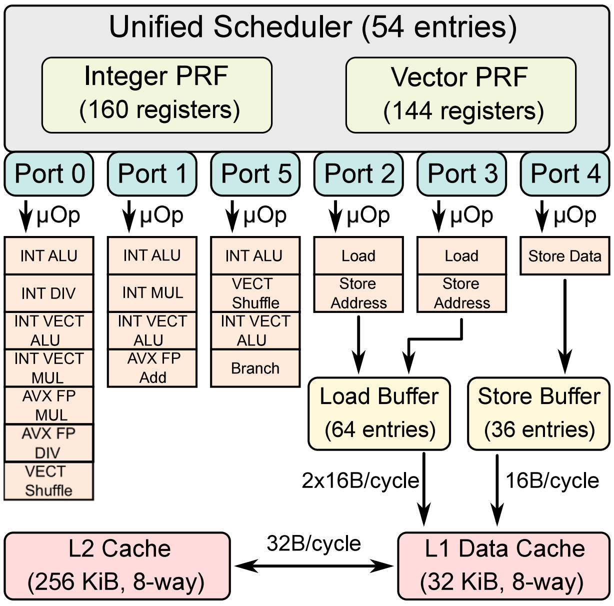

Fig. A.9. The compute engine (back end) of the Intel Ivy Bridge micro-architecture.

A.1.2. Basics of CPU architectures and profiling

The basics of the CPU architectures are introduced via the description of the compute engine (back end) of the Intel Ivy Bridge micro-architecture, see Fig. A.9. This is the architecture of the CPU used throughout this study. In spite of this specific example, the main conclusions hold for other types of processors as well. In Fig. A.9, the front end responsible for the decoding of the instructions into micro-operations are omitted as usually it is not the bottleneck of the solution of ODEs. The execution engine of Ivy Bridge (and every modern CPUs) executes micro-operations in an out-of-order fashion. That is, the instructions are not necessarily executed exactly in the order as they are in the assembly code. Any micro-operation can be dispatched to one of the six ports for execution if it does not depend on the results of other micro-operations. In this sense, a maximum of six micro-operations can be dispatched for execution at the same time. This represents the instruction-level parallelism (ILP) of the CPU, which can be harnessed by the synchronous or the asynchronous parallelisation techniques discussed in Section 2. Note how the different ports are responsible for the execution of different types of instructions. For example, ports 0, 1 and 5 are responsible for arithmetic computations, while ports 2–4 are responsible for memory management (load/store and address calculation). Therefore, floating-point calculations and load requests for subsequent computations can be overlapped.

From Fig. A.9, it is clear that a floating-point multiplication (port 0) and a floating-point addition (port 1) can be performed simultaneously. This alone implies that a certain amount of ILP must be presented in the code. The instruction throughput of floating-point addition/subtraction and multiplication is 1 per clock cycle. However, the latency

Listing 2: Typical output of a profiling with the utility **perf**.

```
10.049.141.981   Number of clock cycles
18.775.076.456   Number of instruction
 9.759.952.395   Counts 256-bit packed double-precision
 6.579.729.696   L1-dcache-loads
       119.756   L1-dcache-load-misses
 4.252.220.453   L1-dcache-stores
        57.138   L1-dcache-store-misses
 4.698.542.488   Cycles in which a uop is dispatched on port 0
 5.065.437.974   Cycles in which a uop is dispatched on port 1
 5.264.870.995   Cycles in which a uop is dispatched on port 2
 5.568.711.371   Cycles in which a uop is dispatched on port 3
 4.261.921.012   Cycles in which a uop is dispatched on port 4
 1.979.687.970   Cycles in which a uop is dispatched on port 5
 6.955.451.533   Cycles stalled due to Resource Related reason
 2.684.174.187   Cycles stalled due to no elig. RS entry avail.
 4.639.678.096   Cycles stalled due to no store buffers avail.
         7.148   Cycles stalled due to re-order buffer full
```

of multiplication is 5 cycles, and the latency of addition/subtraction is 3 cycles. Therefore, an even larger "amount" of ILP must be presented in the code to be able to feed ports 0 and 1 with enough instructions. The issue of fully hiding the latency is problem-dependent, one has to make experiments system-by-system. Observe that the Ivy Bridge microarchitecture does not support the fused multiply-add (FMA) instructions. This feature, however, is already built-in in the next-generation Haswell processors. Nevertheless, the equal number of addition/subtraction and multiplication is a must to get close to the peak theoretical performance.

The division operation is dispatched to port 0. Again, it has a reciprocal throughput of 20–44 clock cycles per instructions, and latency of 21–45 cycles. In addition, it blocks port 0 for multiplications further reducing the performance of the code. The negative effect of including division is demonstrated in Appendix A.3. For a thorough analysis of the throughput and latency of different instructions on different microprocessor architectures (Intel, AMD and VIA), the interested reader is referred to the excellent work of Fog, A. [103].

Observe the AVX prefix in front of the floating-point addition, multiplication and division panels in Fig. A.9. This means the support of the Advanced Vector eXtension instruction set to perform the Single Instruction Multiple Data (SIMD) operations. The Ivy Bridge architecture is capable of holding 4 double or 8 single-precision floating-point numbers. In Appendix A.1.3, the usage of VCL library is demonstrated to ensure the usage of vector registers for parameter studies of ODE systems.

In order to get a detailed picture about the behaviour of the CPU core, the **perf** profiler utility [104] was used, which is a tool for Linux based systems. It can collect events and metrics from the Performance Monitoring Unit (PMU) of the processor such as the number of cycles, instructions retired, L1 cache misses or instructions issued to each port, to name a few. Although the achieved floating-point performance and efficiency cannot be retrieved directly, they can be calculated by the elapsed time and the number of the executed floating-point instructions. A typical output for a few events are presented in Lst. 2. The discussion of the full capabilities of **perf** is beyond the scope of the present paper. As a tabulated data, the most important events and indirectly calculated data (e.g., achieved GFLOPS and FLOP efficiency) are summarised in Table A.2. The column with the header *simple* is related to the naïve approach introduced in Appendix A.1.1. It can be seen that the number of clock cycles of the "stall reason due to no eligible RS entry is available" ("No RS" in the table for short) is only slightly smaller than the total number of clock cycles. The abbreviation RS is the Reservation Station, also called the Unified Scheduler, see Fig. A.9. This implies that the CPU is idle most of the time because data is not available from a previous computation (most likely) or it has not arrived from the memory subsystem (less likely, small cache misses). This explains the poor performance and low FLOPS efficiency. Note that Table A.2 collects profiling data also for all the other versions of the introductory example by Eq. (A.1) discussed in the following subsections, and for the test cases of using divisions and transcendental functions, see Appendix A.3.

A.1.3. Explicit vectorisation

In Appendix A.1.2, it has been shown that vectorisation (using the vector registers) is mandatory for good performance. An optimising compiler might automatically vectorise the code; for instance, by unrolling a loop by a factor of two, four or eight depending on the available instruction set. The unrolling can be done if the loop body can be executed independently. This is exactly the case for the outer cycle in Lst. 1. However, due to the inner loop, the relatively complex structure of a single step and the call to an "external" ODE function, the compiler cannot recognise that cycling through the parameters are independent and can be done in parallel in a vectorised form. Therefore, in most of the cases, the best practice is to make the vectorisation by ourselves. We call this explicit vectorisation. Keep in mind that throughout this paper, the Vector Class Library (VCL) written by Fog, A. [75,76] is applied.

Table A.2
Summary of the condensed representation of the profiling information for the introductory examples. The peak processing power of the employed Intel Core i7-4820K CPU is 30.4 GFLOPS (single core). The operating system is Ubuntu 20 LTS, and the C++ compiler is gcc 7.5.0.

Code	simple vcl unroll	simple vcl	simple –	transc. vcl unroll	division vcl unroll
Unroll	8	1	1	2	2
Runtime [s]	0.1342	0.5975	2.383	2.125	1.004
Dev [s]	0.0002	0.0010	0.0047	0.1645	0.0018
Clock Cycles	$4.960 \cdot 10^8$	$2.209 \cdot 10^9$	$8.811 \cdot 10^9$	$7.643 \cdot 10^9$	$3.711 \cdot 10^9$
Number of Instr.	$9.193 \cdot 10^8$	$7.602 \cdot 10^8$	$3.025 \cdot 10^9$	$1.277 \cdot 10^{10}$	$8.587 \cdot 10^8$
Number of X87	10480	17180	41680	87620	72620
Number of SSE	$1.319 \cdot 10^5$	$1.319 \cdot 10^5$	$2.753 \cdot 10^9$	$5.247 \cdot 10^8$	132100.
Number of AVX	$7.088 \cdot 10^8$	$6.882 \cdot 10^8$	0	$5.736 \cdot 10^9$	$8.197 \cdot 10^8$
GFLOPS	21.12	4.607	1.155	11.04	3.267
Efficiency [%]	71.36	15.56	3.904	37.31	11.04
L1 Cache Loads	$3.413 \cdot 10^8$	$1.510 \cdot 10^6$	$2.235 \cdot 10^6$	$5.097 \cdot 10^9$	$3.434 \cdot 10^7$
L1 Cache Misses	$9.378 \cdot 10^4$	$1.128 \cdot 10^5$	$1.839 \cdot 10^5$	$2.055 \cdot 10^5$	$1.280 \cdot 10^5$
Divider Active	$5.082 \cdot 10^4$	$8.369 \cdot 10^4$	$2.004 \cdot 10^5$	$4.067 \cdot 10^5$	$3.687 \cdot 10^9$
Divider [%]	0.01025	0.0037	0.0022	0.0053	99.35
No store buffer	$1.185 \cdot 10^6$	391500.	545400.	519000.	419700.
No RS	$2.581 \cdot 10^8$	$1.996 \cdot 10^9$	$8.018 \cdot 10^9$	$4.060 \cdot 10^9$	$3.410 \cdot 10^9$

Listing 3: Simplified code snippet of the explicit vectorisation approach for solving Eq. (A.1).

```
#include "vectorclass.h"
...
const int N = 1<<16;
const int n = 1000;
const double dt = 0.01;

double* P = linspace(0.1, 1.0, N);
Vec4d x;
Vec4d p;

for (int i=0; i<N; i+=4)
{
    p.load_a(P + i); // Loads parameters from aligned memory
    x = -0.5;

    for (int j=0; j<n; j++)
        OneStep(x,p,dt);
}
```

Listing. 3 demonstrates that with the VCL library, only minor modifications are necessary to exploit the vectorisation capabilities of a CPU. Besides the inclusion of the related header file, only the data types are replaced from double to Vec4d (packed 4 doubles), a special function is used to load 4 parameters from a memory location, and the counter of the outer loop is incremented by 4. Naturally, inside the stepper and the ODE function, the data types have to be replaced as well (not shown here).

The profiling results of the new code are shown in Table A.2 in the column with the header *simple vcl*. There is a massive improvement (×3.988) in the runtime indicating the full exploitation of the capabilities of the vector registers. Accordingly, the calculated FLOPS efficiency is also increased approximately by a factor of 4; however, it is still only 15.56% of the theoretical peak performance. As the stall reason due to no RS entry (last row in Table A.2) is still very high, the code is still bound by latency. Since the number of the L1 cache loads are very low, the latency is caused by the dependency of the arithmetic operations (instruction latency).

A.1.4. Further increasing the parallelism

Due to the reasons discussed in Appendix A.1.2 (out-of-order execution, the latency of the instructions), the further increase of the parallelism is usually necessary to harness the full potential of the CPU cores. Solving ODE systems, two main approaches already discussed in Section 2. For the present case, the synchronous approach defines an ODE function

Listing 4: Simplified code snippet of the instruction level parallelism approach for solving Eq. (A.1).

```
#include "vectorclass.h"
...
const int N = 1<<17;
const int n = 1000;
const int m = 8;
const double dt   = 0.01;
const double dtp2 = dt/2;

double* P = linspace(0.1, 1.0, N);
Vec4d x[m];
Vec4d p[m];
Vec4d kSum[m];
Vec4d kAct[m];

const int PStep = 4*m;
for (int i=0; i<N; i+=PStep)
{
    for (int j=0, offset=i; j<m; j++, offset+=4)
    {
        p[j].load_a(P+offset); // Load parameters
        x[j] = -0.5;
    }

    for (int j=0; j<n; j++)
    {
        // Unrolled automatically by the compiler
        for (int k=0; k<m; k++) // k1
        {
            F(&(kAct[k]), x[k], p[k]); // k1 = F(x)
            kSum[k] = kAct[k];
            kAct[k] = x[k] + dtp2*kAct[k];
        }
        ...
    }
}
```

as

$$
\begin{aligned}
&\dot{x}_1 = x_1^2 - p_1,\\
&\dot{x}_2 = x_2^2 - p_2,\\
&\dots\\
&\dot{x}_m = x_m^2 - p_m.
\end{aligned}
\tag{A.2}
$$

That is, altogether m one-dimensional ODE systems described by Eq. (A.1) are solved simultaneously. As each line in Eq. (A.2) is independent, they can be evaluated in an out-of-order fashion. Naturally, this approach can be combined with vectorisation where each line in Eq. (A.2) represents a pack of 2, 4 or 8 systems (depending on the size of the vector registers). It means an unroll of $2m$, $4m$ or $8m$ of the parameters.

For our "hand-tuned" C++ codes, the asynchronous approach is applied where the time coordinates can be kept separated, and each pack of systems (packed only via the VCL library) can be solved "asynchronously". The idea is to transfer the unroll procedure inside the solver and do it at a stage-level of the Runge–Kutta algorithm. The extended code snippet for this approach is presented in Lst. 4, where only the first stage of the full code is shown for brevity. The compiler can unroll the innermost loop automatically, as its structure is simple enough, with a constant unroll factor m (known at compile time). Thus, several independent code blocks are created, increasing the available ILP, which can easily be regulated by the unroll factor m. Note that the variables x and p are arrays of size m and of type Vec4d (packed 4 doubles). Accordingly, the outermost loop responsible for cycling through the parameters has to be incremented by $4m$ that is also the number of the simultaneously solved ODE systems.

Observe that for cases with fixed time steps, there is no real difference between the two strategies: unroll at the ODE function level (synchronous parallelisation) or unroll at Runge–Kutta stage level (asynchronous parallelisation). However, the latter one allows the use of different time steps by defining a vector as follows

```
const Vec4d dt[m];
```

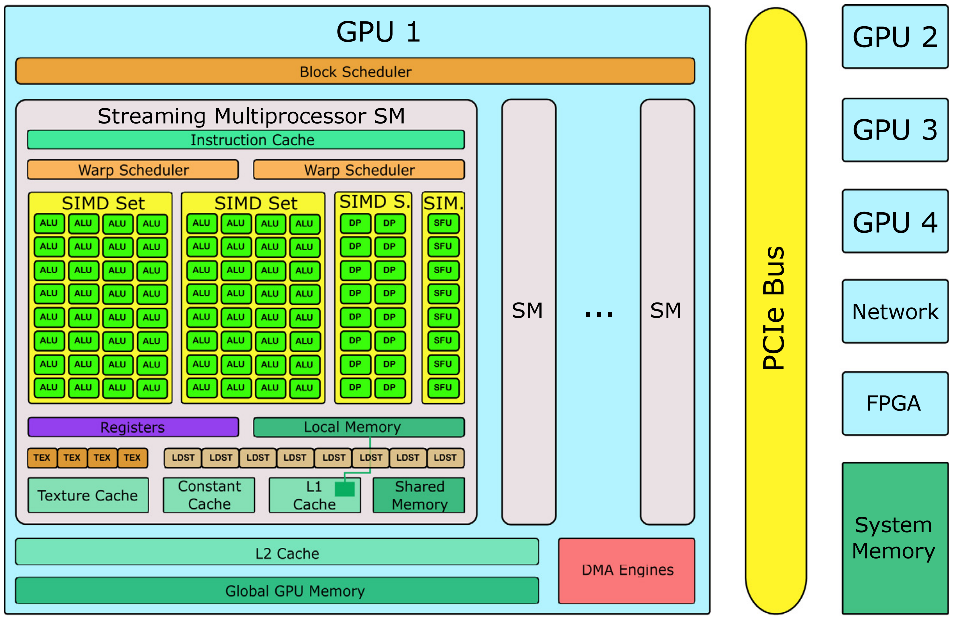

Fig. A.10. A general block diagram of GPU architectures. The details (e.g., the number of warp schedulers and compute units per streaming multiprocessor) are revision-specific features of the architectures. Thus they can vary from GPU to GPU.

and apply different time steps for different instances of Eq. (A.1) at the Runge–Kutta stages of an adaptive solver. In this way, all instances are solved asynchronously. As the solver shown in Lst. 4 is the classic fourth-order Runge–Kutta scheme, the implementation of the application of different time steps unnecessarily increases the overhead without a real benefit (thus it is excluded).

The profiling results for the optimal unroll factor ($m = 8$) are presented in Table A.2 at the column with the header *simple vcl unroll*. Again, a massive performance increase can be observed. The code is now close to the theoretical peak processing power (up to 71%). The stall reason due to no RS entry (last row in Table A.2) is dropped by almost an order of magnitude. Interestingly, the L1 cache loads increased drastically implying that the larger number of ODE systems cannot fit into the physical registers of the core. That is, not only the dependency of the arithmetic instructions are responsible for the latency but also the data requests from the L1 cache. For more detailed information on the effect of the unroll factor m (up to $m = 16$), the interested reader is referred to the GitHub repository of the problem [105]. Proposing further code optimisation strategies are beyond the scope of the present paper. With the explicit vectorisation and the unroll techniques, a large portion of the peak performance of the CPU can be harnessed.

A.2. Maximising the performance on GPU

A.2.1. Basics of GPU architectures and profiling

The basic logical unit performing calculations is a thread. The number of threads simultaneously residing in a GPU can be in the order of hundreds of millions. This is the reason for the widely used term: massively parallel programming. In general, threads in a GPU are organised in a 3D structure called a grid. For our purpose, a 1D organisation is sufficient. That is, a unique identifier of a thread can be characterised by a 1D integer coordinate. The parallelisation technique to solve an ensemble of ODE systems is simple: a single instance of the ODE system, e.g., Eq. (A.1), is assigned to a single GPU thread. That is, the different threads work on different sets of data (parameters or initial conditions). This is the well-known per-thread approach [59].

Each GPU consists of one or more streaming multiprocessors (SMs) which are at the highest level of hierarchy in the hardware compute architecture. Each SM contains many processing units capable of performing integer or single-precision floating-point operations (ALU), double-precision floating-point operation (DP), load/store operations from the global memory or from the L1 cache/shared memory (user-programmable cache) or other specialised instructions, see Fig. A.10. The logical threads have to be mapped to the GPU architecture. For this purpose, the threads are partitioned into thread blocks. The workload in a GPU is distributed to the streaming multiprocessors (SMs) with block granularity. That is, the block scheduler of the GPU fills every SM with blocks of threads until reaching hardware or memory resource limitations.

The thread blocks are further divided into smaller chunks of execution units called warps. Each warp contains 32 threads (as a current CUDA architectural design). The warp schedulers of an SM take warps and assign them to execution units if there are eligible warps for executions and there are free execution units on the SM. A warp is eligible if there is no data dependency from another computational phase or all the required data has already arrived from a memory load operation. Every thread in a warp performs the same instruction but on different data. This is the SIMD unit of a GPU. Therefore, every thread in a warp executes their instruction stream in lock-step. This is the only way the hardware can efficiently handle a massive number of threads: for 32 threads, only one control unit is necessary to track their program state. This results in more place for compute units and higher arithmetic throughput. If an SIMD set in the SM has not enough compute units (e.g., the number of the DP units are 16 in an SM in Fig. A.10), the whole warp is executed in multiple launches (in the case of DP unit, only half of the warp is executed at the same time).

The drawback of the SIMD approach is the possibility of thread divergence occurring when some threads have to do different instructions from the others in the same warp. The simplest case is when the number of the required time steps are different for each system (thread) in an adaptive algorithm. The total execution time of a warp will be determined by its slowest thread forcing the others to be idle. *To minimise the effect of thread divergence, the consecutive systems should have similar parameters and/or initial conditions to ensure similar dynamical behaviour in a warp.*

It is common in both CPUs and GPUs that the system or global memory cannot feed the compute units (in general) with enough data to utilise them fully. Therefore, each architecture heavily relies on its memory subsystem to ease the pressure on the slow system or global memories. The register memory and cache hierarchy are the main components of the memory subsystem. Registers are the fastest memory types (practically without latency); every instruction can be performed only on data already residing in the registers. The cache hierarchy is a tool to prefetch data to fast, low latency memory types and keep the data there for frequent reuse (if possible). Going higher in the cache hierarchy—from L3 (only in CPUs) to L2 and L1—their sizes but also the latencies decrease. The main strategies of CPUs and GPUs for hiding latency are already discussed in Section 2; therefore, it is not repeated here.

Although the amount of registers per thread is relatively high, it is still a scarce resource and the major limiting factor for the maximum number of residing threads in an SM. Keep in mind that more residing threads mean more residing warps to hide latency. In one extreme, using the maximum allowed 255 registers per threads, the maximum number of residing threads in an SM is $65536/255 \approx 256$. At the other extreme, for 2048 residing threads (hardware limit of Kepler), the number of registers per threads should not be higher than $65536/2048 = 32$. The maximum number of registers per thread can be managed by the compiler option

```
--maxrregcount.
```

Another important (hardware) limiting factor is the maximum number of residing blocks in an SM (16 in Kepler). By setting the block size to 32 threads (warp size), the maximum number of residing threads is $16 \cdot 32 = 512$. The ratio of the theoretical maximum of the residing threads and the maximum allowed by the hardware limitation is called occupancy O, e.g., $O = 512/2048 = 0.25 = 25\%$. There are many other resource-related or hardware limiting factors that affect the achievable occupancy. Its calculation is highly non-trivial; thus, the user is referred to the official occupancy calculator of Nvidia [106].

In general, for latency-limited applications, the basic strategy is to increase the occupancy. Higher number of residing warps means a higher possibility of hiding latency. However, when the block size is already large enough, the occupancy can only be increased by decreasing the maximum registers per threads. If the available registers per thread are insufficient, some variables are "spilt" back to global memory further increasing pressure on global memory. Finding an optimal setup (maximum registers, block size and the total number of threads) is not trivial and highly problem dependent.

In some ODE systems (other than the test cases presented here), the on-chip shared memory of the GPU (user-programmable cache) can also be exploited. Moreover, many other issues can affect the final performance of the application (e.g., frequent PCI-E memory transactions), for the details, see our previous publication [58] and the manual of MPGOS [69].

A.2.2. Maximising FLOPS efficiency

In this section, the effect of the total number of threads and the block size of a GPU simulation is demonstrated on the example defined by Eq. (A.1). The structure of the kernel function is very similar to the one shown in Lst. 1; thus, it is not repeated here. The full code can be found in the corresponding GitHub repository [105]. No special care is taken for unrolling and explicit vectorisation in the program level: it follows the naïve approach. The latency is handled by the execution configuration (block size) and the register usage. This kernel function is called by every thread and executed independently but with different parameter values (selected by a unique thread identifier). Compiling the code with the option

```
--ptxas-options=-v,
```

the register usage per thread is printed, which is only 16 in this case (32 is enough for $O = 100\%$ occupancy). Therefore, it is a perfect example of how to deal with latency via the block size and the total number of threads when the number of registers is not an issue.

First of all, to distribute the workload to the streaming multiprocessors (SM) evenly, the block size B and the total number of threads N (also the total number of instances of the ODE system solved simultaneously) have to be properly

Table A.3
Summary of the condensed representation of the profiling information for the introductory example. The peak double-precision processing power of the employed Nvidia GeForce Titan Black GPU is 1707 GFLOPS.

B (block size) N (total threads)	32 480	32 7680	32 61440	64 61440	128 61440
Runtime [μs]	114.56	125.67	961.42	918.00	868.52
SM activity [%]	95.03	94.83	98.53	94.48	97.14
Achieved occup.	0.0156	0.245	0.249	0.460	0.878
Elig. warps	0.24	4.8	4.9	17.0	42.1
Efficiency [%]	4.97	74.79	79.25	84.62	86.86
Exec. dep. [%]	91.03	83.13	83.11	47.84	24.92
Not sel. [%]	2.52	10.25	10.41	48.31	71.02

Listing 5: Typical output of a profiling with the utility **nvprof**.

```
Multiprocessor Activity                                    96.41%
Achieved Occupancy                                       0.402929
Eligible Warps Per Active Cycle                         13.914399
Floating Point Operations(Double Precision Add)          15360000
Floating Point Operations(Double Precision Mul)                 0
Floating Point Operations(Double Precision FMA)         153600000
Integer Instructions                                      3932160
FLOP Efficiency(Peak Double)                               82.36
L1/Shared Memory Utilisation                             Low (1)
L2 Cache Utilisation                                     Low (1)
Device Memory Utilisation                                Low (1)
Load/Store Function Unit Utilisation                     Low (1)
Arithmetic Function Unit Utilisation                    Max (10)
Issue Stall Reasons (Pipe Busy)                             0.66
Issue Stall Reasons (Execution Dependency)                 52.36
Issue Stall Reasons (Data Request)                          0.26
Issue Stall Reasons (Not Selected)                         43.02
```

selected. The example is tested on an Nvidia Titan Black GPU that have $N_{SM} = 15$ SMs. To distribute the same amount of blocks to each SM, the total number of threads have to be $N = B \cdot N_{SM} \cdot i$, where i is an arbitrary positive integer; otherwise, some SMs will be idle near the end of the simulations. The block size B should be an integer multiple of the warp size 32; otherwise, during the execution of the last warp, some compute units will be idle. Five configurations are tested, and their profiling data is summarised in Table A.3.

To acquire the profiling data, the Nvidia profiler tool **nvprof** is employed. It enables one to collect events and metrics for kernel functions from the command line. A typical output for a few metrics are presented in Lst. 5. The discussion of the full capabilities of **nvprof** is beyond the scope of the present paper, the employed shell script and the complete profiling data can be found in the GitHub repository [105] of the problem. With the profiler, one can obtain several useful pieces of information directly. For instance, the distribution of floating-point addition, multiplication and fused multiply-add instructions. From them, the maximum possible floating-point efficiency (exploitation of the peak performance in percentage) can be calculated: 95%. In addition, the achieved floating-point efficiency can be measured directly (82.36%). The utilisation of the memory subsystem (cache hierarchy) and the different compute units are important metrics to determine what is the main performance-limiting factor of the application (memory bandwidth, compute units or latency). Next, the stall reasons can further narrow the focus to solve the most severe bottleneck. Finally, the multiprocessor activity is a good measure of how evenly the blocks are distributed to the SMs.

Table A.3 summarises some key metrics for the five different kernel launch configurations of the example given by Eq. (A.1). As a reminder, the task is to perform 1000 steps with the classic fourth-order Runge–Kutta scheme on an ensemble of N ODE systems. Applying a block size of $B = 32$, the theoretical occupancy is only $0.25 = 25\%$. Observe that the runtimes in the first two columns have minor difference even though the total number of solved systems differ by more than an order of magnitude. This is a perfect example that enough threads have to be launched to fully utilise the GPU. The runtime scales linearly approximately above ($N = 7680$), compare columns 2 and 3. That is, the utilisation of the GPU is saturated, which can also be seen from the achieved occupancy. It is close to the theoretical one. Although the floating-point efficiency is already high (above 70%), the stall reasons due to execution dependency still above 80% indicating latency-limited behaviour caused by the arithmetic instructions. The "remedy" is to increase the occupancy and hide latency. It must be stressed that a high percentage of a stall reason does not necessarily indicate poor performance as the number of the stalled cycles can still be small. It is demonstrated by the high floating-point efficiencies in the second and third columns in Table A.3. The major limiting factor of the occupancy is the maximum number of blocks per SM (16): $O = 16 * 32/2048 = 0.25 = 25\%$. In the last two columns of Table A.3, the block size is increased to 64 and

128 to increase occupancy to 50% and 100%, respectively. The latency due to execution dependency gradually disappears, and the major stall reason becomes the "Issue Stall Reasons (No. Selected)". This stall reason means that there are more eligible warps in the SM to perform instruction execution than compute units, and the warp scheduler chooses a different warp to execute. Thus, high floating-point efficiency and high stall reason of "Not Selected" implies high performance.

A.3. The effect of mathematical functions and division

In order to "feel" the consequences of the usage of division and special functions, two additional micro-benchmark examples are provided. They are tested only on the CPU (Ivy Bridge); however, the conclusions are valid for GPUs as well. To test the effect of division, Eq. (A.1) is modified to

$$\dot{x} = \frac{1}{x} - p \cdot x. \tag{A.3}$$

The corresponding profiling results are shown in the last column of Table A.2. The divider is almost 100% active, the floating-point efficiency dropped down to 11% and the runtime is almost an order of magnitude higher than the optimal code (first column) for Eq. (A.1). The optimal unroll factor here is 2, and the VCL library is still used for explicit vectorisation. The cache loads are relatively low; thus, latency by memory is not an issue. On the other hand, the stall due to execution dependency is very high (last row). The reason is the high latency of the division instruction (approximately 21–45 clock cycles). In addition, the divider blocks port-0 for performing multiplication. *The conclusions also hold for general integer division.*

To test the effect of mathematical functions, Eq. (A.1) is modified again as follows

$$\dot{x} = p \sin(x) \tag{A.4}$$

to include the transcendental function sine. The column with the header *transc. vcl unroll* depicts the profiling results. Although the runtime is more than two times higher than in the case of division, the floating-point efficiency is much higher (37.3%). The reason is the way the hardware handles special functions. Even if the instruction set supports the evaluation of such a function, the computation process is decomposed into several micro-operations. For instance, Intel uses polynomial based approximations to compute trigonometric functions. Therefore, the higher runtime is due to the increased number of required instructions, see again Table A.2. In case of no hardware support of a special function, external libraries have to be used to approximate by software. Another important issue is the exploitation of vector registers. Fortunately, the VCL library supports the evaluation of mathematical functions (transcendental, power or root). It is claimed that the vectorised versions are much faster than most of the commonly used scalar alternatives [75]. The effect of the increased number of operations and the larger number of required intermediate variables can also be seen in the very high number of L1 cache loads. This test has the highest pressure on the L1 cache. Thus, the stall reasons are mainly due to the L1 cache latency. For more detailed instructions and guidance, the reader is referred to the documentation of the VCL library [75] for CPUs, and to the CUDA documentation [107] in case of GPUs.

Appendix B. Relationship between the tolerances, accuracy and the amount of work in the case of the Keller–Miksis equation

In Fig. 5, the performance curves are plotted with both absolute and relative tolerances of 10^{-10}. This is reasonable as the user can manage the global error of the solution via the desired local truncation error. However, the global error over a longer integration phase can vary between the program packages. The reasons are the different types of numerical algorithms employed (Cash–Karp and Dormand–Prince), and the slightly different approach of handling the error control. Therefore, the issue of the tolerances of the local error, the achieved global error (accuracy) and the amount of function evaluations (work) need to be addressed. *In this section, the values of the absolute and relative tolerances are the same for every simulation; therefore from now on, we shall refer to them only as the tolerance T for simplifying the discussion.*

One of the major problems with the Keller–Miksis equation is the absence of analytical solutions for calculating the global error precisely. Therefore, numerical experiments are done on a large scale of tolerances (from $T = 10^{-4}$ to 10^{-15}), where the solution with the most stringent tolerance (10^{-15}) is considered as "exact". This approach is feasible as the "exact" solution is accurate up more than 4 decimal places compared to the solutions with $T = 10^{-10}$ (our main interest here). In addition, the long term solution can be chaotic, that prevents any direct comparison of the global error on a large integration time domain. Thus, the integration time domain is restricted to $\tau = [0, 2]$ (two excitation periods) to minimise the effect of chaotic solutions, and the values of the first components of the state space at the integration phase $y_1(2)$ are compared. In this way, the absolute error can be defined as $E = |y_1(2) - y_1^E(2)|$, where $y_1^E(2)$ corresponds to the solution with tolerance $T = 10^{-15}$ (the "exact" solution). Performing detailed analysis on a large number of frequencies is a cumbersome task; thus, only three characteristic frequency values are examined. Namely, $f_1 = 20\,\text{kHz}$, $100\,\text{kHz}$ and $500\,\text{kHz}$, see again Fig. 4. In the case of ODEINT, the VCL library is not used in this investigation to be consistent with the Julia code. During the comparison of MPGOS and the VCL version of ODEINT, we assumed a constant performance difference of a factor of 2.3 according to the results of Fig. 5.

The results of the numerical experiments are summarised in Fig. B.11. As usual, the colour code is as follows: red is MPGOS, blue is ODEINT and black is Julia. The solid, dashed and the dotted lines are related to $f_1 = 20\,\text{kHz}$, $100\,\text{kHz}$

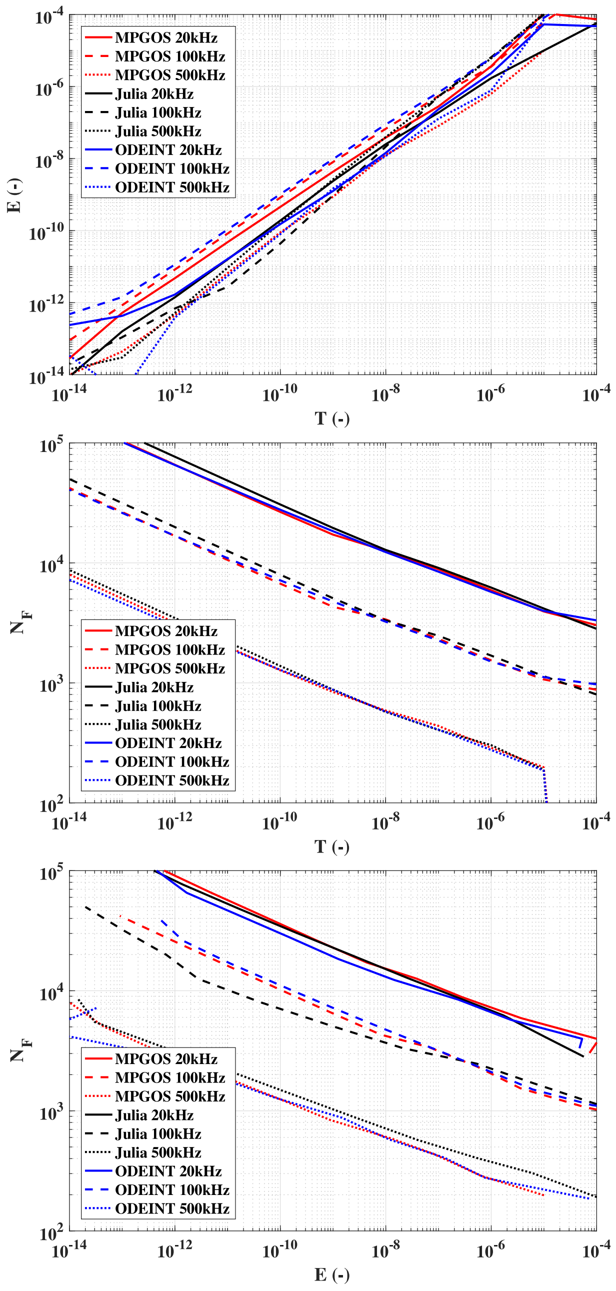

Fig. B.11. Correlation between the tolerance T, global error E and number of function evaluations N_F.

and 500 kHz, respectively. The upper panel shows the error E as a function of the tolerance T, which is important in verifying the convergence of the solutions by applying more stringent tolerance. However, this graph alone is not suitable for judging the real performance of a solver as the lower global error might also be the result of a higher number of steps.

Thus, the number of the total function evaluations N_F (a measure of the total amount of work) has to be included in the analysis. In the middle panel of Fig. B.11, it is plotted as a function of the tolerance. Although Julia always needs slightly more function evaluations (Dormand–Prince) compared to the other program packages (Cash–Karp), for a fixed frequency value, N_F is approximately the same for all cases. Note how the curves having the same style run close to each other. For tolerance $T = 10^{-10}$, the data is presented in Table B.4, where N_F^M, N_F^J and N_F^O are the total number of the function evaluations of MPGOS, Julia and ODEINT, respectively. The results of MPGOS are considered as the baseline, and Table B.4 also shows the ratio with Julia (≈ 14% average increase) and ODEINT (≈ 4% average increase). Accordingly, slightly modified speed-up factors η_W can be defined with the average ratio of the function evaluations to indicate the

Table B.4
Number of function evaluations N_F^M (MPGOS), N_F^J (Julia) and N_F^O (ODEINT) for different frequencies using a tolerance of $T = 10^{-10}$. The last line shows the average ratio of the corresponding function evaluations.

f	N_F^M	N_F^J	N_F^M/N_F^J	N_F^O	N_F^M/N_F^O
20	26580	30675	1.154	27744	1.044
100	6714	7965	1.186	7152	1.065
500	1272	1383	1.084	1282	1.008
Avg.			$\approx$ 1.14		$\approx$ 1.04

Table B.5
Number of function evaluations N_F^M (MPGOS), N_F^J (Julia) and N_F^O (ODEINT) for different frequencies using a global error of $E = 10^{-10}$.

f	N_F^M	N_F^J	N_F^M/N_F^J	N_F^O	N_F^M/N_F^O
20	36000	34500	0.96	30000	0.83
100	10200	7050	0.69	11100	1.08
500	1246	1492	1.20	1247	1.00

Table B.6
Summary of the original and corrected speed-up factors between the different program packages.

	MPGOS Julia	MPGOS ODEINT	ODEINT Julia	MPGOS ODEINT VCL
η	325	299	1.09	130
η_W	285	287	0.99	124
η_E	196–342	238–309	0.63–1.44	103–134

speed-up for the same amount of work. These are listed in the second line of Table B.6. For instance, between MPGOS and Julia, it is calculated with the following expression

$$\eta_W = \eta/1.14. \tag{B.1}$$

Note that in terms of η_W, the 9% performance difference between ODEINT and Julia disappears. Since, Julia does approximately 9% more work (function evaluation).

Finally, the speed-up factors can further be adjusted to determine the performance differences for a prescribed global error. In the bottom panel of Fig. B.11, the number of the function evaluations N_F is plotted as a function of the global error E. Using this diagram, for $E = 10^{-10}$, the data are presented in Table B.5. For simplicity, from now on, the notations N_F^M, N_F^J and N_F^O are related to a global error of $E = 10^{-10}$ instead of to the tolerance of $T = 10^{-10}$. The number of function evaluations in Table B.5 show much larger deviations between the different program packages compared to the values in Table B.4. For instance, Julia needs more than 30% fewer function evaluations than MPGOS to keep the global error below $E = 10^{-10}$ at frequency value $f = 100\,\mathrm{kHz}$. However, the trend is completely the opposite in the case of $f = 20\,\mathrm{kHz}$; it needs approximately 20% more function evaluation. Therefore, instead of determining an average modification factor, we present a range of speed-up factors according to the worst and best-case scenarios. The speed-up factors in terms of fixed global error are denoted by η_E, and between MPGOS and Julia its range is calculated as follows

$$\begin{aligned} \eta_E^{min} &= 0.69 \cdot \eta_W, \\ \eta_E^{max} &= 1.20 \cdot \eta_W. \end{aligned} \tag{B.2}$$

The ranges of the adjusted speed-up factors are printed in the last line of Table B.6. Despite the large differences in the minimum and maximum values, MPGOS is the only viable option for GPUs. In addition, in terms of η_E, ODEINT and Julia have approximately the same performance on average, although there are large differences between the minimum and maximum values of η_E.

References

[1] Luo ACJ, Xing S. Multiple bifurcation trees of period-1 motions to chaos in a periodically forced, time-delayed, hardening Duffing oscillator. Chaos Solitons Fractals 2016;89:405–34.
[2] Englisch V, Parlitz U, Lauterborn W. Comparison of winding-number sequences for symmetric and asymmetric oscillatory systems. Phys Rev E 2015;92(2):022907.
[3] Bonatto C, Gallas JAC, Ueda Y. Chaotic phase similarities and recurrences in a damped-driven Duffing oscillator. Phys Rev E 2008;77(2):026217.
[4] Englisch V, Lauterborn W. Regular window structure of a double-well Duffing oscillator. Phys Rev A 1991;44(2):916–24.
[5] Gilmore R, McCallum JWL. Structure in the bifurcation diagram of the Duffing oscillator. Phys Rev E 1995;51:935–56.

[6] Parlitz U, Lauterborn W. Superstructure in the bifurcation set of the Duffing equation $\ddot{x}+d\dot{x}+x+x^3=f\cos(\omega t)$. Phys Lett A 1985;107(8):351–5.
[7] Krajňák V, Wiggins S. Dynamics of the Morse oscillator: Analytical expressions for trajectories, action-angle variables, and chaotic dynamics. Int J Bifurcation Chaos 2019;29(11):1950157.
[8] Medeiros ES, Medrano-T RO, Caldas IL, de Souza SLT. Torsion-adding and asymptotic winding number for periodic window sequences. Phys Lett A 2013;377(8):628–31.
[9] Knop W, Lauterborn W. Bifurcation structure of the classical Morse oscillator. J Chem Phys 1990;93(6):3950–7.
[10] Scheffczyk C, Parlitz U, Kurz T, Knop W, Lauterborn W. Comparison of bifurcation structures of driven dissipative nonlinear oscillators. Phys Rev A 1991;43(12):6495–502.
[11] Goswami BK. Flip-flop between soft-spring and hard-spring bistabilities in the approximated Toda oscillator analysis. Pramana 2011;77(5):987–1005.
[12] Goswami BK. Self-similarity in the bifurcation structure involving period tripling, and a suggested generalization to period n-tupling. Phys Lett A 1998;245(1–2):97–109.
[13] Goswami BK. 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 1996;122(4):189–99.
[14] Kurz T, Lauterborn W. Bifurcation structure of the Toda oscillator. Phys Rev A 1988;37:1029–31.
[15] Deng Q, Zhou T. Memory-induced bifurcation and oscillations in the chemical Brusselator model. Int J Bifurcation Chaos 2020;30(10):2050151.
[16] Gallas JAC. Periodic oscillations of the forced Brusselator. Modern Phys Lett B 2015;29(35–36):1530018.
[17] Xu Y, Luo ACJ. Frequency-amplitude characteristics of periodic motions in a periodically forced van der Pol oscillator. Eur Phys J Special Top 2019;228:1839–54.
[18] Mettin R, Parlitz U, Lauterborn W. Bifurcation structure of the driven van der Pol oscillator. Int J Bifurcation Chaos 1993;03(06):1529–55.
[19] Parlitz U, Lauterborn W. Period-doubling cascades and devil's staircases of the driven van der Pol oscillator. Phys Rev A 1987;36(3):1428–34.
[20] Meucci R, Salvadori F, Naimee KA, Brugioni S, Goswami BK, Boccaletti S, et al. Attractor selection in a modulated laser and in the Lorenz circuit. Phil Trans R Soc A 2008;366:475–86.
[21] Goswami BK. Control of multistate hopping intermittency. Phys Rev E 2008;78:066208.
[22] Goswami BK. Controlled destruction of chaos in the multistable regime. Phys Rev E 2007;76:016219.
[23] Lorenz EN. Deterministic nonperiodic flow. J Atmos Sci 1963;20(2):130–41.
[24] Lauterborn W, Kurz T. Physics of bubble oscillations. Rep Progr Phys 2010;73(10):106501.
[25] Zhang Y, Zhang Y. Chaotic oscillations of gas bubbles under dual-frequency acoustic excitation. Ultrason Sonochem 2018;40:151–7.
[26] Zhang Y, Zhang Y, Li S. Combination and simultaneous resonances of gas bubbles oscillating in liquids under dual-frequency acoustic excitation. Ultrason Sonochem 2017;35:431–9.
[27] Zhang Y, Zhang Y, Li S. The secondary Bjerknes force between two gas bubbles under dual-frequency acoustic excitation. Ultrason Sonochem 2016;29:129–45.
[28] Yasui K, Tuziuti T, Lee J, Kozuka T, Towata A, Iida Y. The range of ambient radius for an active bubble in sonoluminescence and sonochemical reactions. J Chem Phys 2008;128(18):184705.
[29] Haghi H, Sojahrood AJ, Kolios MC. On amplification of radial oscillations of microbubbles due to bubble-bubble interaction in polydisperse microbubble clusters under ultrasound excitation. J Acoust Soc Am 2018;143(3):1862.
[30] Haghi H, Sojahrood AJ, De Leon Al C, Agata Exner A, Kolios MC. Experimental and numerical investigation of backscattered signal strength from different concentrations of nanobubble and microbubble clusters. J Acoust Soc Am 2018;144(3):1888.
[31] Haghi H, Sojahrood AJ, Karshafian R, Kolios MC. Numerical investigation of the subharmonic response of a cloud of interacting microbubbles. J Acoust Soc Am 2017;141(5):3493.
[32] Gyllenberg M, Jiang J, Niu L, Yan P. On the dynamics of multi-species ricker models admitting a carrying simplex. J Difference Equ Appl 2019;25(11):1489–530.
[33] Cenci S, Saavedra S. Structural stability of nonlinear population dynamics. Phys Rev E 2018;97:012401.
[34] Zeeman ML. Hopf bifurcations in competitive three-dimensional Lotka–Volterra systems. Dyn Stab Syst 1993;8(3):189–216.
[35] Hős CJ, Champneys AR, Paul K, McNeely M. Dynamic behaviour of direct spring loaded pressure relief valves in gas service: II reduced order modelling. J Loss Prev Proc 2015;36:1–12.
[36] Hős CJ, Champneys AR, Paul K, McNeely M. Dynamic behavior of direct spring loaded pressure relief valves in gas service: Model development, measurements and instability mechanisms. J Loss Prev Proc 2014;31:70–81.
[37] Hős C, Champneys AR. Grazing bifurcations and chatter in a pressure relief valve model. Physica D 2012;241(22):2068–76.
[38] Hős C, Champneys AR, Kullmann L. Bifurcation analysis of surge and rotating stall in the Moore–Greitzer compression system. IMA J Appl Math 2003;68(2):205–28.
[39] Molnar TG, Dombovari Z, Insperger T, Stepan G. On the analysis of the double Hopf bifurcation in machining processes via centre manifold reduction. Philos Trans R Soc Lond Ser A Math Phys Eng Sci 2017;473(2207):20170502.
[40] Altintas Y, Stepan G, Merdol D, Dombovari Z. Chatter stability of milling in frequency and discrete time domain. CIRP J Manuf Sci Tech 2008;1(1):35–44.
[41] Takács D, Stépán G. Experiments on quasiperiodic wheel shimmy. J Comput Nonlinear Dyn 2009;4(3):031007.
[42] Takács D, Stépán G, Hogan SJ. Isolated large amplitude periodic motions of towed rigid wheels. Nonlinear Dynam 2008;52:27–34.
[43] Lai YC, Tél T. Transient chaos. New York: Springer; 2010.
[44] Hegedűs F, Klapcsik K, Lauterborn W, Parlitz U, Mettin R. GPU accelerated study of a dual-frequency driven single bubble in a 6-dimensional parameter space: The active cavitation threshold. Ultrason Sonochem 2020;67:105067.
[45] Hairer E, Nørsett SP, Wanner G. Solving ordinary differential equations I. 2nd ed.. Berlin: Springer-Verlag; 1993.
[46] Hairer E, Wanner G. Solving ordinary differential equations II. 2th ed.. Berlin: Springer-Verlag; 1991.
[47] https://unige.ch/~hairer/software.html.
[48] Hindmarsh AC. ODEPACK, a systematized collection of ODE solvers. In: Stepleman RS, editor. Scientific computing. Amsterdam: North-Holland; 1983, p. 55–64.
[49] https://computing.llnl.gov/casc/odepack/.
[50] Hindmarsh AC, Brown PN, Grant KE, Lee SL, Serban R, Shumaker DE, et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. ACM Trans Math Software 2005;31(3):363–96.
[51] https://computing.llnl.gov/projects/sundials.
[52] Ahnert K, Demidov D, Mulansky M. Solving ordinary differential equations on GPUs. In: Kindratenko Volodymyr, editor. Numerical computations with GPUs. Springer International Publishing; 2014, p. 125–57.
[53] http://headmyshoulder.github.io/odeint-v2/.
[54] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html.
[55] Shampine LF, Reichelt MW. The MATLAB ODE suite. SIAM J Sci Comput 1997;18(1):1–22.
[56] https://diffeq.sciml.ai/latest/.

[57] Rackauckas C. A comparison between differential equation solver suites in MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran. Winnower 2018;6:e153459.98975.
[58] Hegedűs F. Program package MPGOS: Challenges and solutions during the integration of a large number of independent ODE systems using GPUs. Commun Nonlinear Sci Numer Simul 2021;97:105732.
[59] Stone CP, Alferman AT, Niemeyer KE. Accelerating finite-rate chemical kinetics with coprocessors: Comparing vectorization methods on GPUs, MICs, and CPUs. Comput Phys Comm 2018;226:18–29.
[60] Niemeyer KE, Sung CJ. Accelerating moderately stiff chemical kinetics in reactive-flow simulations using GPUs. J Comput Phys 2014;256:854–71.
[61] Stone CP, Davis RL. Techniques for solving stiff chemical kinetics on graphical processing units. J Propul Power 2013;29(4):764–73.
[62] Shi Y, Green WH, Wong H-W, Oluwole OO. Accelerating multi-dimensional combustion simulations using GPU and hybrid explicit/implicit ODE integration. Combust Flame 2012;159(7):2388–97.
[63] Brock B, Belt A, Billings JJ, Guidry M. Explicit integration with GPU acceleration for large kinetic networks. J Comput Phys 2015;302:591–602.
[64] Dindar S, Ford EB, Juric M, Yeo YI, Gao J, Boley AC, et al. Swarm-NG: A CUDA library for parallel n-body integrations with focus on simulations of planetary systems. New Astron 2013;23–24:6–18.
[65] Kovac T, Haber T, Reeth FV, Hens N. Heterogeneous computing for epidemiological model fitting and simulation. BMC Bioinformatics 2018;19(1):101.
[66] Al-Omari A, Arnold J, Taha T, Schüttler HB. 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 2013;1:770–7.
[67] Fazanaro FI, Soriano DC, Suyama R, Madrid MK, Oliveira JR, Muñoz IB, et al. Numerical characterization of nonlinear dynamical systems using parallel computing: The role of GPUs approach. Commun Nonlinear Sci Numer Simul 2016;37:143–62.
[68] Rodríguez M, Blesa F, Barrio R. OpenCL parallel integration of ordinary differential equations: Applications in computational dynamics. Comput Phys Comm 2015;192:228–36.
[69] Hegedűs F. MPGOS: GPU accelerated integrator for large number of independent ordinary differential equation systems. Budapest, Hungary: Budapest Univesity of Technology and Economics; 2019.
[70] www.gpuode.com.
[71] https://github.com/FerencHegedus/Massively-Parallel-GPU-ODE-Solver.
[72] Nobile SM, Cazzaniga P, Besozzi D, Mauri G. ginSODA: Massive parallel integration of stiff ODE systems on GPUs. J Supercomput 2018;75(12):1–12.
[73] https://www.boost.org/.
[74] Keller JB, Miksis M. Bubble oscillations of large amplitude. J Acoust Soc Am 1980;68(2):628–33.
[75] Fog A. VCL C++ vector class library manual. Copenhagen, Denmark: Technical University of Denmark; 2020.
[76] https://github.com/vectorclass.
[77] Soyata T. GPU parallel program development using CUDA. Boca Raton, Florida: CRC Press; 2018.
[78] https://docs.nvidia.com/cuda/thrust/index.html.
[79] https://github.com/nnagyd/ode_solver_tests.
[80] Immler F. A verified ODE solver and the Lorenz attractor. J Autom Reasoning 2018;61:73–111.
[81] Graça DS, Rojas C, Zhong N. A verified ODE solver and the Lorenz attractor. Trans Amer Math Soc 2018;370:2955–70.
[82] Stewart I. The Lorenz attractor exists. Nature 2000;406:948–9.
[83] Sparrow C. The Lorenz equations: Bifurcations, chaos, and strange attractors. New York: Springer-Verlag; 1982.
[84] https://github.com/nnagyd/ode_solver_tests/tree/master/Lorenz_RK4.
[85] https://github.com/ddemidov/vexcl.
[86] Hegedűs F, Lauterborn W, Parlitz U, Mettin R. Non-feedback technique to directly control multistability in nonlinear oscillators by dual-frequency driving. Nonlinear Dynam 2018;94(1):273–93.
[87] Hegedűs F, Krähling P, Aron M, Lauterborn W, Mettin R, Parlitz U. Feedforward attractor targeting for non-linear oscillators using a dual-frequency driving technique. Chaos 2020;30(7):073123.
[88] https://github.com/nnagyd/ode_solver_tests/tree/master/Keller_Miksis_RK45.
[89] https://github.com/nnagyd/ode_solver_tests/tree/master/Valve_RK45.
[90] Freire JG, Calderón-C A, Varela H, Gallas JA C. Phase diagrams and dynamical evolution of the triple-pathway electro-oxidation of formic acid on platinum. Phys Chem Chem Phys 2020;22(3):1078–91.
[91] Varga R, Klapcsik K, Hegedűs F. Route to shrimps: Dissipation driven formation of shrimp-shaped domains. Chaos Solitons Fractals 2020;130:109424.
[92] Marcondes DWC, Comassetto GF, Pedro BG, Vieira JCC, Hoff A, Prebianca F, et al. Extensive numerical study and circuitry implementation of the watt governor model. Int J Bifurcation Chaos 2017;27(11):1750175.
[93] Freire JG, Gallas JAC. Stern–Brocot trees in cascades of mixed-mode oscillations and canards in the extended Bonhoeffer–van der Pol and the FitzHugh–Nagumo models of excitable systems. Phys Lett A 2011;375(7):1097–103.
[94] Freire JG, Gallas JAC. Stern-Brocot trees in the periodicity of mixed-mode oscillations. Phys Chem Chem Phys 2011;13(26):12191–8.
[95] de Souza SLT, Lima AA, Caldas IL, Medrano-T RO, Guimarães-Filho ZO. Self-similarities of periodic structures for a discrete model of a two-gene system. Phys Lett A 2012;376(15):1290–4.
[96] Medeiros ES, de Souza SLT, Medrano-T RO, Caldas IL. Replicate periodic windows in the parameter space of driven oscillators. Chaos Solitons Fractals 2011;44(11):982–9.
[97] Medeiros ES, de Souza SLT, Medrano-T RO, Caldas IL. Periodic window arising in the parameter space of an impact oscillator. Phys Lett A 2010;374(26):2628–35.
[98] Medrano-T RO, Rocha R. The negative side of Chua's circuit parameter space: Stability analysis, period-adding, basin of attraction metamorphoses, and experimental investigation. Int J Bifurcation Chaos 2014;24(09):1430025.
[99] Celestino A, Manchein C, Albuquerque HA, Beims MW. Stable structures in parameter space and optimal ratchet transport. Commun Nonlinear Sci Numer Simul 2014;19(1):139–49.
[100] Nicolau NS, Oliveira TM, Hoff A, Albuquerque HA, Manchein C. Tracking multistability in the parameter space of a Chua's circuit model. Eur Phys J B 2019;92(5):106.
[101] Jousseph CAC, Abdulack SA, Manchein C, Beims MW. Hierarchical collapse of regular islands via dissipation. J Phys A 2018;51(10):105101.
[102] Jousseph CA, Kruger TS, Manchein C, Lopes SR, Beims MW. Weak dissipative effects on trajectories from the edge of basins of attraction. Physica A 2016;456:68–74.
[103] Fog A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Copenhagen, Denmark: Technical University of Denmark; 2020.
[104] https://perf.wiki.kernel.org/index.php/Tutorial.
[105] https://github.com/nnagyd/ode_solver_tests/tree/master/Basic_tests_RK4.
[106] https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.
[107] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#instruction-optimization.