Budapest University of Technology and Economics
Faculty of Mechanical Engineering
Department of Hydrodynamics Systems

Massively Parallel GPU-ODE Solver (MPGOS) PerThread v3.0

GPU accelerated integrator for large number of independent ordinary differential equation systems

Ferenc Hegedűs
fhegedus@hds.bme.hu
hegedus.ferenc.82@gmail.com

February 13, 2020

Contents

1 Installation and file hierarchy

Program code MPGOS is written in C++ and in CUDA C software environments. In order to use MPGOS, one needs only to include appropriate source files and prepare a system definition file. All the required source files can be downloaded from the GitHub repository: https://github.com/FerencHegedus/Massively-Parallel-GPU-ODE-Solver. The list of the directories:

- `Doc/`
- `SourceCodes/`
- `Tutorials_SingleSystem_PerThread/`

The directory `Doc/` contains the documentation of the program package. In the directory `SourceCodes/`, one can find all the necessary source codes needs to be included for the usage of the program package. Inside the folder `Tutorials_SingleSystem_PerThread/` there is a collection of tutorial examples help the user getting started with MPGOS. The term `SingleSystem_PerThread/` indicates the parallelisation strategy discussed in more details throughout the manual.

The user has to prepare three files for a given problem:

- `MainCode.cu`
- `SystemDefinition.cuh`
- `makefile` (only for compiling under Linux system)

The name of the first two files can be arbitrary; however, their extension should still be `.cu` and `.cuh`, respectively (indicating that they are CUDA source code and header files). The `MainCode.cu` contains the C++ main function and controls the whole computational process. This is entirely built-up by the user using the objects defined in the library `SourceCodes/`.

Since function pointers cannot be passed as arguments to a GPU kernel function (lambda expressions cannot be used in the present version), the ODE functions (system) and all the other related functions have to be given through pre-declared device (GPU) functions. These functions are collected in the `System Definition.cuh` file. **It must contain all the necessary device function**, which has unique names and must be visible by the program package. The best practice is to take such a file from one of the tutorial example, rename the file to a desired name and rewrite the function bodies. **The function names and the input/output arguments must not be modified**.

The preparation of a makefile is not necessary. However, MPGOS is developed under Linux environment. The file `makefile` is responsible to compile the code corresponding to the tutorial examples by simply type *make* into the command line (as long as the required compilers are installed). The command *make clean* clears the directory from the files generated during the compilation process and the program run (executables and text files). Thus use it with caution. The proposed `makefiles` are very simple examples using only the basics of the makefile environment, the interested user is referred to https://www.gnu.org/software/make/manual/html_node/Introduction.html for more details.

To be able to use MPGOS program package the following inclusions (**in exactly the same order**) have to be made at the beginning of the main code (e.g. the file `MainCode.cu`):

```
#include "Reference_SystemDefinition.cuh"
#include "SingleSystem_PerThread.cuh"
```

The first inclusion is the system definition file mentioned previously. The second header file can be found in the `SourceCodes/` folder. Therefore, during the compilation, the system definition file and the `SourceCodes/` folder must be visible to the compiler. The best practice is to put the system definition file in the same directory where the main code is, and setup a proper path for the `SourceCodes/` folder in the `makefile` (the

first line of the `makefiles` of the tutorial examples do this job). The file `SingleSystem_PerThread.cuh` itself includes many other header files from the `SourceCodes/` folder, thus, the whole content must be visible.

Observe that many parts of the code are actually implemented in header files. This programming style goes against the practice; however, MPGOS exhaustively use C++ **Template Metaprogramming** to produce specialised code according to the requirements. To be able to compile the templated functions, the implementations must be put into header files so that all the templates are visible for the compiler. In addition, in this case, the Nvidia compiler **nvcc** has much more flexibility during the optimization (e.g. function inlining) as effectively there is only one source code file.

Throughout this manual, the basics features and properties of the package is introduced through a reference example that can be found in the first tutorial example (`T1_Reference`).

1.1 Prerequisite for usage

In order to use MPGOS, only the basics of C++ programming is necessary. For those who are new to the C++ language, getting through the following tutorial is advised: http://www.cplusplus.com/doc/tutorial/. In my experience, together with the reference case introduced in this manual, it is enough to get started with MPGOS and make a new problem from scratch.

Although some general advice will be given in Sec. 12 to maximise performance, the interested user is referred to the official Nvidia documentation Programming Guide and Best Practice Guide in https://docs.nvidia.com/cuda/.

1.2 Installation

The program code MPGOS needs no explicit installation. As stated above, in order to use MPGOS, one needs only to include some header files. Nevertheless, the proper installation of a C++11 compatible compiler (**e.g., g++**) and an Nvidia compiler (**nvcc**) are mandatory. Moreover, the proper setup of the **PATH** environment variables are also advisable (e.g., in a *.bashrc* or *.profile* file in Linux). The following listing shows an example for such a set up in a *.bashrc* file for CUDA Toolkit 7.5

```
export CUDA_HOME=/usr/local/cuda-7.5
export LD_LIBRARY_PATH=${CUDA_HOME}/lib64

PATH=${CUDA_HOME}/bin:${PATH}
export PATH
```

where the **CUDA_HOME** folder can be queried via the command *which nvcc* (if a proper Nvidia compiler has been already installed).

For a quick installation guide of the necessary compilers for all platforms, the user is referred to the web page https://docs.nvidia.com/cuda/cuda-quick-start-guide/index.html. For more detailed installation guide for Linux platrom see https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html; and for Windows platform see https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html. These guides are parts of the offcial CUDA documantation: https://docs.nvidia.com/cuda/. In general, on Linux sytem, the user needs admin previliges to be able to install the necessary packages. Otherwise, for more detailed help, please ask your IT manager.

2 Quick start on Linux (or on Windows logged into a Linux system)

Here it is assumed that the user has an access to a Linux machine (either using directly or logged in remotely e.g., from Windows). It is also assumed that **g++** and **nvcc** is already installed, and the **PATH** environment variables have been properly extended, see Sec. 1.2. In this section, a BASH command shell is used. The reference case served to guide the user through the basics of the program package that can be downloaded and run with only a handful of commands. First, the package has to be downloaded from the GitHub repository directly from the site (as a zipped file)

https://github.com/FerencHegedus/Massively-Parallel-GPU-ODE-Solver

and unzip into a directory. Or use the following command if git is already installed:

```
git clone https://github.com/FerencHegedus/Massively-Parallel-GPU-ODE-Solver
```

It downloads the package to the current directory. Next, go to the directory of the first tutorial example in which the files for the reference test case are resides:

```
cd Massively-Parallel-GPU-ODE-Solver/Tutorials_SingleSystem_PerThread/T1_Reference/
```

then compile the source files and finally run the created executable:

```
make
./Reference.exe
```

2.1 Useful tools using windows to login into a Linux machine

In this section, two useful tools are introduced. The first provides a command window to perform the compiling process and run the executable. Personally, I use the SSH client PuTTY, see Fig. 1. After specifying the Host (either via its name or its IP address) and the used port, an SSH command window shall be available where the user can login via his/her username and password.

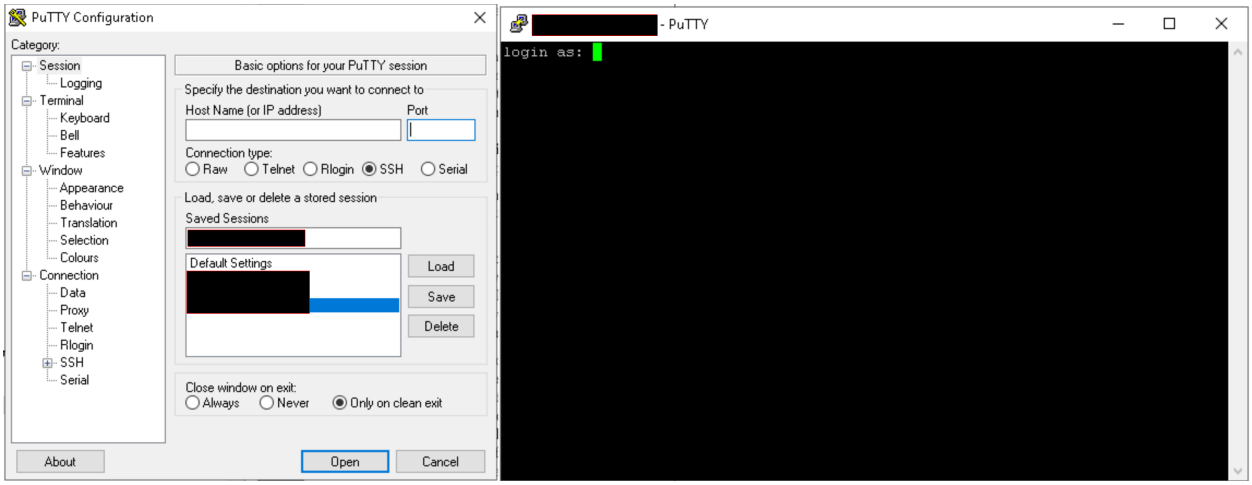

Figure 1: Login with the SSH client PuTTY.

The second tool called winSCP offers an easy way for file transfer and it automatically uploads back the edited source files to the remote machine. The login is very similar as in case of PuTTY, see Fig. 2.

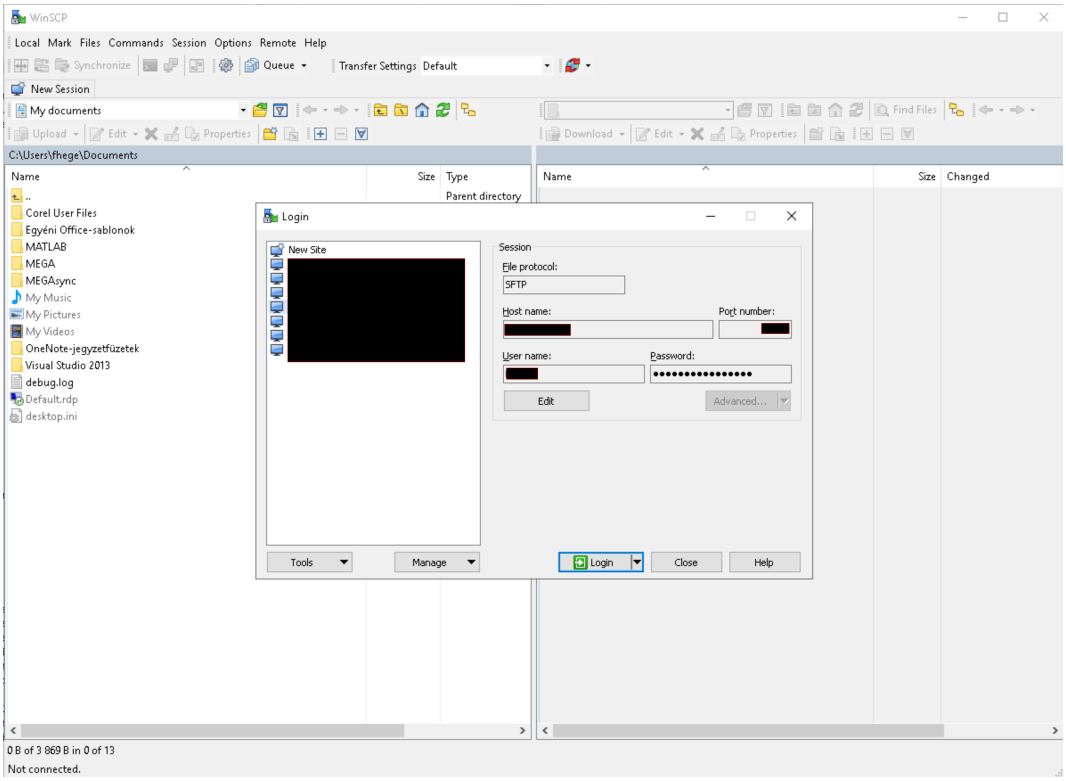

Figure 2: Login with the tool WinSCP for file exchange and editing files.

In the left panel, there is the directory and file structure of the machine from which the user has been logged on (guest windows machine); in the right panel, there is the directory and file structure of the Linux machine to where the user has been logged on (host machine). Under the menu *Option/Preferences...*, in the *Editors* panel, the user can specify and add editors (I prefer **Notepad++**), see Fig. 3. By double clicking on a source file, it will be opened with the preferred editor. By saving it, its new content is automatically updated in the host machine (Linux machine). Thus, in the PuTTY shell the new code can immediately be compiled and run.

3 Quick start on Windows

It must be emphasized that the program package MPGOS is developed and tested continuously under Linux environment. Thus, the smooth operation under Windows cannot be guaranteed. Nevertheless, in this section, a possible technique that was working with the package is introduced.

One possibility to use MPGOS on Windows operating system, if one has a supported version of Microsoft Windows, a supported version of Microsoft Visual Studio, and the NVIDIA CUDA Toolkit installed on your PC. The list of the supported version of the softwares are given in the official CUDA installation guide: https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/. The Microsoft Visual Studio and the CUDA toolkit can be downloaded from https://visualstudio.microsoft.com/downloads/ and from http://developer.nvidia.com/cuda-downloads, respectively. In order to install CUDA toolkit properly, please follow the NVIDIA's installation guide accurately.

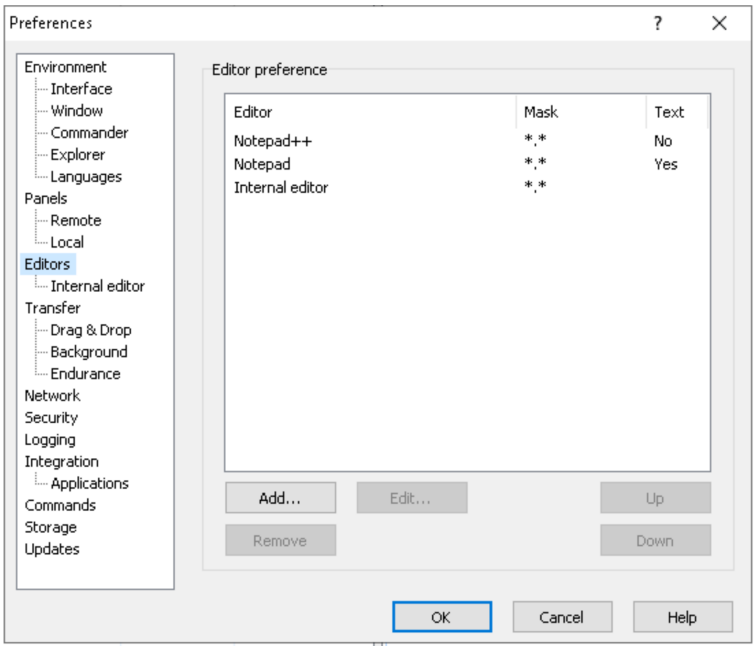

Figure 3: Changing the preferred editor in WinSCP under the menu *Option/Preferences*.

3.1 Running MPGOS in Visual Studio

Here it is assumed that the above software are already installed. In this case, you are able to create a New CUDA project by using Microsoft Visual Studio (MVS). First, run Microsoft Visual Studio and create a new project (File/New/Project or using the hotkeys ctrl+shift+N). In the New Project window (see. Fig. 4), choose the CUDA project. You can name your project and specify its location on your hard drive, then click OK.

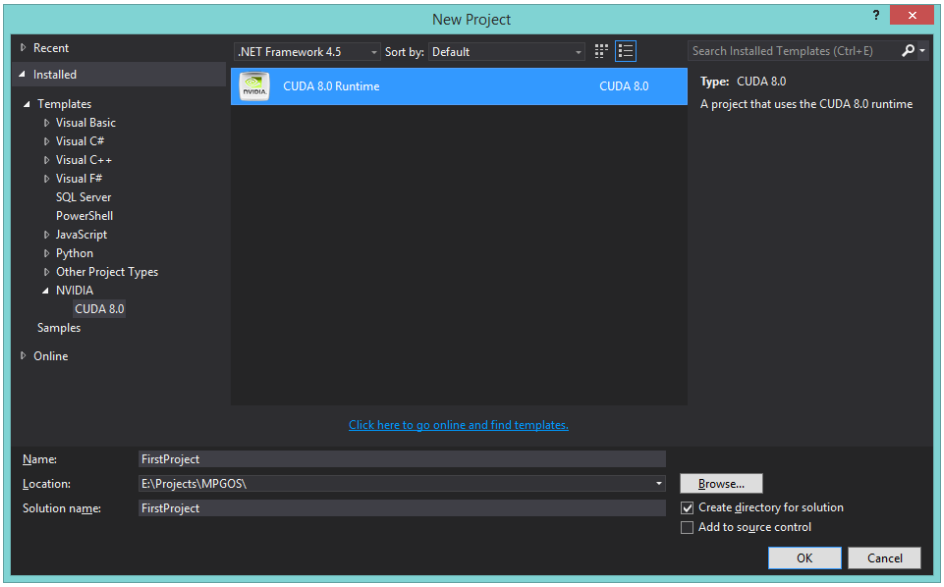

Figure 4: Creating a new CUDA project

Now, Visual Studio sets up the initiated CUDA project and generates a sample source file named kernel.cu (Fig. 5). It is worth mentioning that one may experience errors in this code, e.g.: the command of the

Kernel launch " <<< >>> " is not recognized properly by the code editor. If CUDA toolkit has been installed properly beforehand, as you click on the "Play" button ("Local Windows Debugger"), the source code has to be compiled and run by Visual Studio without any complication. Since this kernel.cu file is a simple default example that comes by creating a new CUDA project, it has to be deleted from the project.

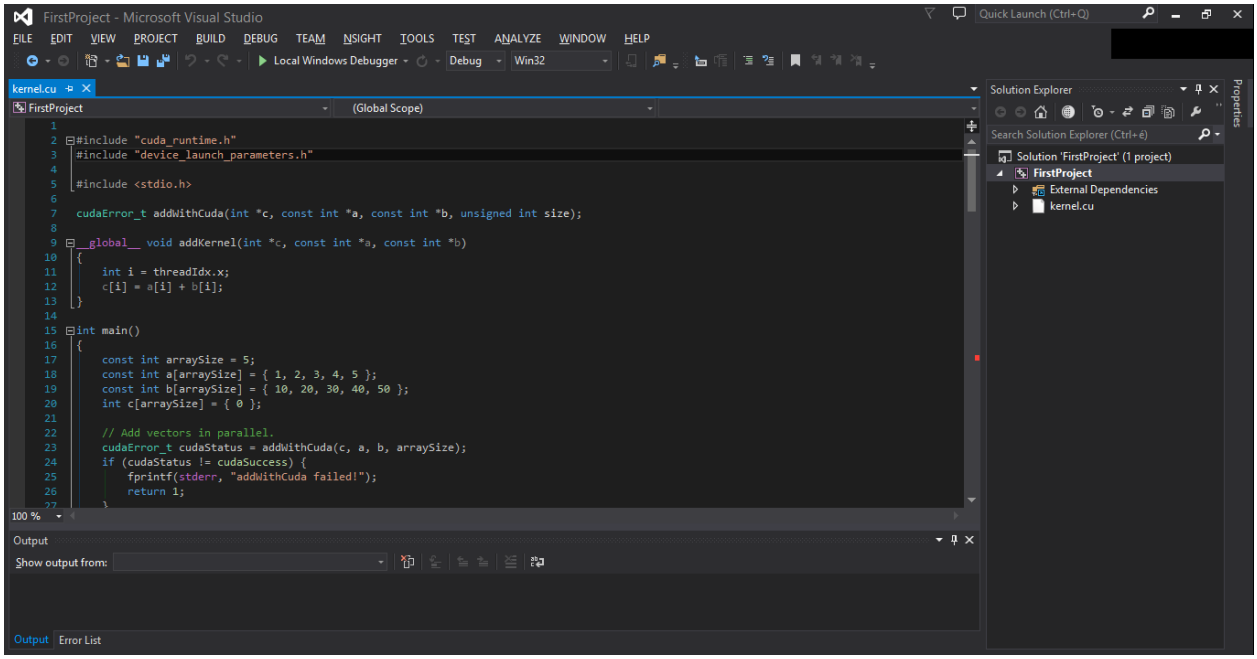

Figure 5: The default CUDA project in the Microsoft Visual Studio

At this point, the MPGOS package has to be downloaded from the GitHub repository: https://github.com/FerencHegedus/Massively-Parallel-GPU-ODE-Solver. By clicking on the "Clone or download" button and choosing "Download Zip", the package can be downloaded into your PC. Unzip, then the program package can be used after make a copy of a system definition file and the directory `SourceCodes/` into the current CUDA project folder. One has to add these source files to the project as well. In Visual Studio, under the "Project" menu, click on "Add Existing Item" or use the hotkeys (Shift+Alt+A), browse your project folder and add the above listed files. Now, an MPGOS project is ready. You will see the MPGOS source files in the Solution Explorer tab, see Fig. 6. You can build and run this project, and it will compute the Duffing equation provided by this manual as a reference example.

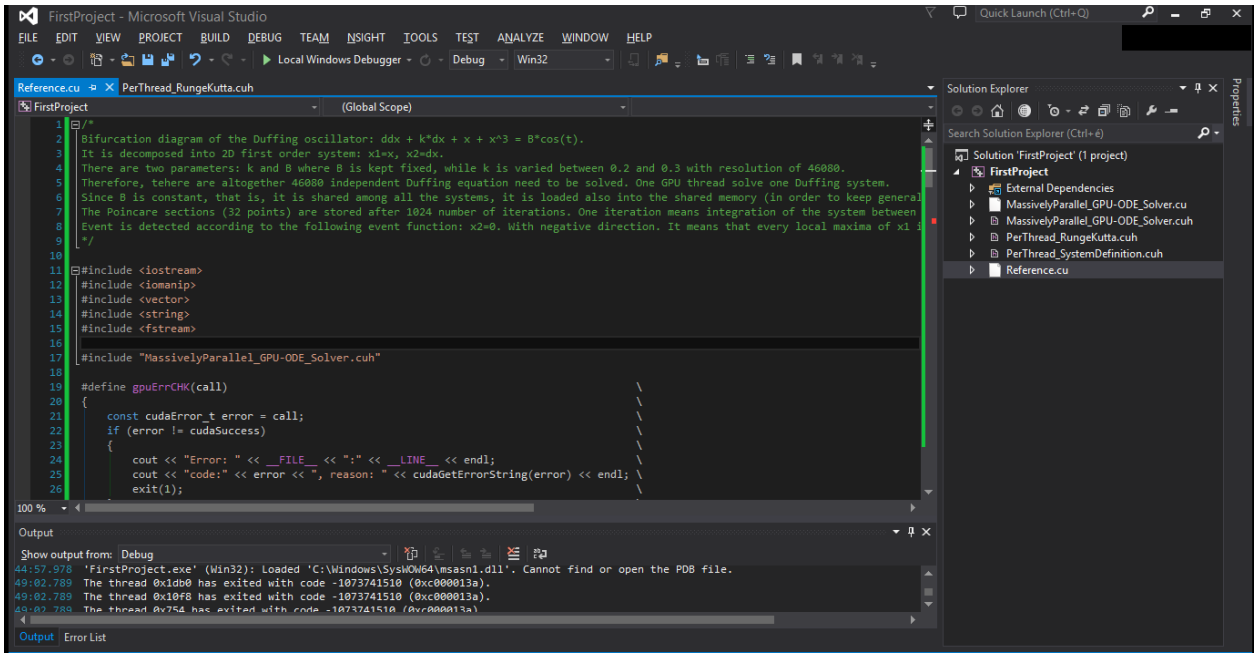

Figure 6: MVS after adding the MPGOS source files

4 Overview of the capabilities

The program is designed to solve (integrate) a large number of independent ordinary differential equation systems of the following form:

$$\dot{\underline{x}} = f(\underline{x}, t; \underline{p}), \tag{1}$$

where $\underline{x}$ is the vector of the state variables, t is the time and $\underline{p}$ is the vector of the parameters. The dot stands for the derivative with respect to time. It is important that the function f must be the same for all simulations; that is, the code **solves many instances of the same system simultaneously** but with different parameter sets and/or initial conditions. This is an essential requirement if one intends to utilize the massively parallel architecture of professional GPUs. Therefore, huge **parameter studies**, examination of **multistability** or the investigation of the **domain of attraction** are among the most suitable situations the MPGOS package can efficiently handle. We shall see in the forthcoming Chapters that **MPGOS in an efficient tool to investigate dynamical systems rather than a plain ODE solver**.

It is important to note that parameters are divided into three subcategories called **control parameters**, **shared parameters** and user programmable parameter called **accessories**. The sets of **control parameters** are different from instance-to-instance. That is, every independent system (instances) have their own sets of control parameters.

The **shared parameters** are common for all instances of the investigated system. They are parameters of Eq. (1); however, their values do not change from instance-to-instance. This is the reason they are called shared parameters (shared among all the instances). Defining common parameters as shared, the required number of slow memory transactions can be significantly reduced, see also Sec. 12. This can be extremely important in memory bandwidth limited applications. There are **two types** of shared parameters: **int** (store common indices) and **double** (store common floating point numbers).

The last kind of parameters are called **accessories** which are multi-purpose (user programmable) parameters. They are not strictly a parameter of Eq. (1) rather than storages that can be updated after every successful time step or after every successful event detection. In this regard, the number of accessories are independent of Eq. (1), and it is absolutely under the control of the user. Accessory variables are very efficient tools to continuously calculate, monitor and store special properties of the solutions/trajectories without the require-

ment for storing a dense output of every instances. This can be crucial for high performance. Throughout the following sections (the first tutorial example) and in the additional tutorial examples in Sec. 13, the efficient use and the capabilities of the accessories are emphasized. There are **two types** accessories as well: **int** (store indices) and **double** (store floating point numbers).

The program package MPGOS offers a framework that completely hides GPU programming from the user. The syntax is kept as simple as possible; for instance, the right-hand side of the ODE system can be given with the same syntax as in MATLAB. It is not required from the user to understand the details of hardware architectures and CUDA programming techniques to quickly assemble a problem. However, for a very complex ODE function, it may be necessary to understand some basics about performance issues, see the corresponding discussion in Sec. 12.

One of the main strength of the package MPGOS is the built-in event detection algorithm. Its usage is very easy and similar to that of available in MATLAB. However, some additional features are incorporated: cooperation with the user programmable accessories, and the possibility to define "action" after a successful event detection to efficiently handle non-smooth dynamics (e.g., impact dynamics, see Sec. 13.4).

Another important feature of the program package is the possibility to manipulate the trajectories of the solutions at the beginning and at the end of every integration phases, after every successful event detection (mentioned above) and after every successful time step. These "actions" can be done on the device (GPU) side immediately during an integration process without terminating the integration. In this way, if a trajectory needs to be manipulated (e.g., in case of an impact), the slowest memory transactions through the PCI-E bus can be avoided. All these actions can be implemented similarly as the right-hand side of the ODE system.

Dense output of the solutions is supported; however, with some limitations. Only a fixed number of time instances can be stored for each trajectory. If the allocated memory if filled, the storing of the points is stopped. Reallocation of global GPU memory have to be avoided at all costs for performance reasons. Although dense output is supported, it must be use with care. One reason using GPUs is to simulate tens of thousands of instances of an ODE system in parallel. Actually, GPUs are less efficient using only a handful of threads. Therefore, the global memory capacity of a GPU can be easily overused even with only a few thousands of stored points for each trajectory. Use dense output only for debugging purposes, and use the user-programmable parameters (accessories) to store a required property of a solution. If dense output is not used, only the endpoints and the accessories of the integration phase is stored and can be accessed. For usage hints and techniques, the reader is referred to the tutorial examples in Sec 13.

The present version of the package supports only explicit Runge–Kutta solvers: the classical 4^{th} order Runge–Kutta scheme, and the adaptive Runge–Kutta–Cash–Karp with embedded error estimation of orders 4 and 5.

As a final remark, the present version of the code can handle **only double precision** floating point arithmetic operations on real numbers. That is, no ordinary float, complex or other types are supported.

5 Important terms and definitions

There is no such term as GPU programming. During code development, the CPU has a major role to organize workload to a GPU or to many GPUs. Therefore, there is always a CPU and GPU programming. In this regard, the GPU can be viewed as a co-processor which handles the most resource intensive parts of a simulation. It has an extremely high computational throughput, but it cannot manage the main control flow of a program. Always the CPU is who tells the GPU what to do and when. Although the present program package hides the details of GPU programming, some important terms and definitions must be clarified for a better understanding before proceeding further. These are summarised in Tab. 1.

Table 1: Summary of the frequently user terms

Term	**Description**
Host	Synonym of CPU
Device	Synonym of GPU
Host Function	An "ordinary" function called from the Host and running on the Host as well.
Kernel Function	A function called from the Host and running on the Device. Declared with the __global__ qualifier.
Device Function	A function called from the Device and running on the Device. Declared with the __device__ qualifier. It can be called from a kernel function or from another device function.
Host Code	Pars of the program running on the CPU
Device Code	Part of the program running on the GPU. Code blocks inside a kernel function or code blocks used in a device function.
System Memory	A memory type visible by the codes running on the CPU. It is the memory plugged into the motherboard and managed by the the operating system.
Global Memory	A memory type visible by the codes running on the GPU. It is the memory that can be found on the graphics card and managed by the Nvidia graphics driver.
Shared Memory	A programmable and fast memory on the device (GPU). It is on-chip; that is, it is near the computational units. Actually, it is a programmable cache.
Registers	The fastest memory type available on the GPU. Every variable declared inside a kernel or a device function are stored here (except arrays or large structures). The compiler also uses them as intermadiate storages during the computations.
built-in `class ProblemSolver`	A built-in class of the program package designed to perform integration on a number of instances of an ODE system. It has an interface to fill-up with valid data of the instances, to manage memory transactions between the CPU and GPU, and to perform the integration itself.

6 Detection and selection of CUDA capable devices

Before using any GPUs as co-processors, the proper selection of a suitable device is mandatory. Nvidia provides a bunch of Application Program Interfaces (APIs) in order to obtain information on a device and to properly select one. To further ease this task, MPGOS offer a few built-in, specialised and simple functions based on the Nvidia APIs. For instance, the function call

```
ListCUDADevices();
```

lists all the CUDA capable devices in a machine and print their most important properties to the screen. In our test PC, the print results can be seen in the listing below. There are two devices: A GeForce GTX TITAN Black (device number 0) and a GeForce GTX 550 Ti (device number 1). This is a very easy way to obtain the serial numbers (device numbers) of the existing devices. It is important since different GPUs can have very different processing powers and capabilities. Moreover, a CUDA code has to be compiled by specifying the architecture of the used GPU. This is characterised by two number: a major revision and a minor revision. In the example below, for instance, the TITAN Black card has a major revision 3 and a minor revision 5, which is indicated by the number 3.5 in the *Compute capability* row (CC 3.5 for short). For computations throughout this manual, the TITAN Black card is used due to its much higher double precision floating point processing power (1707 GFLOPS). Therefore, the source files are compiled by the option `--gpu-architecture=sm_35` to indicate CC 3.5, see the `makefile` in the folders of the tutorial examples. Compiling with this option and selecting a device with a lower compute capability in the code (e.g., the GTX 550 Ti: CC 2.1) the program execution will terminate.

After the inspection of the list of devices, the required GPU can be associated to an instance of a built-in `class` called `ProblemSolver`. It is introduced in Sec. 9. In this way, the proper selection of a device (GPU) is automatically handled transparently by the object of the `ProblemSolver` in every GPU related instruction/operation.

If a GPU is required according to a specific compute capability, it is possible to obtain a device number automatically which has the closest required revision. Then this serial number can be associated to a `ProblemSolver` object. The code snippet

```
int MajorRevision  = 3;
int MinorRevision  = 5;
int SelectedDevice = SelectDeviceByClosestRevision(MajorRevision, MinorRevision);
```

returns a device number having its revision number closest to CC 3.5. In order to check the properties of a specific device, call the following function:

```
PrintPropertiesOfSpecificDevice(SelectedDevice);
```

```
Device number: 0
Device name:   GeForce GTX TITAN Black
-------------------------------------
Total global memory:        6082 Mb
Total shared memory:        48 Kb
Number of 32-bit registers: 65536
Total constant memory:      64 Kb

Number of multiprocessors:  15
Compute capability:         3.5
Core clock rate:            980 MHz
Memory clock rate:          3500 MHz
Memory bus width:           384 bits
Peak memory bandwidth:      336 GB/s

Warp size:                  32
Max. warps per multiproc:   64
Max. threads per multiproc: 2048
Max. threads per block:     1024
Max. block dimensions:      1024 * 1024 * 64
Max. grid dimensions:       2147483647 * 65535 * 65535

Concurrent memory copy:     1
Execution multiple kernels: 1
ECC support turned on:      0


Device number: 1
Device name:   GeForce GTX 550 Ti
-------------------------------------
Total global memory:        963 Mb
Total shared memory:        48 Kb
Number of 32-bit registers: 32768
Total constant memory:      64 Kb

Number of multiprocessors:  4
Compute capability:         2.1
Core clock rate:            1940 MHz
Memory clock rate:          2100 MHz
Memory bus width:           192 bits
Peak memory bandwidth:      100.8 GB/s

Warp size:                  32
Max. warps per multiproc:   48
Max. threads per multiproc: 1536
Max. threads per block:     1024
Max. block dimensions:      1024 * 1024 * 64
Max. grid dimensions:       65535 * 65535 * 65535

Concurrent memory copy:     1
Execution multiple kernels: 1
ECC support turned on:      0
```

7 The Duffing equation as a first tutorial example

Throughout the next few sections, the features and capabilities of the program is introduced by using the well-known Duffing oscillator written in a simplified form as

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

$$\dot{x}_2 = x_1 - x^3 - kx_2 + B\cos(t), \tag{3}$$

which is a second-order ordinary differential equation describing a periodically forced steel beam deflected between two magnets. Observe that the equation is immediately rewritten into a first-order system suitable for numerical integration. Observe also that the **system dimension** is 2. Here k is the damping factor which is the control parameter varied between 0.2 and 0.3 and distributed evenly with a resolution of 46080. That is, altogether 46080 Duffing system will be solved in parallel with different values of k. The amplitude of the harmonic driving is $B = 0.3$. Since this parameter is constant for all systems, it is a perfect example of a shared parameter. In summary, there is 1 **control parameter** and 1 **shared parameter** (of type `double`). For the sake of simplicity, the angular frequency of the excitation is unity. Consequently, the period of the excitation is exactly $T = 2\pi$.

The following tasks shall be accomplished during this first introductory tutorial:

- Storage of the Poincaré sections. This can be done simply by integrating the system over the time domain $t \in [0, T]$ several times, and register the subsequent end points of each integration phase. For a thorough discussion on how to define the right-hand side of the system and how to setup the adaptive time marching, see Secs. 9.1 and 11.1.
- Two events are detected. The first event function is $F_{E1} = x_2$, which detects the states where $x_2 = 0$. As a specialisation, only events with a negative tangent of F_{E1} is detected. Since $\dot{x}_1 = x_2$, this event will detect only the local maxima of x_1 (excluding local minima due to the negative tangent restriction). To show how multiple event functions can be defined, a second event function is prescribed as $F_{E2} = x_1$ without restriction on the direction (tangent of F_{E2}). For more details, see Secs. 9.1 and 11.2.
- Based on the event detections, the local maxima of x_1 and the value of x_2 at the second detected event related to F_{E2} are stored during each integration phases. This needs 2 **accessories** (of type `double`) for each system as additional user programmable storage elements. For the details on how to take advantage of the feature "action" after every successful event detection, see Sec. 11.3.
- The global maxima of each integration phases are also stored. This needs an additional, **third accessory** (of type `double`). The process is discussed in Sec. 11.4.
- Finally, $N_{DO} = 200$ points are stored for each trajectory as a dense output.

The notations used in this manual to characterise the sizes of the different variables of the system are summarized in Tab. 2. These values are the same for each instances of Eq. (1). They are important quantities for the allocation of System Memory and Global Memory.

Table 2: Summary of the size related notations

Notation	Description
N_{SD}	System dimension; that is, the number of the **state variables** x. The length of vector $\underline{x}$ in Eq. (1). In the reference tutorial example (Duffing equation) it is $N_{SD} = 2$.
N_{CP}	Number of the **control parameters**. In the reference tutorial example it is $N_{CP} = 1$.
N_{SP}	Number of the **shared parameters** of type `double`. In the reference tutorial example it is $N_{SP} = 1$.
N_{SPi}	Number of the **shared parameters** of type `int`. In the reference tutorial example it is $N_{SPi} = 0$.
N_{ACC}	Number of the user programmable **accessories** of type `double`. In the reference tutorial example it is $N_{ACC} = 3$. It is under the control of the user independently from the actual system.
N_{ACCi}	Number of the user programmable **accessories** of type `int`. In the reference tutorial example it is $N_{ACCi} = 0$. It is under the control of the user independently from the actual system.
N_{EF}	Number of the event functions. In the reference tutorial example it is $N_{EF} = 2$.
N_P	The total number of problems one intends to solve. Its value is $N_P = 46080$ for the reference case.
N_T	The number of GPU threads during an integration. Its value is $N_T = 23040$ for the reference case. For more details, see Sec. 9.
N_{DO}	The maximum number of points stored for each trajectory is the dense output is activated. Its value is $N_{DO} = 200$ for the reference case.

8 Workflow in a nutshell

The main purpose of using GPUs (having high processing power) is to perform huge parameter studies. Typical situations are when millions of systems at different parameter sets need to be solved. In practice, these systems are not solved one at a time on a GPU. Usually, one creates smaller chunks of problems and integrate only a moderate number of systems on a single GPU. One reason can be the limited amount of global memory or the efficient usage of other GPU resources. However, this moderate number is still in the order of tenth or hundreds of thousands. Another reason to chop the overall number of the problem into smaller chunks is to distribute the workload to different GPUs. Whatever the reason is, objects of the built-in `class ProblemSolver` manage this situation.

Throughout this manual, the `ProblemSolver` is the `class` that defines the behaviour of a corresponding object. And the term SolverObject shall be used to refer to an instance (an object) of the `ProblemSolver`. Its responsibility is to make the necessary memory allocations on both the System Memory (Host side) and Global Memory (Device side), to provide an interface to fill up its data structure with data, to perform the numerical integration on the Device and to copy/synchronise the results between the System Memory (Host side) and the Global Memory (Device side). On a single machine/node there can be more than one instances

of the class `ProblemSolver`. Each such an object can be responsible for computations performed on a specific portion of the overall number of problems and on a specific GPU card, see the schematic draw in Fig. 7. The configuration structure collects all the necessary information for the memory allocations using the quantities summarised in Tab. 2, for details, see also Sec. 9. In our reference tutorial example, there is only one instance of the `ProblemSolver class`.

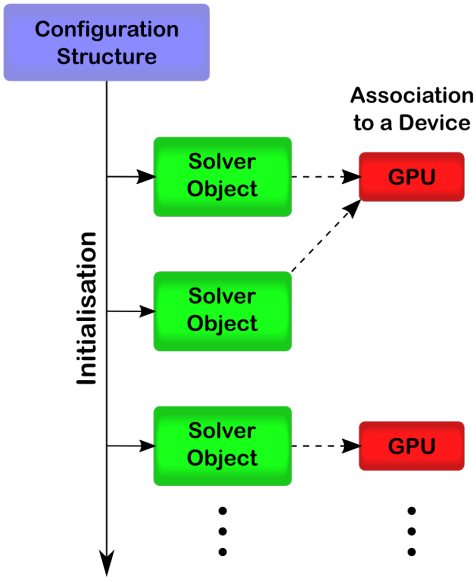

Figure 7: Initialisation of SolverObjects on a machine. The configuration structure collects all the necessary information for the memory allocations.

The computational workflow can be briefly summarised by the following steps using a single solver object and a single GPU:

1. Collect all the necessary information for memory allocations into a configuration data structure.
2. Create an instance (a SolverObject) of the `ProblemSolver`.
3. Configure the SolverObject with its member function `SolverOption()`.
4. Fill the data structure of the SolverObject (time domain, initial conditions ...) with its member function `setHost()`.
5. Synchronise the data from the Host to the Device via the member function `SynchroniseFromHostTo Device()`.
6. Perform integration by calling the member function `Solver()`.
7. Synchronise the data back from the Device to the Host via `SynchroniseFromDeviceToHost()`.
8. Retrieve the necessary data by the `GetHost()` member function.
9. Repeat points 6, 7 and 8) as many time as it is necessary (e.g., to eliminate initial trasients and/or collect data from subsequent integration phases).
10. Repeat phases between points 4 and 8; for instance, to process other portion of problems.

To be able to perform the integration phase, proper implementation of a system is required. For instance, the right-hand side of the ODEs or the event functions for event detection. These functions have to be implemented inside a few pre-declared device functions, which can be found in the system definition file,

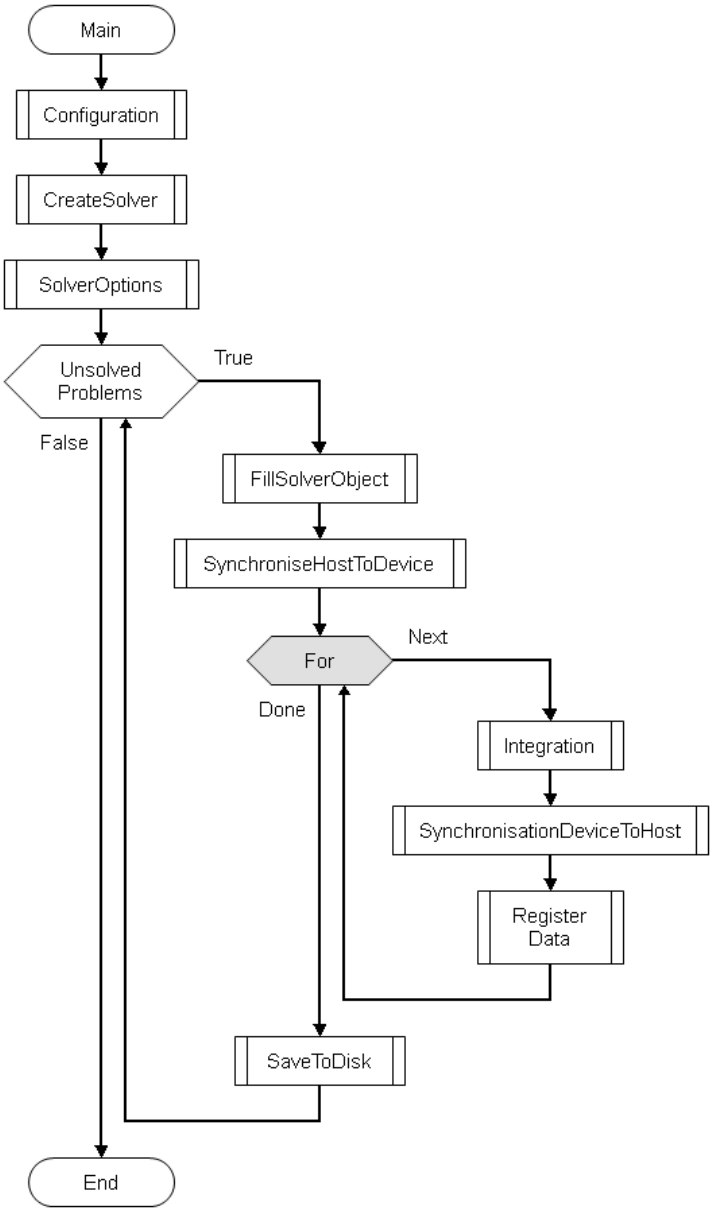

Figure 8: Typical workflow using a single solver object and a single GPU.

e.g., in the `Reference_SystemDefinition.cuh` in the reference tutorial example. The name of these functions should not be changed (the name of the file can be changed but properly included as stated in Sec 1); otherwise, the solver would not find them. This is a necessary inconvenience as function pointers cannot be passed to a kernel function. These pre-declared user functions are very important parts of a project; therefore, a whole section is devoted to their proper description in Sec. 11.

It must be emphasised that the workflow introduced here is used during the main part of this manual through the reference example. That is, as stated above, using a single solver object and a single GPU. Due to such simplicity, this simple example uses functionalities of the SolverObject that cannot overlap CPU and GPU computations, and cannot use multiple GPUs even if they are in a single node/machine. The reason for this is to introduce the main features/ideas of the program package as simply as possible. However, MPGOS offers the aforementioned computation possibilities. Since they need more complicated control logic, they are omitted from the main part of the manual, and these features are introduced via additional tutorial

examples, see e.g., Sec. 13.5 for an example to overlap CPU and GPU computations, or Sec. 13.6 for an example of multi-GPU usage.

9 The Solver Object

In order to initialise a Solver Object, it needs to know information on how much System Memory and Global Memory has to be allocated. This depends on the various size parameters summarized in Tab. 2. Moreover, to be able to produce highly optimised code, these numbers should be passed as template arguments to the Solver Object. In this way, the program package can perform optimisations during compile time specific to certain needs. The consequence is that these values must be known at compile time. Therefore, they are allocated as global `const` variables summarised below

```
// Solver Configuration
#define SOLVER RKCK45 // RK4, RKCK45
const int NT    = 23040; // NumberOfThreads
const int SD    = 2;     // SystemDimension
const int NCP   = 1;     // NumberOfControlParameters
const int NSP   = 1;     // NumberOfSharedParameters
const int NISP  = 0;     // NumberOfIntegerSharedParameters
const int NE    = 2;     // NumberOfEvents
const int NA    = 3;     // NumberOfAccessories
const int NIA   = 0;     // NumberOfIntegerAccessories
const int NDO   = 200;   // NumberOfPointsOfDenseOutput
...
int main()
{
   int NumberOfProblems = 46080; // 2*NT;
   ...
}
```

Observe how the values of the data members agree with the values given also in Tab. 2. The exception is the total number of problems $N_P = 46080$ that is not needed as a template parameter. The Solver Object is responsible on N_T number of problems at a time (here $N_T = 23040$). Therefore, the total number of systems $N_P = 46080$ cannot be solved at once. One needs to deal with the first half of the problems: fill the Solver Object with the corresponding data; perform the necessary integrations, data collections and save operations (first launch). Next, one has to do the same tasks on the second half of the problems by rewriting the data inside the Solver Object (second launch). In general, the required launches for a simulation is the ratio of the problem size and the applied number of threads N_P/N_T (if only one Solver Object is used).

The declaration of an instance of the solver object `ProblemSolver` can be done with the following instruction

```
ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,PREC> ScanDuffing(SelectedDevice);
```

where the instance (Solver Object) of the `class ProblemSolver` is called `ScanDuffing` in our reference case. It is important to note that a proper device must already be associated via the input argument `DeviceNumber`. The reason is that during the initialisation of the Solver Object, memory allocations are already performed on a corresponding Device. In case of multiple GPUs and multiple Solver Objects, the devices must be properly associated to every Solver Object. The list of the devices can be acquired via the function call `ListCUDADevices()`, see the discussion again in Sec. 6.

The parameters between the angled brackets <> are the template parameters, and thus they must be known at compile time. The first 9 template parameters are the ones already given at the first code snippet of this section, and their meaning is given in Tab. 2. The details of the last two template parameters are summarized in Tab. 3. Keep in mind that the order of the template parameters is fixed.

Table 3: Summary of the last two template parameter options of the Solver Object

Template parameters	Options	Description
SOLVER	RK4 RKCK45	The used algorithm. RK4: classic fourth order Runge–Kutta with fixed time step. RKCK45: the adaptive Runge–Kutta–Cash–Karp method with embedded error estimation of orders 4 and 5.
PREC	double	Specify the precision of the used data structure of a computation. In the present version, only double precision is supported (actually, this parameter has no effect).

9.1 Setup of the Solver Object

Each Solver Object can be customised by its member function `SolverOption()`. The sytax is

```
ScanDuffing.SolverOption( Option, Value);
```

or

```
ScanDuffing.SolverOption( Option,  Serial number, Value);
```

depending on the specified Option. The possible Options and their descriptions are summarised in Tab. 4. All options have a default vale. Therefore, an option need to be specified if the default vale need to be overwritten.

Table 4: Summary of the Options in the `SolverOption()` member function of the Solver Object

Option	**Serial number / Value**	**Description**
`ThreadsPerBlock`	Serial number: - Value: `int` (hardware specific limits). Default: warps size (32 in all the present CUDA devices)	The number of threads residing in a block. It can has an impact on the performance, for details see Sec. 12.
`InitialTimeStep`	Serial number: - Value: `double` Default: $1 \cdot 10^{-2}$	The initial time step of the integration phases. This is the employed time step for algorithm with fixed time steps.
`ActiveNumberOf Threads`	Serial number: - Values: `int` between 0 and $N_T - 1$ Default: N_T	Integration for problem number greater then `ActiveNumberOfThreads` will not be performed. It is important if the total number of problems N_P is not an integer multiple of the number of the threads N_T. In this case, before the last simulation launch, the user can modify this option to specify how many number of threads should perform the integration.
`DenseOutputTimeStep`	Serial number: - Values: `double` Default: $-1 \cdot 10^{-2}$	Any negative or zero value means that the points are registered at every successful time steps. Positive value specifies the time step interval of the sampling. Due to specifics of the floating point representation of this number, make a clear distinction between positive and negative numbers; this avoid the use of zero values! Keep in mind that using a solver algorithm with fix step size bigger than this value, the actual time step size will be the value specified by the `DenseOutputTimeStep` (if it is positive).

Table 5: Continuation of Tab. 4

Option	Serial number / Value	Description
`MaximumTimeStep`	Serial number: - Values: `double` Default: $1 \cdot 10^{6}$	The allowed maximum time step during an integration phase. Used only in adaptive solvers.
`MinimumTimeStep`	Serial number: - Values: `double` Default: $1 \cdot 10^{-12}$	The allowed minimum time step during an integration phase. Used only in adaptive solvers.
`TimeStepGrowLimit`	Serial number: - Values: `double` Default: 5	The allowed maximum growth rate of the time step in case of an accepted step. Used only in adaptive solvers.
`TimeStepShrinkLimit`	Serial number: - Values: `double` Default: 0.1	The allowed minimum shrink rate of the time step in case of a rejected step. Used only in adaptive solvers.
`MaxStepInsideEvent`	Serial number: - Values: positive `int` Default: 50	Specifies after how many time step the integration should stop if the final state of of a trajectory is an equilibrium point that resides on an event detection surface.
`MaximumNumberOfTime Steps`	Serial number: - Values: positive or zero `int` Default: 0	Specifies after how many time step the integration should stop. It can be important for debugging purposes. The value 0 means disabled of such stop condition.
`RelativeTolerance`	Serial number: `int` between 0 and $N_{SD} - 1$ Values: `double` Default: $1 \cdot 10^{-8}$	Relative tolerance for each component of the system.
`AbsoluteTolerance`	Serial number: `int` between 0 and $N_{SD} - 1$ Values: `double` Default: $1 \cdot 10^{-8}$	Absolute tolerance for each component of the system.
`EventTolerance`	Serial number: `int` between 0 and $N_E - 1$ Values: `double` Default: $1 \cdot 10^{-6}$	Absolute tolerance to detect special points by event.
`EventDirection`	Serial number: `int` between 0 and $N_E - 1$ Values: -1, 0, 1 Default: 0	Detection of event with positive (1) or negative (-1) tangent of the event function. The value 0 means detection in both directions.
`EventStopCounter`	Serial number: `int` between 0 and $N_E - 1$ Values: positive or zero `int` Default: 0	After how many event detection the solver should stop. With the serial number, every event function can have different values. The value of 0 means no stop at event detection.

9.2 Set and Get the data structure of the Solver Object

Before performing integration, it is mandatory to properly set the data structure of the Solver Object via its member function `SetHost()`. It has three different kind of argument lists. For `TimeDomain`, `ActualState` (initial condition in case of the first integration phase), `Accessories` (the user-programmable parameter) and `ControlParameters` use the form

```
ScanDuffing.SetHost( ProblemNumber, Variable, SerialNumber, Value);
```

For `DenseTime` (modification of the time instances of the dense output, if necessary) use the form

```
ScanDuffing.SetHost( ProblemNumber, Variable, TimeStep, Value);
```

For `DenseState` (modification of the state variables of the dense output at a specific time instance, again if necessary) use

```
ScanDuffing.SetHost( ProblemNumber, Variable, SerialNumber, TimeStep, Value);
```

Finally, for `SharedParameters` use

```
ScanDuffing.SetHost( Variable, SerialNumber, Value);
```

It is important to emphasize that the member function `SetHost()` modifies the data structure inside the Solver Object only in the Host (CPU) side. To be the modifications visible in the Device (GPU) side, data synchronisation is necessary, see the Sec. 9.3.

In a very similar way, it is possible to retrieve a value of a given data inside the Solver Object. The corresponding member function is `GetHost()` having again different forms according to the type and number of the arguments:

```
Value = ScanDuffing.GetHost( ProblemNumber, Variable, SerialNumber);
Value = ScanDuffing.SetHost( ProblemNumber, Variable, TimeStep);
Value = ScanDuffing.SetHost( ProblemNumber, Variable, SerialNumber, TimeStep);
Value = ScanDuffing.SetHost( Variable, SerialNumber);
```

The only difference is that the `Value` argument become a return value. For the summary and description of the arguments and return values, see Tab. 6.

Table 6: Summary of the arguments and return values of the `SetHost()` and `GetHost()` member functions of the Solver Object

Argument	Value	Description
`ProblemNumber`	`int` between 0 and $N_T - 1$	The serial number of the instance of the ODE system. Keep in mind that a single Solver Object solves N_T number of instances simultaneously.

`Variable`	`TimeDomain`
	`ActualState`
	`ControlParameters`
	`SharedParameters`
	`IntegerSharedParameters`
	`Accessories`
	`IntegerAccessories`
	`DenseTime`
	`DenseState`
		Selects the variable itself that is about to set. This variable determine how many arguments have to be used in the `SetHost()` or `GetHost()` member functions. The proper fill-up of the Solver Object with the first 5 variables is mandatory. The `Accessories` are optional, use only if global initialisation is necessary as they can also be initialised for each integration phases via a pre-declared device function discussed in Sec. 11. The last two options related to the dense output are absolutely optional, and it will be used almost only in `GetHost()` member function.

`SerialNumber`	`TimeDomain`: `int` between 0 and 1, `ActualState`: `int` between 0 and $N_{SD} - 1$, `ControlParameters`: `int` between 0 and $N_{CP} - 1$, `SharedParameters`: `int` between 0 and $N_{SP} - 1$, `IntegerSharedParameters`: `int` between 0 and $N_{SPi} - 1$, `Accessories`: `int` between 0 and $N_{ACC} - 1$, `IntegerAccessories`: `int` between 0 and $N_{ACCi} - 1$, `DenseState`: `int` between 0 and $N_{SD} - 1$	The serial number of a given variable. For instance, the serial number of t_1 (the upper limit of the time domain) is 1. Similarly, the serial number of the second state variable x_2 is 1, and the serial number of the third control parameter is 2. Keep in mind that in C++, the indexing starts from zero.
`TimeStep`	`int` between 0 and $N_{DO} - 1$	In case of dense output, it is the serial number of the time step.
`Value`	`double`	The value indent to be set by the `SetHost()` member function, or the return value of the `GetHost()` member function.

9.3 Synchronise data between the Host and the Device

Before the integration process, the data in the properly filled Solver Object must be synchronised from the Host to the Device. Similarly, after performing the required number of integration phases, the resulted data must be synchronised back from the Device to the Host. So that it can be accessible by the `GetHost()` member function. The member functions responsible for the synchronisation processes are

```
ScanDuffing.SynchroniseFromHostToDevice( Variable );
ScanDuffing.SynchroniseFromDeviceToHost( Variable );
```

The summary and the description of the argument list is given in Tab. 7. Both member functions are asynchronous with respect to the CPU; that is, after the function call, the control is immediately get back

to the CPU to perform other instructions regardless of whether the memory transactions between the CPU and GPU via the PCI-E bus is completed or not. For details see Secs. 9.4 and 10.

Table 7: Summary of the arguments and return values of the `SynchroniseFromHostToDevice()` and `SynchroniseFromDeviceToHost()` member functions of the Solver Object

Argument	Value	Description

`Variable`	`TimeDomain`
	`ActualState`
	`ControlParameters`
	`SharedParameters`
	`IntegerSharedParameters`
	`Accessories`
	`IntegerAccessories`
	`DenseOutput`
	`All`
		Selects the variable to synchronise. The option `DenseOutput` synchronises both the time instances and state variables of the dense output. The option `All` synchronises everything.

9.4 Perform integration and synchronise CPU and GPU operations

After the proper data management operations, the integration can be performed simply by calling the member function

```
ScanDuffing.Solve();
```

which integrates every instances of the ODE system Eq. (1) simultaneously between their time domain t_0 and t_1. It is important to emphasize again, that each instance has its own time domain; that is, they can march with different time steps.

The above C++ command initiates the integration process on the Device (GPU) and returns the control immediately back to the Host (CPU). Therefore, the CPU is free to use any other instructions immediately in parallel with the already running GPU computations. Such a behaviour is called asynchronous operation with respect to the GPU. Every GPU related action in the program package MPGOS is asynchronous by default. This feature of the CUDA architecture is necessary to be able to overlap CPU-GPU computations, or simply to dispatch computations to different GPUs using multiple Solver Objects. If the CPU would wait for the GPU to finish its work after the `Solve()` instruction, it could not be initiate a new `Solve()` instruction on another Solver Object. Whatever the situation is, at some point, synchronisation of the program is necessary between the CPU thread and the GPU computations.

To perform CPU-GPU synchronisation, one has to insert a synchronisation point and perform the synchronisation as follows:

```
ScanDuffing.Solve();
ScanDuffing.InsertSynchronisationPoint();
ScanDuffing.SynchroniseSolver();
```

In the above code snippet, after calling the `Solve()` member function, the CPU immediately proceed further and insert a synchronisation point in the GPU computations (in the GPU working queue). The next line forces the CPU to wait until the GPU computations in the corresponding Solver Object reaches the inserted synchronisation point. That is, the CPU thread is synchronised with respect to the Solver Object. Such a very simple combination of function calls works perfectly using a single GPU and a single Solver Object (the present reference case), for more complicated synchronisation patterns the reader is referred to the tutorial examples in Sec. 13.5 and 13.6.

The last option

```
ScanDuffing.SynchroniseDevice();
```

synchronises the CPU with respect to a Device the calling Solver Object is assigned to. That is, even if there are multiple Solver Objects assigned to the same GPU, the CPU will wait at this synchronisation point until all the computations are finished in the particular device (regardless of which Solver Object was the caller).

9.5 Print the content of the Solver Object

With the `Print()` member functions the content of the data structure of the Solver Object can be printed into file(s). According to the number of the arguments it has two forms:

```
ScanDuffing.Print( Variable );
ScanDuffing.Print( Variable, ProblemNumber );
```

The summary and the description of the argument list is given in Tab. 8.

Table 8: Summary of the arguments of the `Print()` member function of the Solver Object

Argument	Value	Description

`Variable`	`TimeDomain`
	`ActualState`
	`ControlParameters`
	`SharedParameters`
	`IntegerSharedParameters`
	`Accessories`
	`IntegerAccessories`
	`DenseOutput`
		Selects the variable to print the variable into a file. The name of the text file is *.txt where the asterisk is the name of the `Variable` (except the `DenseOutput`, see below). The columns represent the serial number of the actual `Variable`, while the rows represent the `ProblemNumber` (the serial number) of the instance of the ODE system. This feature is useful for debugging purposes to check the content of the Solver Object.

`ProblemNumber`	`int` between 0 and $N_T - 1$	The serial number of the instance of the ODE system whose data is written into a file. Keep in mind that a single Solver Object solves N_T number of instances simultaneously. The name of the file is tid_*.txt, where the asterisk is the `ProblemNumber`. In the first 10 lines the `ControlParameters`, `SharedParameters` and `Accessories` are printed with together with text lines for explanations. From line 11, the data corresponding to the particular problem is written. The columns are the `TimeDomain` and then the components of the `ActualState`.

10 The complete reference case

The code snippet provided in this section illustrates the `main()` function of the reference case. We will examine the code thoroughly, step-by-step.

- First, some libraries from the C++ Standard Template Library is included to be able to use; for instance, the `string` or `vector` containers.
- Next, the required header files of the program package MPGOS is included, see also Sec. 1. It is necessary to use the program package. Keep in mind again that the order of the inclusion is strict. Observe that the system definition file has the name `Reference_SystemDefinition.cuh` here, which is arbitrary. The name of the second header file is pre-defined.
- The irrational number π is defined as `PI`. Next, the line `using namespace std` allows us to use the elements of the Standard Template Libraries without the scope operator `std::`. The next define statement serves as a simplified modification of the algorithm template parameter (now RKCK45). Finally, size-related template parameters are declared as constant variables.
- The next three lines define the prototypes of auxiliary functions that are not the part of the program package. However, they make the code more readable.
- Inside the `main()` function, the total number of problems and the block size during the integration is defined. The control parameter is the damping variable of the Duffing equation k divided uniformly between 0.2 and 0.3 with a resolution of $N_P = 46080$, see again Sec. 7. Since the number of threads in the Solver Object is $N_T = 23040$, two simulation launches will be necessary to process all the parameters ($N_P/N_T = 2$), see also the details below. The block size can be an important parameter for performance. Here it is important to not that it should be an integer multiple of 32 (warp size), for the details see Sec 12. If the user have no solid confidence, use the default value.
- The next few lines display all the CUDA capable devices in the used machine, select a device according to a revision number (3.5 :major 3 and minor 5) and store it to the variable `SelectedDevice` for further use, and finally, lists the properties of the selected device.
- The initial conditions are fixed with $x_1 = -0.5$ and $x_2 = -0.1$, and the excitation amplitude B is set to 0.3 (it is a shared parameter).
- Next, the values of the control parameter k is generated and stored in the variable `Parameters_k_Values` using the auxiliary `Linspace()` function (see its definition in the bottom of the code snippet). Its type is the `vector` container of the Standard Template Library.
- Only one Solver Object is created called `ScanDuffing` using the selected device `SelectedDevice`. For demonstration purposes, all the possible solver options are modified via the `SolverOption` member function.
- The number of the required simulation launches is determined by dividing the total number of problems N_P (total number of instances of the ODE system) and the used threads in the Solver Object N_T. If there is a remainder during the division, the number of the simulation launches is increased by 1 to process the remaining problems. In such cases, do not forget to adjust the `ActiveNumberOfThreads` to the remainder; this is not the case in the reference example.
- The next few lines setup the data file for saving the results, and declare some variables for runtime measurements.
- A single `for` cycle is created to loop through the number of simulation launches (now it is $N_P/N_T = 2$).
- The first step for each simulation launch is to fill the Solver Object with valid data. For this purpose, an auxiliary function is created called `FillSolverObject`. Again, it is not the part of the program package MPGOS, it is implemented only for this specific reference problem. Its implementation details

is discussed below. What it is important here is that data structure is filled up according to the launch counter. First, only the first half of the values of the damping parameters k is used. In the second loop, the other half is used.

- With the member function `SynchroniseFromHostToDevice()`, all the data is copied from the Host to the Device. It is also an asynchronous operation similarly to the `Solve()` member function. That is, after this line, the control is immediately passed back to the CPU and continue the work. Therefore, in order to measure only the runtime of the transients on the GPU and exclude the time necessary to copy data to the Global Memory of the device, synchronisation point is inserted, and the device is synchronised with the CPU. In this way, it can be ensured that the CPU does not start the time measurement before copying all the required data.

- The first 1024 number of integration phases are regarded as initial transients and discarded. Since the Duffing system, the reference case, is periodic in time with period 2π, it is not necessary to modify the data structure in the Solver Object to continue the integration phases if the time domain is set to $t_0 = 0$ and $t_1 = 2\pi$ for every instances of the ODE system. This is exactly the case here, see also the discussion of the problem filling below. Observe that how a synchronisation point is inserted after the call to the `Solve()` member function, and the GPU is synchronised inside the loop of the transient simulation. Parenthetically, it is not necessary to include synchronisation after copying the data from the Host to the Device via `SynchroniseFromHostToDevice()`. The GPU has a linear working queue; thus, the `Solve()` instruction will start only after finishing the previous task in the queue that was the data synchronisation, even though the CPU issued these task to the GPU with almost no delay. In this sense, it would be necessary only to insert a synchronisation point after the transient loop. However, in such a situation, the CPU will issue immediately 1024 number of task to the GPU working queue. To avoid overflow of the GPU working queue, it is a good practice to insert synchronisation after few issued tasks whenever it is feasible.

- For each transient iterations, the total computation time is printed to the screen (more precisely, to the standard output). Without the synchronisation point inside the transient loop, the measured CPU time of the transients would be orders of magnitude smaller (only the time to issue the 1024 number of tasks would be measured, not the time to complete it).

- The next 32 number of integration phases, organised in a loop, is served to save the data of the converged trajectories into a file in each phases. This is done via the auxiliary function called `SaveData()`, again this is not the part of the program package and for the details see the discussion below. Observe that the synchronisation back to the device is called with the `All` input argument. It is simple but it is less efficient; for instance, the `SharedParameters` and the `ControlParameters` do not need to be copied back to the Host (they simply do not change). In the tutorial example, we left the argument to `All` for simplicity. However, the user can decide which variable has to be really synchronised.

- Finally, the total simulation time is printed to the standard output, and the used file is closed.

- For testing and debugging purposes, the dense output of the problem numbers 0, 4789, and 14479 of the final simulation launch are printed into a file.

- Auxiliary function `Linspace()`: it do exactly what its name suggests, it creates a uniformly distributed values between B (begin) and E (end) with number of points N. Observe that how the `vector()` x is passed by reference. That is, x here is an alias of `Parameters_k_Values` meaning that if some values in x changes in this function, so does in `Parameters_k_Values`.

- Auxiliary function `FillSolverObject()`: it fills the Solver Object with data. Observe that the difference between the variables `k_begin` and `k_end` is exactly the number of the used threads N_T in the Solver Object. The loop fills the data corresponding to the `TimeDomain` (fixed for each thread/instance), `ActualState` (initial condition, fixed also for each threads), `ControlParameters` (different from instance-to-instance, this is the damping parameter k with different values distributed between 0.2 and 0.3), and finally the `Accessories` (initialised by zero). Observe that the `SharedParameters` have to be set only once (outside the loop). In this reference case there is only one shared parameter (excitation amplitude B); that is, there is only one corresponding function call.

- Auxiliary function `SaveData()`: it save the required data to the specified file. It takes the solver Object as a reference for performance reasons. Using the `GetHost()` member function, the sole `ControlParameters` and the `SharedParameters` are written into the first two column. In the next two columns, the endpoints of the integration phases are stored (`Actualstates`) that is also the Poincaré section of the system. Finally, in the last three column, the three `Accessories` are stored, see also Sec. 7 for details.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <fstream>

#include "Reference_SystemDefinition.cuh"
#include "SingleSystem_PerThread.cuh"

#define PI 3.14159265358979323846

using namespace std;

// Solver Configuration
#define SOLVER RKCK45 // RK4, RKCK45
const int NT    = 23040; // NumberOfThreads
const int SD    = 2;     // SystemDimension
const int NCP   = 1;     // NumberOfControlParameters
const int NSP   = 1;     // NumberOfSharedParameters
const int NISP  = 0;     // NumberOfIntegerSharedParameters
const int NE    = 2;     // NumberOfEvents
const int NA    = 3;     // NumberOfAccessories
const int NIA   = 0;     // NumberOfIntegerAccessories
const int NDO   = 200;   // NumberOfPointsOfDenseOutput

void Linspace(vector<double>&, double, double, int);
void FillSolverObject(ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double>&,
    const vector<double>&, double, double, double, int, int);
void SaveData(ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double>&, ofstream&,
    int);

int main()
{
// INITIAL SETUP ---------------------------------------------------------------------

   int NumberOfProblems = 46080; // 2*NT;
   int BlockSize        = 64;

   ListCUDADevices();

   int MajorRevision  = 3;
   int MinorRevision  = 5;
   int SelectedDevice = SelectDeviceByClosestRevision(MajorRevision, MinorRevision);

   PrintPropertiesOfSpecificDevice(SelectedDevice);


   double InitialConditions_X1 = -0.5;
   double InitialConditions_X2 = -0.1;
   double Parameters_B = 0.3;

   int NumberOfParameters_k = NumberOfProblems;
   double kRangeLower = 0.2;
    double kRangeUpper = 0.3;
      vector<double> Parameters_k_Values(NumberOfParameters_k,0);
      Linspace(Parameters_k_Values, kRangeLower, kRangeUpper, NumberOfParameters_k);
```

```
    ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double> ScanDuffing(
        SelectedDevice);

    ScanDuffing.SolverOption(ThreadsPerBlock, BlockSize);
    ScanDuffing.SolverOption(InitialTimeStep, 1e-2);
    ScanDuffing.SolverOption(ActiveNumberOfThreads, NT);

    ScanDuffing.SolverOption(DenseOutputTimeStep, -1e-2);

    ScanDuffing.SolverOption(MaximumTimeStep, 1e3);
    ScanDuffing.SolverOption(MinimumTimeStep, 1e-14);
    ScanDuffing.SolverOption(TimeStepGrowLimit, 10.0);
    ScanDuffing.SolverOption(TimeStepShrinkLimit, 0.2);
    ScanDuffing.SolverOption(MaxStepInsideEvent, 50);
    ScanDuffing.SolverOption(MaximumNumberOfTimeSteps, 0);

    ScanDuffing.SolverOption(RelativeTolerance, 0, 1e-9);
    ScanDuffing.SolverOption(RelativeTolerance, 1, 1e-9);
    ScanDuffing.SolverOption(AbsoluteTolerance, 0, 1e-9);
    ScanDuffing.SolverOption(AbsoluteTolerance, 1, 1e-9);

    ScanDuffing.SolverOption(EventTolerance, 0, 1e-6);
    ScanDuffing.SolverOption(EventTolerance, 1, 1e-6);
    ScanDuffing.SolverOption(EventDirection,   0, -1);
    ScanDuffing.SolverOption(EventDirection,   1,  0);
    ScanDuffing.SolverOption(EventStopCounter, 0,  0);
    ScanDuffing.SolverOption(EventStopCounter, 1,  0);

// SIMULATIONS ------------------------------------------------------------------

    int NumberOfSimulationLaunches = NumberOfProblems / NT + (NumberOfProblems % NT == 0 ?
         0:1);

    ofstream DataFile;
    DataFile.open ( "Duffing.txt" );

    clock_t SimulationStart = clock();
    clock_t TransientStart;
    clock_t TransientEnd;

    for (int LaunchCounter=0; LaunchCounter<NumberOfSimulationLaunches; LaunchCounter++)
    {
       FillSolverObject(ScanDuffing, Parameters_k_Values, Parameters_B,
           InitialConditions_X1, InitialConditions_X2, LaunchCounter * NT, NT);

       ScanDuffing.SynchroniseFromHostToDevice(All);
       ScanDuffing.InsertSynchronisationPoint();
       ScanDuffing.SynchroniseSolver();

       TransientStart = clock();
       for (int i=0; i<1024; i++)
       {
          ScanDuffing.Solve();
          ScanDuffing.InsertSynchronisationPoint();
          ScanDuffing.SynchroniseSolver();
       }
       TransientEnd = clock();
          cout << "Launches: " << LaunchCounter << "  Simulation time: " << 1000.0*(
              TransientEnd-TransientStart) / CLOCKS_PER_SEC << "ms" << endl << endl;

       for (int i=0; i<32; i++)
       {
          ScanDuffing.Solve();
          ScanDuffing.SynchroniseFromDeviceToHost(All);
          ScanDuffing.InsertSynchronisationPoint();
          ScanDuffing.SynchroniseSolver();

          SaveData(ScanDuffing, DataFile, NT);
```

```
      }
   }

   clock_t SimulationEnd = clock();
      cout << "Total simulation time: " << 1000.0*(SimulationEnd-SimulationStart) /
          CLOCKS_PER_SEC << "ms" << endl << endl;

   DataFile.close();

   ScanDuffing.Print(DenseOutput, 0);
   ScanDuffing.Print(DenseOutput, 4789);
   ScanDuffing.Print(DenseOutput, 15479);

   cout << "Test finished!" << endl;
}

// AUXILIARY FUNCTION -------------------------------------------------------------

void Linspace(vector<double>& x, double B, double E, int N)
{
    double Increment;

   x[0]   = B;

   if ( N>1 )
   {
      x[N-1] = E;
      Increment = (E-B)/(N-1);

      for (int i=1; i<N-1; i++)
      {
         x[i] = B + i*Increment;
      }
   }
}

void FillSolverObject(ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double>&
    Solver, const vector<double>& k_Values, double B, double X10, double X20, int
    FirstProblemNumber, int NumberOfThreads)
{
   int k_begin = FirstProblemNumber;
   int k_end   = FirstProblemNumber + NumberOfThreads;

   int ProblemNumber = 0;
   for (int k=k_begin; k<k_end; k++)
   {
      Solver.SetHost(ProblemNumber, TimeDomain,  0, 0 );
      Solver.SetHost(ProblemNumber, TimeDomain,  1, 2*PI );

      Solver.SetHost(ProblemNumber, ActualState, 0, X10 );
      Solver.SetHost(ProblemNumber, ActualState, 1, X20 );

      Solver.SetHost(ProblemNumber, ControlParameters, 0, k_Values[k] );

      Solver.SetHost(ProblemNumber, Accessories, 0, 0 );
      Solver.SetHost(ProblemNumber, Accessories, 1, 0 );
      Solver.SetHost(ProblemNumber, Accessories, 2, 0 );

      ProblemNumber++;
   }

   Solver.SetHost(SharedParameters, 0, B );
}

void SaveData(ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double>& Solver,
    ofstream& DataFile, int NumberOfThreads)
{
   int Width = 18;
```

```
    DataFile.precision(10);
    DataFile.flags(ios::scientific);

    for (int tid=0; tid<NumberOfThreads; tid++)
    {
        DataFile.width(Width); DataFile << Solver.GetHost(tid, ControlParameters, 0) <<',';
        DataFile.width(Width); DataFile << Solver.GetHost(SharedParameters, 0) << ',';
        DataFile.width(Width); DataFile << Solver.GetHost(tid, ActualState, 0) << ',';
        DataFile.width(Width); DataFile << Solver.GetHost(tid, ActualState, 1) << ',';
        DataFile.width(Width); DataFile << Solver.GetHost(tid, Accessories, 0) << ',';
        DataFile.width(Width); DataFile << Solver.GetHost(tid, Accessories, 1) << ',';
        DataFile.width(Width); DataFile << Solver.GetHost(tid, Accessories, 2);
        DataFile << '\n';
    }
}
```

It can be seen that the main strength of the program package is that one needs only a three pages long code block to perform a detailed parameter study of a dynamical system. The resulted bifurcation diagram is shown in Fig. 9, where the first component of the Poincaré section $P(x_1)$ is plotted as a function of the control parameter k. The total simulation time (including the writing of the data into file) of the above-described `main()` function is merely 11.7 s on an Nvidia Titan Black card (1707 GFLOPS theoretical peak processing power). To measure the performance related to pure the computations, the trasient part of the simulation took only 4.9 s.

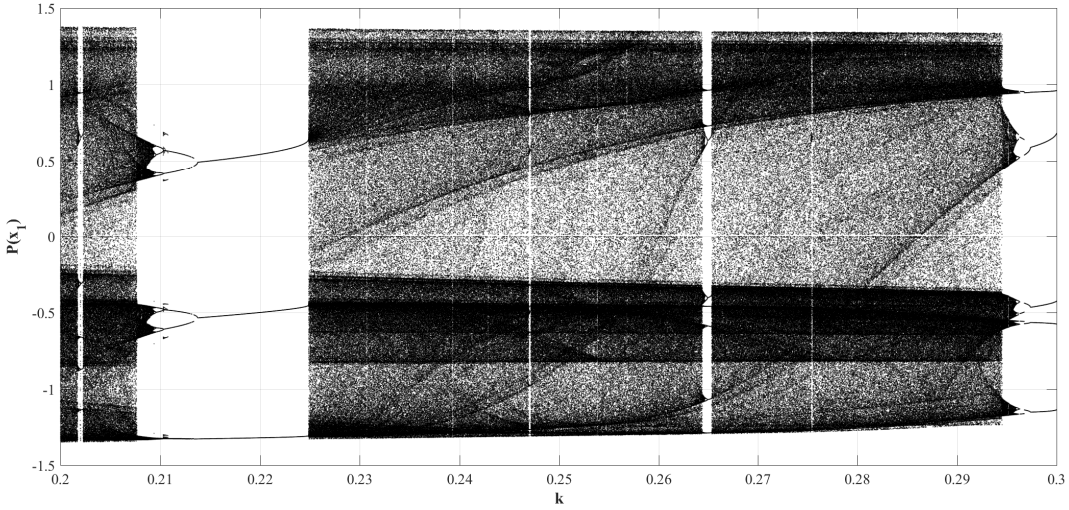

Figure 9: The bifurcation diagram of the Duffing oscillator; that is, the first component of the Poincaré section $P(x_1)$ as a function of the control parameter k. The control parameter is linearly distributed between 0.2 and 0.3 with a resolution of 46080.

11 Details of the pre-declared device functions

The last step to set-up a problem is to define at least the right-hand side of the ODE system. We shall see that there are similar functions that have an effect on the behaviour of the integrator (e.g., the event function). This section dedicated to describe them. To fully grasp the ideas behind these functions, the internal structure of the `Solver()` member function has to be discussed in more details. Figure 10 shows such a workflow for adaptive solvers with event handling. This is the most complex flow structure; using solvers with fixed time steps or without event handling, the structure is simpler. The workflow is as follows

- First, variables shared by all the instances of the ODE system is loaded into the shared memory (fast, user-programmable cache) of the GPU. These are the `SharedParameters()` given by the user and the tolerance related solver options.
- Next, during an initialisation procedure, a special, `Initialisation()` pre-declared device function is called. This can be implemented similarly as the right-hand side of the ODE system. Its purpose is to initialise the integration process. The user can access every variable inside this function, therefore even the trajectories can be manipulated at the beginning of each integration phases, if necessary. Basically, any kind of control flow can be implemented here according to requirements.
- After that, the solver checks whether the integration is finished or not. The integration is stopped if the time reaches the end of the time domain ($t = t_1$) or if an event is triggered with a stop condition. If the integration is not finished then time stepping is performed.
- Inside a time stepping, the first task is to perform the actual step and estimate the error (in case of adaptive solver). If the step is rejected, a new time step is estimated and the stepping is repeated with the new smaller time step. If it is accepted, a series of instructions are performed discussed below. Using solvers with fixed time steps, the step is always accepted. This phase successively calls the right-hand side of the ODE system called `OdeFunction()` that is also a pre-declared device function. The right-hand side can be implemented here very similarly as in case of MATLAB.
- Even if the time step is accepted, a new time step is estimated from the local truncated error in case of the adaptive solvers.
- In solvers with fixed time step, the time step estimation is completely ignored.
- After that, the event function is evaluated via the pre-declared device function `EventFunction()`. Again, it can be given very similarly as in case of MATLAB.
- Next, the solver checks for event detection. If an event is not detected, the time step might be adjusted to detect the event with the prescribed absolute tolerance. The adjustment depends on the value of the estimated time step from the local error and the predicted one for a precise event detection.
- If an event is detected, a special pre-declared device function `ActionAfterEventDetection()` is called. Inside this function, the user can implement any kind of control logic to manipulate the trajectories or store properties of the solution into the user-programmable parameters, into the `Accessories()`.
- All the processes related to the event detection is omitted if a solver is set-up with a zero number of events.
- The next step is to update the state of the trajectory. In addition, if number of the dense output is not zero, the new state is also stored in the corresponding data structure.
- Nearly at the end of the accepted branch, another special, pre-declared device function `ActionAfter SuccessfulTimeStep()` is called. Again, its control flow is absolutely under the control of the user. With this function, the trajectory can be manipulated or special properties can be stored after every successful time step.
- As a final part of the time stepping procedure, the time step is updated according to the estimated one.
- Observe that pre-declared device function `ActionAfterSuccessfulTimeStep()` is called **after the update of the state but before the update of the time step**.
- If the integration is finished, the last type of specialised pre-declared device function `Finalisation()` is called. Again, it is a user-manageable function that can manipulate the trajectory or store special properties at the end of an integration phase.

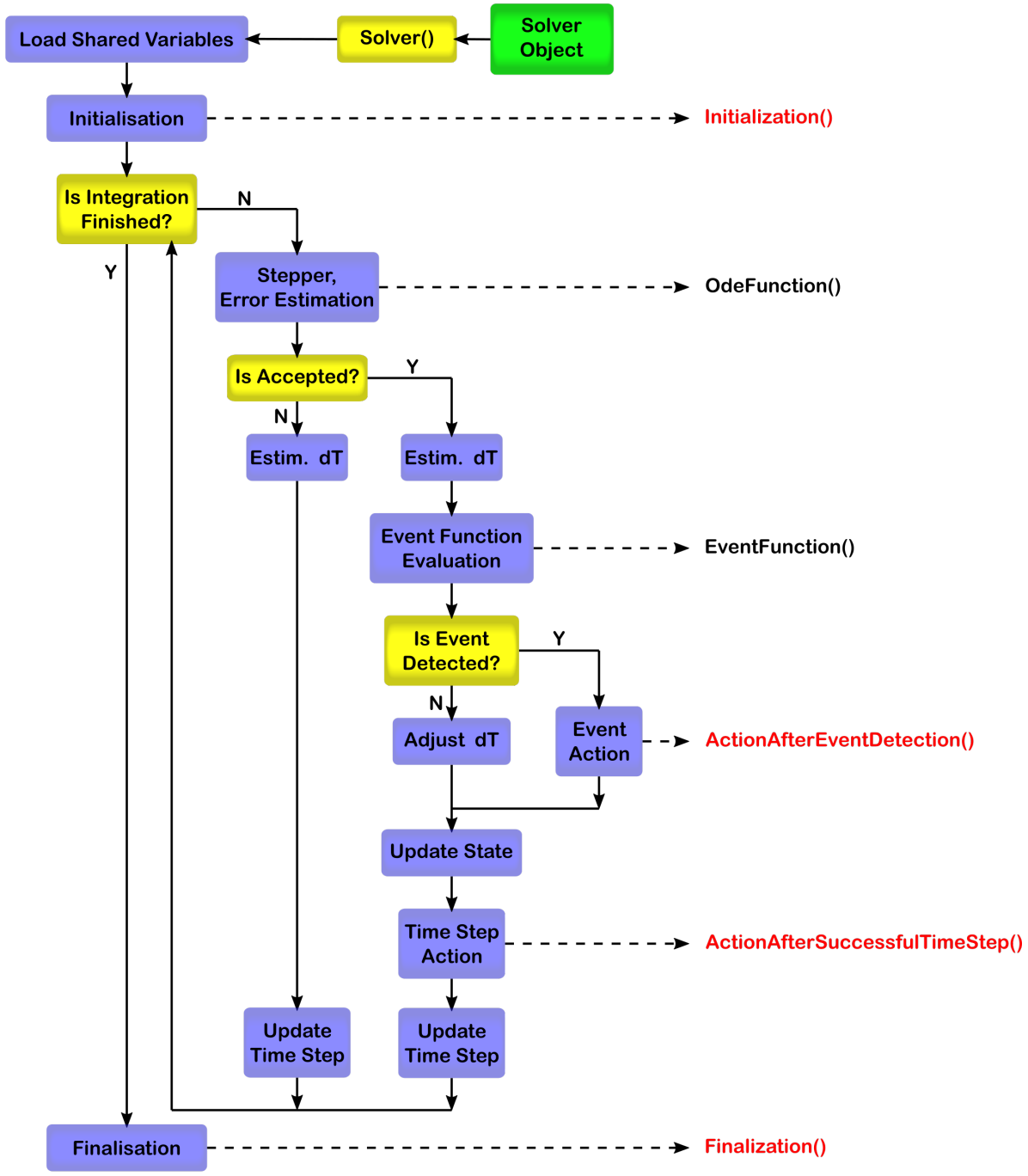

Figure 10: The general, internal structure of the `Solver()` member function.

In Fig. 10, the pre-declared, user-programmable special functions with black colour is also presented in MATLAB. All the other functions marked by red colour is a specialty of the present program package MPGOS. They provide a great flexibility to manipulate and store properties of the trajectories during an integration process. But why is it so important to incorporate altogether 6 functions for a GPU solver compared to conventional CPU version of ODE solvers (a right-hand side and an even function). Let us highlight the importance via examples.

Imagine, that one needs to compute the largest Lyapunov exponent of the trajectories during the integration

phases. For this, usually a linearised version of the original ODE is attached to the system and solved simultaneously. The tricky part is that after every integration phases, the trajectory corresponding to the linearised subspace need to be normalised to magnitude unity. Without the ability to do that at the end or at the beginning of each integration phases, all the data of all the threads (instances of the ODE system) has to be copied to the CPU side, perform the normalisation, copy the data back to the GPU side, and finally continue the integration. Usually to obtain a precise Lyapunov exponent, thousand or even tens of thousands of integration phases is required. Since the copy process between the GPU and CPU takes place via the PCI-E bus, which is the slowest memory transaction during a program run, this can cause tremendous amount of overhead. With the `Initialisation()` or `Finalisation()` pre-declared device functions, the normalisation can be done immediately on the device practically without no overhead. In this way, if one needs only the largest Lyapunov exponent, with a simple loop (see again Sec. 10 for an example), the required quantity can be computed with only two copy through the PCI-E bus (one to initialise the whole GPU computations, and one for copy the final data back to the CPU).

The `Initialisation()` pre-declare device function can be also useful to initialise the user-programmable parameters (`Accessories`). If one needs, for instance, the maximum of a component of a trajectory at every integration phases, an `Accessories` has to be allocated to store the maximum. During the integration phase, this quantity can be updated after every successful time step via the device function `ActionAfterSuccessfulTimeStep()`. However, a proper initialisation of this quantity is needed to obtain good results. The proper initialisation is to set the initial value of the `Accessories` to the initial state of the system. This can be easily done by the `Initialisation()` device function. The above described process is useful for producing magnification diagrams. Using the aforementioned device functions has many advantages. The most important are that there is no requirement to store the dense output of a trajectory and there is no requirement to copy the whole dense output to the CPU side and apply a maximum algorithm to every trajectories. Thus, again a tremendous amount of overhead is eliminated, since everything is computed immediately in the GPU side with small amount of additional instructions.

As a final example, imagine that one has a non-smooth dynamical system that can exhibit impact dynamics. In this case at least one event function has to be incorporated to be able to detect the impact. In case of an impact detection, however, an impact law has to be applied. Again, we can face with the previous problem. If impact law can be applied only in the CPU side then the integration procedure has to be stopped, the corresponding data has to be copied to the CPU side, the trajectory can be manipulated here according to the impact law, the new data has to be synchronised back to the GPU side and finally the integration process can be continued. Besides the overhead of the data transfer via the PCI-E bus, the bigger problem is that it is very unlikely that all the threads in a warp (the smallest number of thread organisation unit performing an instruction, for details see Sec 12) exhibit an impact at the same time. What should we do with trajectories having no impact? Stop the whole integration process just because of a single impact detection? Or make the thread(s) idle and wait for other threads exhibit impact as well? But what if some threads will never exhibit an impact? In every cases, the control flow can be extremely complicated if both the CPU and GPU have to be involved. The device function `ActionAfterEventDetection()` offers a very elegant solution for this problem. It is called after every successful event detection. Thus the impact law can be implemented directly here and can be performed immediately on the GPU. It is called only for thread(s) having impact, the rest will be idle. After the call, all the threads continue the integration process immediately. The idling state of the other threads is minimal, since applying an impact law is a usually less resource intensive computation compared to a complete time stepping. Moreover, impact (the source of idling state) usually happens only few times during an integration process compared to the total number required time steps.

All of the aforementioned special device functions are found in the system definition file, here it is named as `Reference_SystemDefinition.cuh`. It is very important that these functions inside this file cannot be renamed; otherwise, the solver algorithm will not be able to find them. If one intends to examine another system, it is possible to create a completely new system definition file in the sysme directory and include this new system definition file. Observe also that the system file is actually a CUDA header file with an extension `.cuh` which is included simply into the main source file. This “trick” is important due to performance consideration. In this way, the whole program is actually compiled together in the same

module; thus, the compiler has many options for optimisation (e.g., function inlining) that would not be possible otherwise. Compiling the solver file and the system file in a separate module, and then link them together afterwards cause approximately 30% performance loss even in case of the simple reference case.

Before proceeding further into the details of the pre-declared function, the abbreviated names of the used arguments and their descriptions are summarised in Tab. 9.

11.1 The right-hand side of the system

The righ-hand side of the system has to be defined in the pre-declared device function called `PerThread_Ode Function()`. For the reference case, its code snippet is as follows

```
__forceinline__ __device__ void PerThread_OdeFunction(int tid, int NT, double* F, double*
     X, double T, double* cPAR, double* sPAR, int* sPARi, double* ACC, int* ACCi)
{
   F[0] = X[1];
   F[1] = X[0] - X[0]*X[0]*X[0] - cPAR[0]*X[1] + sPAR[0]*cos(T);
}
```

Observe that the qualifier of the function is __device__; that is, it is callable from the Device (from a kernel or a device function) and runs on a Device. The __forceinline__ qualifier forces the compiler to inline the device function; thus, allowing optimisation. The compiler can automatically do it even without using this qualifier. Observe that the implementation of the right-hand side is very similar as in the case of MATLAB. But still, keep in mind again that the indexing starts from 0 in C++.

Keep in mind, that there is NO bound check for performance reasons. Therefore, double check the indexing and the sizes of each variables. The `SharedParameters` (`sPAR`) are immediately loaded into the so-called shared memory that is a really fast memory type of GPUs (similarly to CPUs, these memories are caches of GPUs). In addition, a single request from a shared memory can be "broadcasted" to every thread further reducing the pressure on memory operations. Therefore, it is advisable to define every parameters common to all instances of an ODE system as shared.

Observe that via the user programmable accessories *ACC*, the user has great flexibility during the evaluation of the right-hand side. For instance, if a term should be approximated to the first order, it can be stored in an accessory. Then, the related accessory can be updated after every successful time step (see Sec, 11.3) refreshing the first order approximation. Next, it can be used during the evaluation of the right-hand side during the next step. Although such a hybrid "algorithm" is seldom used, there can be cases where it can save a tremendous amount of computations if the proper evaluation of such terms is extremely complicated.

11.2 The event functions

In the reference calculations, two events are specified as follows

```
__forceinline__ __device__ void PerThread_EventFunction(int tid, int NT, double* EF,
    double* X, double T, double* cPAR, double* sPAR, int* sPARi, double* ACC, int* ACCi)
{
   EF[0] = X[1];
   EF[1] = X[0];
}
```

The first event function is $F_{E1} = x_2$ evaluated into the variable EF[0] meaning that a special point is detected if $x_2 = 0$. Since $\dot{x}_2 = x_1$, this means the local maxima or minima of the varible x_1, see Sec. 7. The second event function is $F_{E2} = x1$ evaluated into the variable EF[1]; that is, a special point is detected if the value of x_1 is zero (zero displacement). Observe that most of the possible arguments listed in Tab. 9 are passed to this function providing a great fexibility to define an event function. For instance, the number of the succesful time steps can be registered in a user programmable accessory variable `ACC`, then it can be used to specify an event corresponding to a maximum number of time steps. After that, even a stop condition can be specified (see Sec. 9).

Table 9: Summary of the input arguments of the pre-declared user functions

Argument	**Description**
tid	The serial number of the threads started from 0 to $N_T - 1$. Since one thread solves one system in the Solver Object, this is also the serial number of the systems.
NT	The number of the threads N_T. In the reference case it is $N_T = 23040$. It is also the number of the systems solved simultaneously.
F	The array to where the right-hand side of the system is evaluated. If the system dimension is denoted by N_{SD}, its total length is $N_{SD} \times N_T$.
X	The actual state of the systems.
T	The actual time of the systems.
dT	The actual time step.
TD	The time domain of the integration. There are two values for each system; t_0 and t_1. Therefore, its total length is $2 \times N_T$.
cPAR	The control parameters of the systems. If the number of the control parameters of a system is denoted by N_{CP}, its total length is $N_{CP} \times N_T$.
sPAR	The shared parameters of the system. If the number of the shared parameters of a system is denoted by N_{SP}, its total length is exactly N_{SP}, as it is shared among all the threads/systems.
sPARi	The same as sPAR but for type `int`. If the number of the integer shared parameters of a system is denoted by N_{SPi}, its total length is exactly N_{SPi}, as it is shared among all the threads/systems.
ACC	The accesories of the systems. If the number of the accessories of a system is denoted by N_{ACC}, its total length is $N_{ACC} \times N_T$.
ACCi	The same as the accesories but for type `int`. If the number of the integer accessories of a system is denoted by N_{ACCi}, its total length is $N_{ACCi} \times N_T$.
EF	The array to where the event functions are evaluated. If the number of event functions is denoted by N_{EF}, its total length is $N_{EF} \times N_T$.
IDX	The serial number of the detected event function in case of succesful event detection.
CNT	The number of the already detected events of the event function corresponding to the event serial number IDX.

11.3 Action after every succesful event detection

After every successful event detection, a special pre-defined user function is called:

```
__forceinline__ __device__ void PerThread_ActionAfterEventDetection(int tid, int NT, int
    IDX, int CNT, double &T, double &dT, double* TD, double* X, double* cPAR, double*
    sPAR, int* sPARi, double* ACC, int* ACCi)
{
   if ( X[0] > ACC[0] )
      ACC[0] = X[0];

   if ( (IDX==1) && (CNT==2) )
      ACC[1] = X[1];
}
```

Inside this function, many special operations can be performed. For instance, the trajectory can be manipulated or some properties related to an event can be stored into an accessory. In case of the example above, the local maxima (local, as it is detected via an event) of the Duffing system is stored into the first accessory (its serial number is 0) via testing the value of x_1 with the previous value of the related accessory.

In the last part of the code snippet, the value of x_2 is stored into the second accessory (index 1) at the second successful detection ($CNT == 2$) of the second event function ($IDX == 1$). Keep in mind again that the indexing starts from 0; therefore, the value of the event index IDX should be 1. This example has no particular physical importance, it is just an example to demonstrate the flexibility and strengths of the implemented event detection mechanism.

A very powerful feature of this pre-declared user function relates to impact dynamics as it is discussed above. After the detection of an impact, the corresponding trajectory can be "thrown away" to another location determined by a suitable impact law. This can be done simply by overwriting the variable X, see also Sec. 13.4. Since any kind of control flow algorithm can be implemented inside the function body, usefulness of this function is dependent on the imagination of the user.

11.4 Action after every successful time step

Similarly to the Sec. 11.3, an action implemented by a control flow inside the following function

```
__forceinline__ __device__ void PerThread_ActionAfterSuccessfulTimeStep(int tid, int NT,
    double T, double dT, double* TD, double* X, double* cPAR, double* sPAR, int* sPARi,
    double* ACC, int* ACCi)
{
   if ( X[0] > ACC[2] )
      ACC[2] = X[0];
}
```

can be performed after every successful time step. The states are already updated including the actual time instance but the time step is still NOT updated, see also Fig 10 and the corresponding discussion. The only difference is that properties related to event handling are not passed as arguments. The above example stores the global maxima of x_1 into the third accessory (its index is 2). It is global since the accessory shall be overwritten at every time step even if x_1 increases monotonically.

11.5 Initialisation before every integration phase

It is possible that some variables need to be initialized properly before performing the integration. It can be done inside the following function

```
__forceinline__ __device__ void PerThread_Initialization(int tid, int NT, double T,
    double &dT, double* TD, double* X, double* cPAR, double* sPAR, int* sPARi, double*
    ACC, int* ACCi)
{
   ACC[0] = X[0];
```

```
    ACC[1] = X[1];
    ACC[2] = X[0];
}
```

This function is called only once at the beginning of each integration phase. Here, the first three accessories are initialised according to the initial state of the system (in order: the accessories stores the local maxima of x_1, the x_2 value detected via the second event function, and finally the global maxima of x_1).

11.6 Finalisation after every integration phase

The function shown below is called after the end of every integration phase. Again, any kind of control logic can be implemented inside. In this specific example, it is left empty. Thus the compiler will optimise out the corresponding section of the program code, and there will be no function calls at all during the program run.

```
__forceinline__ __device__ void PerThread_Finalization(int tid, int NT, double T, double
    dT, double* TD, double* X, double* cPAR, double* sPAR, int* sPARi, double* ACC, int*
    ACCi)
{
}
```

12 Performance considerations

For small systems (approximately $N_{SD} \approx 10 - 20$), with the "default" setup introduced via the reference tutorial example, the performance will likely suffer **no** harm. The only issue the user has to take care about is to set a sufficiently large number of threads in order to fully utilise the GPU's arithmetic processing units (approximately $N_T > 5000 - 10000$).

For larger systems, e.g., those that come from a discretisation of a partial differential equation, the improper setup can have a significant impact on the performance. To avoid very inefficient and suboptimal code, at least some basic knowledge are needed of the thread organisation; of the GPU architectures; of the memory hierarchy; and finally of the mapping between hardware resources, data and threads. Through the following subsections, these fundamentals will be introduced as painlessly as possible for those who are new to GPU programming. Hopefully, these general guidelines can help to write the pre-declared user functions (e.g., the right-hand side) efficiently.

12.1 Threading in GPUs

The basic logical unit performing calculations is a thread. The number of threads simultaneously reside 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 *grid*. However, 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 total number of threads N_T are divided into thread blocks. The variable `ThreadsPerBlock` introduced in Sec. 9 defines the number of threads that can reside in a single block. In the reference case, the number of threads is $N_T = 46080$ and the block size is $N_{BS} = 64$; that is, the number of the blocks is $N_B = N_T/N_{BS} = 720$.

12.2 The parallelisation strategy

The parallelisation strategy of MPGOS is to assign one instance of the ODE system to a single thread. That is, one thread solves one instance of Eq. (1), but different threads work on a different set of data (time domain, initial condition, control parameters and accessories). We call this technique a per-thread approach.

12.3 The main building block of a GPU architecture

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 a number of processing units capable of performing floating point operations, load/store operations from the Global Memory or from the Shared Memory, control flow operations or other specialised instructions. The workload in a GPU is distributed to SMs with block granularity. That is, the block scheduler of the GPU (Giga Thread Engine) fills every SM with blocks until reaching hardware or resource limitation, for the details see Secs. 12.5 and 12.7. The maximum number of the residing threads in an SM is dependent on the compute capability of the hardware. Unfortunately, there is no CUDA API to query this information, the user have to retrieve this number from the CUDA Programming Guide [1] according the Compute Capability of the hardware. For the Kepler architecture used during the reference calculations (Compute Capability 3.5, see also Sec. 6), the maximum number of blocks residing in an SM simultaneously is 16. For the same architecture, the number of the SMs are 15, see again Sec. 6 on how to query this information. This means that the total number of blocks $N_B = 720$ cannot be distributed to the SMs all at the same time. The strategy of the Giga Thread Engine is that if computation of a block is finished in an SM, its computational resources are freed and a new block is immediately assigned to the SM. This cycle continues until all the block have been processed. This distribution technique has a consequence that blocks can be processed in any order; thus, the user should not write a code assuming a specific order of block assignment. Processing of blocks is independent of each other. The program package MPGOS fulfils this requirement as every thread is inherently independent of each other.

The above discussion has an impact on the performance: the total number of blocks should be an integer multiple of the number of the SMs. In this way, the total computational workload can be distributed evenly between the SMs, assuming that the time required to process the blocks are nearly the same. Otherwise, during the last phase of the computation, some SMs shall be idle. This phenomenon is called tailing effect. In the reference case, the total number of the blocks during one simulation launch is $N_B = N_T/N_{t/n} = 23040/64 = 360$, where $N_{t/b} = 64$ is the number of threads in a block (the block size), see the code snippet of the full code in Sec. 10. Thus, the number of blocks assigned to an SM during a simulation launch is $360/15 = 24$. Naturally, if the total number of the blocks during a run is very high, the tailing effect is minimal. In summary, the number of threads N_T, the block size $N_{t/b}$ and the number of the SMs together determine the tailing effect.

12.4 Warps as the smallest units of execution

The thread blocks are further divided into smaller chunks of execution units called warps. Each warp contains 32 number of threads (as a current CUDA architectural design). The warp schedulers of an SM take warps and assign them to execution units one after another until there are eligible warps for executions or 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 perform the same instruction but on different data. This technique is known as single instruction multiple data paradigm (SIMD). Therefore, every thread in a warp executes their series of instructions in order. This is the only way the hardware can efficiently handle a massive number of threads, as for 32 number of threads only one control unit is necessary to track their program state. This results in more place for compute units and more arithmetic throughput.

The drawback of the SIMD approach is the possibility of thread divergence occurring when some threads have to do different instructions from the others. The simplest case is the `if-else` conditional statements. If some threads take the `if` path and the rest take the `else` path, then the two paths can be evaluated only in a serial manner. First, the `if` path is evaluated to the related threads while the rest of the threads are idle, then the `else` path is evaluated in a similar way. As a rule-of-thumb, the user should minimize the conditional statements in the pre-declared device functions summarised in Sec. 11.

Another structure can cause divergent threads are loops where different threads do a different number

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

of cycles. Again some threads shall be idle and waiting for completing the computations of the thread performing the highest number of cycles. This case is interesting using adaptive algorithms. Since different parameter sets usually need a different number of required time steps during an integration phase, thread divergence is always presented for the adaptive algorithms. However, if the user fills up the SolverObject so that the consecutive systems have similar parameter sets, the thread divergence phenomenon can be minimised.

12.5 Hardware limitations for threads, blocks and warps

There are several hardware limitations on how many threads, blocks and warps can reside simultaneously in an SM. There are also limitations on how many threads can reside in a single block, and on the maximum dimensions of the grid of blocks and block of threads. Keep in mind that the block and grid dimensions are 3D structures; however, for our purpose, a linear 1D organisation is sufficient. Thus, only the first component of their dimensions are relevant for the program package MPGOS. Most of these hardware restrictions can be queried by the built-in function `ListCUDADevices()`, see again Sec. 6. The only exception is the number of blocks that can simultaneously reside in an SM discussed already in Sec. 12.3. The information has to be obtained from the CUDA Programming Guide [2].

12.6 The memory hierarchy

During the evolution of CPUs and GPUs, they followed different design paths. It is a common problem that the processing power of both types of processing units is so high that the data transfer bandwidth from the System Memory (CPU) or from the Global Memory (GPU) is far less than the required to fully utilise the computing capacities. Moreover, the latencies (the elapsed time between the request and the arrival of the data) of these memory types are really high. To address these problems, the CPU and GPU manufacturers gave different solutions. CPUs have large low latency and high bandwidth on-chip caches (compared to the number of its threads) which store a large amount of data that can be reused during a computation. This approach is called latency-oriented design, and it is suitable for a relatively small number of parallel threads and to handle complex control flows. It is called latency-oriented since the purpose of the large amount of on-chip caches is to reduce the access latency of a data.

On the other hand, GPUs have a small amount of on-chip caches (on each SM) compared to its number of threads. In order to hide the large latencies, GPU operates with a massive number of threads (instead of storing a large amount of data in cache); that is, there is a high probability that the GPU can find a warp that is eligible to perform an instruction (its data has already arrived from the Global Memory). The GPU core can switch issuing instructions between warps almost with no cost in time. The advantage of such an approach (keep latency high) is that the memory bandwidth can be dramatically increased. Therefore, GPUs are designed to handle latency with a massive number of threads and process a huge amount of data.

Although the bandwidth of the Global Memory (GPU side) is much higher than the System Memory (CPU side), it is still much lower than the required to fully utilise the arithmetic processing units of a GPU. Therefore, the memory hierarchy plays an important role in GPU programming as well. In fact, it is even more important to understand the details of the memory hierarchy since fast memories (e.g., shared memory and registers) are scarce compared to a CPU design. In the next subsections, the most important details of the different memory types are summarised. Fortunately, the program package MPGOS is designed so that it already exploits the possibilities of the memory hierarchy. The user has to keep in mind only the short details below to achieve high-efficiency code. This means mainly an implementation of an efficient right-hand side.

[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

12.6.1 Global memory

The Global Memory is the slowest memory type of a GPU. As a compensate, it has the biggest size, in the orders of Gbytes. The data resides in the global memory is accessed via 128 byte (consecutive) memory transactions [3]. This is the smallest amount of data that can be delivered in a single request. Therefore, in order to fully utilise the memory bus, threads in a warp during a global memory load/store operation should access consecutive memory locations. For instance, if each thread in a warp require 4 bytes (e.g. a float), it can be delivered via a single 128 byte memory transactions, and memory bus utilisation is 100 %. In any other cases, the number of memory transactions are increased, and the global memory load/store efficiency decreases. Therefore, such coalesced memory accesses are mandatory for highly efficient code. Fortunately, the users do not have to worry about coalesced accesses. The program package is already written to access global memory in an optimal, coalesced way.

12.6.2 Shared memory

Shared memories of the GPUs are on-chip, low latency and high-bandwidth memory types. It is on-chip as every SM has a certain amount of its own shared memory. The total shared memory/SM is an architectural property, and it can be listed with the function call `ListCUDADevices();`, see Sec. 6. Shared memory can be allocated by block granularity; that is, each block has its own amount of allocated shared memory. All threads within a block can "see" the content of the corresponding shared memory. In this way, threads in a block can cooperate with each other. It is important to note that threads within different blocks cannot cooperate as they cannot access the shared memory of another block. Remember that the execution order of the blocks and the block assignment to SMs are not deterministic, see Sec. 12.3. Thus, one cannot rely on the cooperation between threads residing in different blocks.

The program package MPGOS offers the possibility to exploit the advantage of shared memory by putting parameters common to all threads into the shared memory, for the technical details see Sec. 9. The main advantage is that the expensive (in time) Global Memory load operations can be minimised with the proper use of the shared memory. For the reference tutorial example, this technique does not have a significant impact as the number of the shared parameters is only one, which could have been hard-coded into the right-hand side evaluation.

However, there can be situations where shared memories can have a significant impact on performance. A perfect example if the evaluation of the right-hand side involves matrix-vector multiplication where the matrix is identical for every system. Such cases occur, for instance, if the instances of Eq. (1) involves discretisation of a partial differential equation. Here the matrix-vector multiplication comes from the spatial derivation of the state variables, where the matrix is the so-called differentiation matrix shared among all the systems. Keep in mind that each instance of systems must have the same form; namely, the same function of right-hand side: function f in Eq. (1). Therefore, one has a large amount of a semi-discretised partial differential equation with different parameter sets, where the spatial discretisation scheme must be the same with the same distribution of collocation points to ensure that the function f will be identical. By placing the differentiation matrix into the shared memory, the pressure on global memory load operations can be reduced even by orders of magnitudes: from $n^2 + n$ (load of matrix elements and the state variables) to only n (load only of the state variable), where n is the dimension of the matrix and the vector (involved state variables). It is important that multi-dimensional arrays are not supported; thus, the **matrix have to be stored as a linear sequence of shared parameters**. This needs special care of indexing by the user in the pre-declared device functions.

Some other examples for the possible use of shared memories: coupled systems where the **coupling marix** is the same for all systems; the right-hand side of the system is very complicated and contains **many pre-computed coefficients**, and some of them can be shared among all systems, for an example see Sec. 13.3.

[3] It is not as simple; however, it is sufficient to understand the explanation during this section.

12.6.3 Registers

Registers are the fastest memory types of a GPU, practically it has no latency. The total amount of registers of every SM is also an architectural property. The total amount of 32-bit registers can be queried by the function call `ListCUDADevices()`, see Sec. 6. Parenthetically, a `double` (8 byte) requires 2 number of 32-bit registers. The allocation of registers takes places by thread granularity. That is, every thread has its own set of registers that can be accessed only by the corresponding thread. Variables allocated inside a kernel or device functions (e.g., the pre-declared user functions) are placed into registers. For instance in the following code snippet

```
double x1 = X[0];
double x2 = X[1];

double p1 = cPAR[0];
double p2 = sPAR[0];

F[i0] = x2;
F[i1] = x1 - x1*x1*x1 - p1*x2 + p2*cos(T);
```

the variables `x1`, `x2`, `p1` and `p2` are put into the registers. Keep in mind, however, that registers are scarce resources. Allocating large arrays or structures inside a pre-declared user function (e.g., in the right-hand side evaluation), the compiler will automatically place them into the slow global memory if there is not enough available registers. This phenomenon is called register spilling done "behind the scenes". An optimising compiler also allocates some amount of registers for intermediate storages to accelerate computations. The total required registers depends on the systems complexity and usage of transcendental functions like sin, log, division or power. Compiling the code with the option **`--ptxas-options=-v`**, the compiler will write information about the required registers for a kernel function:

```
ptxas info    : Compiling entry function '_Z20PerThread_RKCK45_EH027Integrator...
ptxas info    : Function properties for _Z20PerThread_RKCK45_EH027IntegratorInternal...
    32 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 175 registers, 32 bytes smem, 464 bytes cmem[0], 372 bytes cmem[2]
ptxas info    : Function properties for __internal_accurate_pow
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Function properties for __internal_trig_reduction_slowpathd
    40 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
```

Here, the kernel function **PerThread_RKCK45_EH0** requires 175 number of 32-bit registers for each thread. Letting the compiler use as many registers as it requires may not be feasible. If a thread needs a large number of registers than the total number of threads residing in an SM can be significantly reduced, for details see Sec. 12.7. This can have a significant impact on the performance as a small number of residing threads have less opportunity to hide the high Global Memory latency, see again the introduction of Sec. 12.6.

The only way the user can limit the maximum number of registers used by a thread is to compile the code with the compiler flag **`-maxrregcount=64`**, which limits the maximum number of used registers/thread to 64 (it can be any kind of integer number below the architectural limit [4]; in most devices, it is 255). The compiler output now looks like this:

```
ptxas info    : Function properties for _Z20PerThread_RKCK45_EH027IntegratorInternalVariables
    40 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 64 registers, 32 bytes smem, 464 bytes cmem[0], 372 bytes cmem[2]
ptxas info    : Function properties for __internal_accurate_pow
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Function properties for __internal_trig_reduction_slowpathd
    40 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
```

[4] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

It must be stressed that the maximum number of registers is an important parameter for code optimisation. The user has to try several options for his/her system. However, some rules-of-thumb can be given. For simple systems like the Duffing or Lorenz, a relatively small number of registres are enough (64, 80, or 128). For more computationally intensive systems, the number of registres should be 128 or above. Meanwhile, according to our experiments, the block size should be 32 or 64. Trying the combination of these numbers, the user will very likely obtain efficient code.

12.7 Resource limitations and occupancy

The number of simultaneously residing threads and blocks in an SM is not limited only by the hardware restrictions (Sec, 12.5) but also by their resource usage (registers and shared memory). If the maximum number of registers per SM is denoted by R_{SM} and the number of allocated registers per thread is R_T, the maximum number of threads can simultaneously reside in the SM is $N_{TSM} = R_{SM}/R_T$. For instance, in our Titan Black card $R_{SM} = 65536$ and $R_T = 64$ (compiler option); the maximum number of residing threads in this case are $N_{TSM} = 1024$. The hardware limitation is 2024 (see Sec. 6); that is, only with $R_T = 32$ can the hardware limitation be saturated. Therefore, setting the compiler option for register usage smaller than 32 is meaningless.

Similar considerations can be made corresponding to the shared memory and the maximum residing blocks in an SM. If the total amount of shared memory of an SM is S_{SM} and the shared memory required for a block is B_{SM}, then the maximum number of block that can reside in an SM is $N_{BSM} = S_{SM}/B_{SM}$. For instance, if the total amount of shared memory of an SM is 48 Kb and the required shared memory of a block is 8 Kb (e.g. a 32×32 matrix of `doubles`), then the maximum number of block that can reside in an SM is 6 that is below the hardware limitation of our Titan Black card: 8.

In general, the total amount of threads residing in an SM during a computation is limited by the strongest constraint: limitation by register, limitation by shared memory or limitation by hardware. For instance, even if the maximum registers used by a thread is 32, the hardware limitations cannot be achieved (2024) when the limitations for blocks is 6 with a block size of 64. In such a situation, the total amount of threads is limited by shared memory: $6 * 64 = 384$ that is far from the hardware limitation.

The terminology called occupancy O is a measure of the maximum residing threads in an SM compared to the hardware limitation in percentage. In the case of the above example, the occupancy is $O = 384/2024 = 0.19$ (approximately 20%) that is rather low. With lower occupancy, there is a lower chance to hide the latency of the Global Memory load/store operations. The calculation of theoretical occupancy can be a cumbersome task as one have to take into account many factors. In order to ease this problem, CUDA offers an Excel datasheet called "CUDA_Occupancy_calculator" [5] to easily calculate the theoretical occupancy as a function of the compute capability of the hardware, used registers and allocated shared memory. The latest version can also be found in the GitHub repository of the program package MPGOS.

It must be emphasised that hunting for 100% occupancy is not always required. Kernel functions which require intensive arithmetic operations compared to global memory transactions works fine with low occupancy as there is little pressure to hide the latency of the relatively few memory operations. On the other hand, if the ratio of the number of the arithmetic operations and the global memory transactions is low, the high occupancy is a must. In this case, any smart exploitation of the shared memory can be crucial.

During the creation of a SolverObject, the memory requirements are automatically checked. If the required resources are not enough, the program run will be terminated.

12.8 Maximising instruction throughput

As a general rule, floating point and integer addition/multiplication can be performed within one clock cycle; that is, these operations are fast. However, there are really expensive arithmetic computations such as the

[5] https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

usage of any kind of **trascendental functions** (e.g., sine, cosine, logarithm etc.), integer or floating point **division** and the commonly used **power** function `pow()`. These operations have much larger latencies and need many clock cycles to perform the instruction. Moreover, they require a large number of registers during their intermediate computations. Therefore, the user should minimise their usage as much as possible:

- If the reciprocal of a variable is necessary more than one times, it is best to calculate only once in an intermediate variable and then apply multiplication successively.
- Put such expensive operations inside a loop only if it is really necessary. Otherwise, the performance of the code will perish.
- Do not use the function `pow()` unless it is the only possible option to perform the calculation. For integer power, use successive multiplications: instead of `pow(x,2)` use `x*x`. For fractional power, try to combine division, square root function `sqrt()`, reciprocal square root function `rsqrt()` and their cubic counterpart [6]. For instance, instead of `r=pow(x,2/3)` use `r = cbrt(x); r = r*r`. For details, see the link in the footnote.
- If both the sine and the cosine values of a variable are required then use the function `sincos(x,sptr, cptr)`, where the variables `sptr` and `cptr` are passed by reference and will contain the calculated trigonometric values, sine and cosine, respectively.
- One can try to calculate the expensive transcendental function with type `float` instead of `double`. The only thing the user has to do is to replace the corresponding function with its `float` version: e.g. use `sinf(x)` instead of `sin(x)`. The output is less accurate; thus, use it with care.
- The calculation of transcendental functions can be further accelerated by use their intrinsic: e.g., use `__sinf(x)` instead of `sinf(x)`. The only difference is the prefix `__`. This is even less accurate but faster as the GPU can use dedicated compute units (Special Function Units, SFUs) during the calculations. Again use them with care. The best way is to replace the expensive transcendental functions one after another and carefully monitor the effect on the accuracy.

12.9 Profiling

The program package MPGOS provides a well prepared Linux shell script `Profile.sh` or for exhaustive profiling. Run it simply via the command

```
./Profile.sh executable.exe output
```

where the first option is the executable (including the file extension) and the second one is the name of the output file (without extension). It has three phases. First, an aggregated statistics is written about the used kernel functions. Second, an output file is created for visual profiler `output.nvprof`. Third, the most important event and metrics are listed about the related kernel functions. The simplified output of the third phase of the profiling is listed below for the reference tutorial example The first two phases is written into the file `output.log`.

The detailed analysis of all these metrics and events are beyond the scope of the present manual, only the most important metrics are summarised briefly:

- Multiprocessor Activity: it measures the utilisation of the SMs with blocks. It is 83.98% meaning that some SM shall be idle during the last phase of the computations. The reason is most probably due to thread divergence caused by the adaptive algorithm and intensive usage of events. Nevertheless, this value is still considered good. This number can be increased by using more blocks (possibly with more threads/systems).

[6] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#other-arithmetic-instructions

- Achieved Occupancy: it is the achieved occupancy during the kernel run. The theoretical occupancy here is 0.5. Therefore, the 0.42 achieved occupancy is considered good.
- Global (Local Memory) Store Throughput and Global (Local Memory) Load Throughput: if the sum of these throughput values (203 Gb/s) are close the theoretical bandwidth (for our Titan Black card 336 Gb/s), the code is very like to be limited by memory bandwidth. In this case, the Global Memory load/store transactions should be reduced; for instance, via more extensive usage of the shared memory (shared parameters). The reference tutorial example is not bound by memory bandwidth.
- Eligible Warps Per Active Cycle: the average number of eligible warps in an active cycle. A warp is eligible if it is ready for computations; that is, all the data required for the next calculation have already arrived from the Global Memory. This value should be above the number warp schedulers of an SM. In the Titan Black card, it is 4.
- Arithmetic Function Unit Utilization: it measures the arithmetic function utilisation of an SM in a scale between 0 and 10. This is the most important metrics, as one would like to utilise the full processing power of a GPU. As we can see, it is already at its maximum; therefore, the kernel here is arithmetic compute bound. Optimising memory accesses, in this case, will give no performance increase.
- Comparison of the Arithmetic Function Unit Utilization and Global Store/Load Throughput: if both the memory bandwidth and the arithmetic function unit are underutilised, the kernel code is bound by latency. In this case, try to increase the occupancy to hide latency. If the occupancy is already high, try to reorganise the code to reduce data dependencies. That is, try to collect instructions close to each other which can be performed independently so that a warp does not have to wait for data from a previous computation, and can continue doing something.

The Nvidia profiling tool nvprof used in the above profiling script is deprecated. Thus, its proper functioning is not guaranteed. However, the profiling script is still kept as a part of the program package. In a later version, the profiling script will be updated.

There is a simplified version (omit the visual profiler output and reduce the number of the investigated metrics) of the profiling script for faster profiling:

```
./ProfileSimplified.sh executable.exe output
```

Metric Description	Min	Max	Avg
Multiprocessor Activity	83.98%	83.98%	83.98%
Achieved Occupancy	0.422526	0.422526	0.422526
Eligible Warps Per Active Cycle	5.687683	5.687683	5.687683
Texture Cache Throughput	1.3638MB/s	1.3638MB/s	1.3638MB/s
Device Memory Read Throughput	8.5473GB/s	8.5473GB/s	8.5473GB/s
Device Memory Write Throughput	48.127GB/s	48.127GB/s	48.127GB/s
Global Store Throughput	48.121GB/s	48.121GB/s	48.121GB/s
Global Load Throughput	155.25GB/s	155.25GB/s	155.25GB/s
Local Memory Load Throughput	0.00000B/s	0.00000B/s	0.00000B/s
Local Memory Store Throughput	4.7957GB/s	4.7957GB/s	4.7957GB/s
Shared Memory Load Throughput	32.003GB/s	32.003GB/s	32.003GB/s
Shared Memory Store Throughput	148.22MB/s	148.22MB/s	148.22MB/s
L2 Throughput (Reads)	155.26GB/s	155.26GB/s	155.26GB/s
L2 Throughput (Writes)	48.125GB/s	48.125GB/s	48.125GB/s
L2 Throughput (L1 Reads)	155.25GB/s	155.25GB/s	155.25GB/s
L2 Throughput (L1 Writes)	48.133GB/s	48.133GB/s	48.133GB/s
L2 Throughput (Texture Reads)	98.811KB/s	98.811KB/s	98.811KB/s
Global Memory Load Efficiency	100.00%	100.00%	100.00%
Global Memory Store Efficiency	100.00%	100.00%	100.00%
Shared Memory Efficiency	97.05%	97.05%	97.05%
Instructions Executed	85376141	85376141	85376141
Instructions Issued	180234200	180234200	180234200
Executed IPC	0.668142	0.668142	0.668142
Issued IPC	1.406563	1.406563	1.406563
FP Instructions(Double)	713292507	713292507	713292507
Integer Instructions	1021221389	1021221389	1021221389
Bit-Convert Instructions	50912200	50912200	50912200
Control-Flow Instructions	71323160	71323160	71323160
Load/Store Instructions	309966656	309966656	309966656
Misc Instructions	391405296	391405296	391405296
FLOP Efficiency(Peak Double)	44.42%	44.42%	44.42%
L1/Shared Memory Utilization	Low (1)	Low (1)	Low (1)
L2 Cache Utilization	Low (3)	Low (3)	Low (3)
Texture Cache Utilization	Low (1)	Low (1)	Low (1)
Device Memory Utilization	Low (2)	Low (2)	Low (2)
System Memory Utilization	Low (1)	Low (1)	Low (1)
Load/Store Function Unit Utilization	Low (2)	Low (2)	Low (2)
Arithmetic Function Unit Utilization	Max (10)	Max (10)	Max (10)
Control-Flow Function Unit Utilization	Low (1)	Low (1)	Low (1)
Texture Function Unit Utilization	Low (1)	Low (1)	Low (1)
Issue Stall Reasons (Pipe Busy)	52.24%	52.24%	52.24%
Issue Stall Reasons (Execution Dependenc	17.96%	17.96%	17.96%
Issue Stall Reasons (Data Request)	7.67%	7.67%	7.67%
Issue Stall Reasons (Instructions Fetch)	2.83%	2.83%	2.83%
Issue Stall Reasons (Texture)	0.00%	0.00%	0.00%
Issue Stall Reasons (Not Selected)	16.98%	16.98%	16.98%
Issue Stall Reasons (Immediate constant)	0.01%	0.01%	0.01%
Issue Stall Reasons (Memory Throttle)	1.32%	1.32%	1.32%
Issue Stall Reasons (Synchronization)	0.02%	0.02%	0.02%
Issue Stall Reasons (Other)	0.97%	0.97%	0.97%

13 MPGOS tutorial examples

This section serves as an additional material to highlight the flexibility and features of the program package MPGOS. The list of the tutorial examples will be continuously extended. The first tutorial example if the reference case discussed in detail above.

13.1 Tutorial 2: the Lorenz system

The second tutorial example computes 1000 number of time steps with the classic 4^{th} order Runge–Kutta (RK) method. Since the Lorenz equation

$$\dot{y}_1 = 10.0\,(y_2 - y_1), \tag{4}$$
$$\dot{y}_2 = p\,y_1 - y_2 - y_1\,y_3, \tag{5}$$
$$\dot{y}_3 = y_1\,y_2 - 2.666\,y_3, \tag{6}$$

is very simple (consists only additions and multiplications) it is a perfect example to test the "raw" efficiency of the GPU card. Moreover, due to the fixed time step, there is no thread divergence. In Eq. (6), p is the parameter varied between 0.0 and 21.0 with a resolution N distributed uniformly. Observe that values of p do not affect the performance, the code has to perform a fixed, 1000 number of steps, and each step requires a well-defined amount of work independently of the actual value of the parameter. The main aim of this section is to demonstrate the performance of the code by comparing the runtimes of this simple example with other program packages in the market. The first one is the odeint package[7] written in C++. The runtimes are digitalised from the publication of Ahnert et. al[8] from Fig. 7.1. The second program package is the DifferentialEquations.jl[9] written in the rapidly developing Julia language.

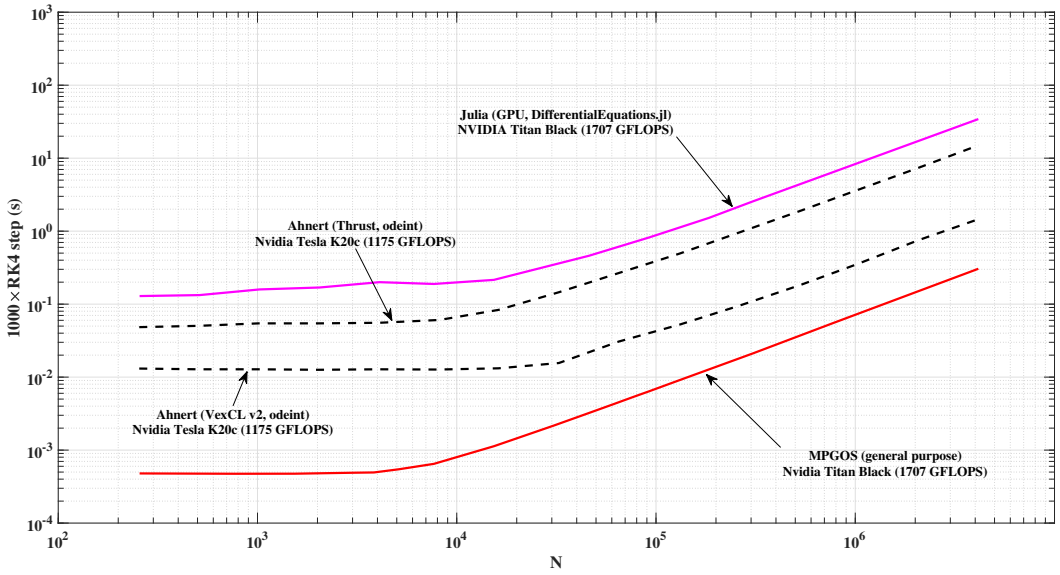

Figure 11: Comparison of the performance of different program packages; that is, the runtimes of the solution on N Lorenz system with the simple 4^{th} order Runge–Kutta scheme. The vertical axis is the runtime in seconds of 1000 steps.

[7] www.odeint.com
[8] 2014 Solving Ordinary Differential Equations on GPUs, Numerical Computations with GPUs pp. 125-157
[9] https://docs.juliadiffeq.org/stable/index.html

Figure 11 shows the comparison of the runtimes as a function of the resolution N. The red, black, and purple curves are the results of the MPGOS, odeint, and Julia program packages, respectively. It is clear that even in this simple example, MPGOS has a superior performance.

13.2 Tutorial 3: Poincaré section of the Duffing equation

The third tutorial example serves to investigate the effect of event handling and the excessive usage of accessories. For this purpose, the present example is a simplified version of the Reference example. It keeps only the pre-declared member function of the right-hand side of the ODE. Therefore, the code snippets do not repeat here, only the runtimes of the computations: the transient runtime is reduced to 3.30 s from 4.94 s, the total runtime is reduced to 6.18 s from 11.72 s. We can experience approximately 50% increase in the runtime using 2 event handling function and storing special properties into 3 accessories. The decrease of the total runtime is also due to the smaller number of data saved to disc.

13.3 Tutorial 4: quasiperiodic forcing

The model of the fourth tutorial example is the Keller–Miksis equation describing the evolution of the radius of a gas bubble placed in a liquid domain and subjected to external excitation. The second-order ordinary differential equation reads as

$$\left(1-\frac{\dot{R}}{c_L}\right)R\ddot{R}+\left(1-\frac{\dot{R}}{3c_L}\right)\frac{3}{2}\dot{R}^2=\left(1+\frac{\dot{R}}{c_L}+\frac{R}{c_L}\frac{d}{dt}\right)\frac{(p_L-p_\infty(t))}{\rho_L}, \tag{7}$$

where $R(t)$ is the time dependent bubble radius; $c_L = 1497.3\,\mathrm{m/s}$ and $\rho_L = 997.1\,\mathrm{kg/m^3}$ are the sound speed and the density of the liquid domain, respectively. The pressure far away from the bubble $p_\infty(t)$ is composed by static and periodic components

$$p_\infty(t) = P_\infty + P_{A1}\sin(\omega_1 t) + P_{A2}\sin(\omega_2 t+\theta), \tag{8}$$

where $P_\infty = 1\,\mathrm{bar}$ is the ambient pressure; and the periodic components have pressure amplitudes P_{A1} and P_{A2}, angular frequencies ω_1 and ω_2, and a phase shift θ. Such a dual-frequency driven gas bubble has paramount importance in the field of acoustic cavitation and sonochemistry.

The connection between the pressures inside and outside the bubble at its interface can be written as

$$p_G + p_V = p_L + \frac{2\sigma}{R} + 4\mu_L\frac{\dot{R}}{R}, \tag{9}$$

where the total pressure inside the bubble is the sum of the partial pressures of the non-condensable gas, p_G, and the vapour, $p_V = 3166.8\,\mathrm{Pa}$. The surface tension is $\sigma = 0.072\,\mathrm{N/m}$ and the liquid kinematic viscosity is $\mu_L = 8.902\times10^{-4}\,\mathrm{Pa\,s}$. The gas inside the bubble obeys a simple polytropic relationship

$$p_G = \left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\left(\frac{R_E}{R}\right)^{3\gamma}, \tag{10}$$

where the polytropic exponent $\gamma = 1.4$ (adiabatic behaviour) and the equilibrium bubble radius is R_E.

The detailed description and the physical interpretation of Eqs. (7)-(10) is omitted here. It must be emphasized, however, that the physical parameters of the system are the excitation properties: P_{A1}, P_{A2}, ω_1, ω_2, θ and the bubble size: R_E (if the material properties and the static pressure are fixed). This large parameter space is reduced by setting the bubble size to $R_E = 10\,\mu\mathrm{m}$ and the phase shift to $\theta = 0$. The main aim is to investigate the achievable maximum expansion ratio of the bubble radius $(R_{\max} - R_E)/R_E$ (important measure of the efficiency of sonochemistry) as high-resolution bi-parametric plots with excitation frequencies ω_1 and ω_2 as control parameters at fixed amplitudes P_{A1} and P_{A2}. Observe that in this case, **the external forcing can be quasiperiodic**; thus, special care have to be taken to handle the time domain during the simulations.

System (7)-(10) is rewritten into a dimensionless form defined as

$$\dot{y}_1 = y_2, \tag{11}$$

$$\dot{y}_2 = \frac{N_{\mathrm{KM}}}{D_{\mathrm{KM}}}, \tag{12}$$

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

$$\begin{aligned} N_{\mathrm{KM}} = (C_0 + C_1 y_2)\left(\frac{1}{y_1}\right)^{C_{10}} - C_2\,(1+C_9 y_2) - C_3\frac{1}{y_1} - C_4\frac{y_2}{y_1} - \left(1-C_9\frac{y_2}{3}\right)\frac{3}{2}y_2^2 \\ - \left(C_5\sin(2\pi\tau) + C_6\sin(2\pi C_{11}\tau + C_{12})\right)(1+C_9 y_2) \\ - y_1\left(C_7\cos(2\pi\tau) + C_8\cos(2\pi C_{11}\tau + C_{12})\right), \end{aligned} \tag{13}$$

and

$$D_{\mathrm{KM}} = y_1 - C_9 y_1 y_2 + C_4 C_9. \tag{14}$$

The coefficients are summarised as follows:

$$C_0 = \frac{1}{\rho_L}\left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{15}$$

$$C_1 = \frac{1-3\gamma}{\rho_L c_L}\left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\frac{2\pi}{R_E\omega_1}, \tag{16}$$

$$C_2 = \frac{P_\infty - p_V}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{17}$$

$$C_3 = \frac{2\sigma}{\rho_L R_E}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{18}$$

$$C_4 = \frac{4\mu_L}{\rho_L R_E^2}\frac{2\pi}{\omega_1}, \tag{19}$$

$$C_5 = \frac{P_{A1}}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{20}$$

$$C_6 = \frac{P_{A2}}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{21}$$

$$C_7 = R_E\frac{\omega_1 P_{A1}}{\rho_L c_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{22}$$

$$C_8 = R_E\frac{\omega_1 P_{A2}}{\rho_L c_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{23}$$

$$C_9 = \frac{R_E\omega_1}{2\pi c_L}, \tag{24}$$

$$C_{10} = 3\gamma, \tag{25}$$

$$C_{11} = \frac{\omega_2}{\omega_1}, \tag{26}$$

$$C_{12} = \theta. \tag{27}$$

Observe that from the implementation point of view, the number of the parameters of the system is 13 (C_0 to C_{12}). Therefore, the physical parameters ω_1, ω_2, P_{A1} and P_{A2} and the appearing systems coefficients as parameters must be clearly separated in the code. Although the usage of the coefficients C_{0-12}—instead of the physical parameters—requires additional storage capacity and global memory load operations, it can significantly reduce the necessary computations as these coefficients are pre-computed on the CPU during the filling of the Solver Object.

13.3.1 The objectives

This tutorial example calculates the collapse strength of the air filled single spherical bubble placed in liquid water and subjected to dual-frequency ultrasonic irradiation. An example for the radial oscillation of a bubble is demonstrated in Fig. 12, in which the dimensionless bubble radius $y_1 = R(t)/R_E$ is presented as a function of the dimensionless time τ. Keep in mind again that $R_E = 10\,\mu$m is the equilibrium bubble radius of the unexcited system. Parenthetically, at certain parameter values, the oscillation can be so violent that at the minimum bubble radius, the temperature can exceed several thousands of degrees of Kelvin initiating even chemical reactions. This phenomenon is called the collapse of a bubble. In the literature, there are various quantities characterising the strength of the collapse which is the keen interest of sonochemistry. For instance, the relative expansion $y_{exp} = (R_{max} - R_E)/R_E = y_1^{max} - 1$ or the compression ratio y_1^{max}/y_1^{min} are good candidates. In this sense, a bubble collapse can be characterized by the radii of a local maximum y_1^{max} and the subsequent local minimum y_1^{min}, see also Fig. 12. Observe that the time scales can be very different near the local maximum and the local minimum.

In general, during a long term oscillation of a bubble, the collapse strengths are different from collapse to collapse (e.g., due to chaotic behaviour or quasiperiodic forcing). Therefore, the properties of multiple

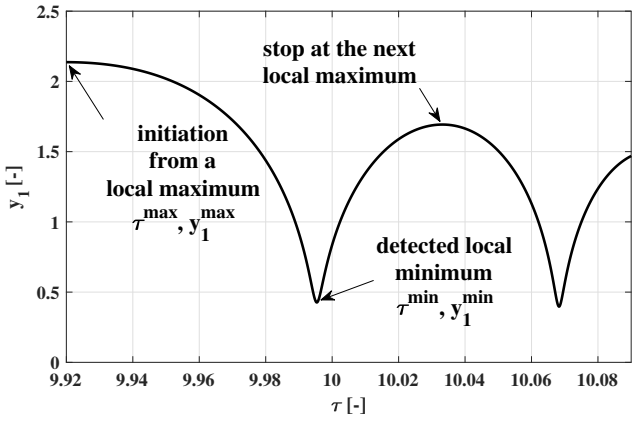

Figure 12: Demonstration of a bubble collapse via a dimensionless bubble radius vs. time curve. An integration start from a local maximum y_1^{max} at time instant τ^{max} and ends at the next local maximum. During the integration, the local minimum y_1^{min} and its time instant τ^{min} is also determined.

collapses are registered in order to obtain a realistic picture about the bubble behaviour. The easiest way to do this is to integrate the system from a local maximum to the next local maximum (one iteration) meanwhile monitoring and detecting the local minimum. The iteration process can be repeated arbitrary many times. After each iteration, the collapse properties are saved. It must be noted that some researchers take into account the elapsed time during a collapse as well. Thus, the time instances of the maxima and minima are also stored. This will need 4 user programmable parameters (accessories).

The present scan involves four physical parameters of the dual-frequency excitation; namely, the pressure amplitudes P_{A1} and P_{A2}, and the frequencies $f_1 = \omega_1/(2\pi)$ and $f_2 = \omega_2/(2\pi)$. Their minimal and maximal values, their resolutions (how many values are taken) and the type of their distribution (linear or logarithmic) are summarized in Tab. 10. Observe that the number of the investigated parameters of each pressure amplitude is only two (only the minimum and the maximum). The overall number of scanned parameters is $2 \times 2 \times 128 \times 128 = 65536$. In each simulation, the first 1024 iteration (collapses) are regarded as initial transients and discarded. The next 64 collapses are used to collect data that is written into a file.

Table 10: Values of the control parameters of the four dimensional scan.

	P_{A1} (bar)	P_{A2} (bar)	f_1 (kHz)	f_2 (kHz)
min.	0.5	0.7	20	20
max.	1.1	1.2	1000	1000
res.	2	2	128	128
scale	lin	lin	log	log

The saved data of a single instance of a system organised in a row are as follows: all the 6 physical parameters P_{A1}, f_1, P_{A2}, f_2, θ and R_E although the last two are constants, the final time of the transient simulation, the period of the next 64 collapses, the values of y_1^{max} of this 64 collapses and finally the values of y_1^{min} of the the 64 collapses. Therefore, there are $136 = 6 + 2 + 2 * 64$ columns in each text file.

13.3.2 Configuration of the Solver Object

The configuration of the Solver Object is as follows:

```
// Physical control parameters
const int NumberOfFrequency1 = 128;
const int NumberOfFrequency2 = 128;
const int NumberOfAmplitude1 = 2;
const int NumberOfAmplitude2 = 2;

// Solver Configuration
#define SOLVER RKCK45 // RK4, RKCK45
const int NT   = NumberOfFrequency1 * NumberOfFrequency2; // NumberOfThreads
const int SD   = 2;      // SystemDimension
const int NCP  = 21;     // NumberOfControlParameters
const int NSP  = 0;      // NumberOfSharedParameters
const int NISP = 0;      // NumberOfIntegerSharedParameters
const int NE   = 1;      // NumberOfEvents
const int NA   = 4;      // NumberOfAccessories
const int NIA  = 0;      // NumberOfIntegerAccessories
const int NDO  = 0;      // NumberOfPointsOfDenseOutput

int main()
{
    ...
    ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double> ScanKellerMiksis(
        SelectedDevice);
    ...
```

The total number of problems (instances of the Keller–Miksis equation) is the multiplication of the resolutions of the control parameters. The strategy is that a simulation launch performs a bi-parametric scan of the frequency plane f_1 and f_2. Therefore, the number of threads in the Solver Object is $128 * 128 = 16384$. This means that one needs 4 launches to complete the whole simulations. The data of each launch are stored in separate text files named according to the actual values of the pressure amplitudes P_{A1} and P_{A2}.

The system dimension is obviously $N_{SD} = 2$ as the Keller–Miksis oscillator is a second order equation. The `NumberOfControlParameters` (here they are the system coefficients and **not** the real physical control parameters) is set to $N_{CP} = 21$, which needs some explanation. Although the number of the system coefficients are 13, we extended the stored parameters also with the reference time t_{ref} and the reference length R_{ref} of the dimensionless system (to be able to retrieve the quantities in SI units) and with the 6 real physical parameters. The purpose is purely a matter of convenience when we would like to store data into the text files. Such an extended parameter space have a very marginal impact on the performance since only the original 13 number of coefficients are used and loaded from the Global Memory.

Although the system coefficients C_{10} and C_{12} could have been defined as shared parameters, such a distinction between the coefficients are omitted to keep code more simple. Thus the number of the shared parameters is $N_{SP} = 0$. This has no impact on the performance as our kernel function is still compute bound.

The number of events $N_{EF} = 1$, only the local maxima of y_1 is detected. Finally, the number of the user programmable parameters is $N_{ACC} = 4$: the local maxima and minima, and their corresponding time instances are stored (y_1^{max}, τ^{max}, y_1^{min} and τ^{min}). However, only y_1^{max} and y_1^{min} are written into the text files.

13.3.3 The pre-declared user functions of the system

In this section, only some of the pre-declared user functions are shown, which provides new insight into the implementation possibilities. The complete system definition file can be found in the corresponding directory of the tutorial.

The implemented right-hand side is as follows

```
__device__ void PerThread_OdeFunction(int tid, int NT, double* F, double* X, double T,
    double* cPAR, double* sPAR, int* sPARi, double* ACC, int* ACCi)
{
   double rx1 = 1.0/X[0];
   double p   = pow(rx1, cPAR[10]);

   double s1;
   double c1;
   sincospi(2.0*T, &s1, &c1);

   double s2 = sin(2.0*cPAR[11]*PI*T+cPAR[12]);
   double c2 = cos(2.0*cPAR[11]*PI*T+cPAR[12]);

   double N;
   double D;
   double rD;

   N = (cPAR[0]+cPAR[1]*X[1])*p - cPAR[2]*(1.0+cPAR[9]*X[1]) - cPAR[3]*rx1 - cPAR[4]*X
       [1]*rx1 - 1.5*(1.0-cPAR[9]*X[1]*(1.0/3.0))*X[1]*X[1] - ( cPAR[5]*s1 + cPAR[6]*s2 )
        * (1.0+cPAR[9]*X[1]) - X[0]*( cPAR[7]*c1 + cPAR[8]*c2 );
   D = X[0] - cPAR[9]*X[0]*X[1] + cPAR[4]*cPAR[9];
   rD = 1.0/D;


   F[0] = X[1];
   F[1] = N*rD;
}
```

Observe that only the first 13 `cPAR` variable is used. Observe also that the frequently used $1/x_1$ is pre-computed (`rx1`).

The accessories are initialised with the initial conditions at the beginning of every integration phase:

```
__device__ void PerThread_Initialization(int tid, int NT, double T, double &dT, double*
    TD, double* X, double* cPAR, double* sPAR, int* sPARi, double* ACC, int* ACCi)
{
   ACC[0] = X[0];
   ACC[1] = T;
   ACC[2] = X[0];
   ACC[3] = T;
}
```

Only the local minima are updated after every successful time step as the local maxima (beginning of the collapse phase) is already properly set during the initialisation:

```
__device__ void PerThread_ActionAfterSuccessfulTimeStep(int tid, int NT, double T, double
    dT, double* TD, double* X, double* cPAR, double* sPAR, int* sPARi, double* ACC, int*
    ACCi)
{
   if ( X[0]<ACC[2] )
   {
      ACC[2] = X[0];
      ACC[3] = T;
   }
}
```

In general, the dual-frequency excitation can be quasiperiodic (or at least nearly quasiperiodic due to the finite resolution of the frequencies). Therefore, the proper track of the time domain is crucial. The end of

the time domain is set to a very high number; namely, 10^{10}, the simulation will be stopped by the event. At the end of each integration phase, the beginning of the time domain must be properly adjusted as follows

```
__device__ void PerThread_Finalization(int tid, int NT, double T, double dT, double* TD,
    double* X, double* cPAR, double* sPAR, int* sPARi, double* ACC, int* ACCi)
{
   TD[0] = T;
}
```

This means that the beginning of the next integration phase will be the end of the previous one.

13.3.4 Results

In Fig. 13, the relative expansion ratio y_{exp} is presented via four bi-parametric contour plots. The colour code indicates the magnitude of the relative expansion saturated at $y_{exp} = 5$. The higher the value of y_{exp} the stronger the bubble collapse (red domains). The control parameters in each subplot are the excitation frequencies ω_1 and ω_2. The pressure amplitude combinations are highlighted in the labels of the axes. At a given parameter set, the largest value out of the 64 stored relative expansions is depicted. The total simulation time of the 65536 number of Keller–Miksis equation is 2.7 min on our Titan Black card (saving of the data to text files is included). This number is good as the minimum time step during the integration can be as small as 10^{-12} for some systems (mainly in the red regions in Fig. 13). The employed numerical scheme was the Runge–Kutta–Kash–Carp. Both the absolute and relative tolerances were 10^{-10}.

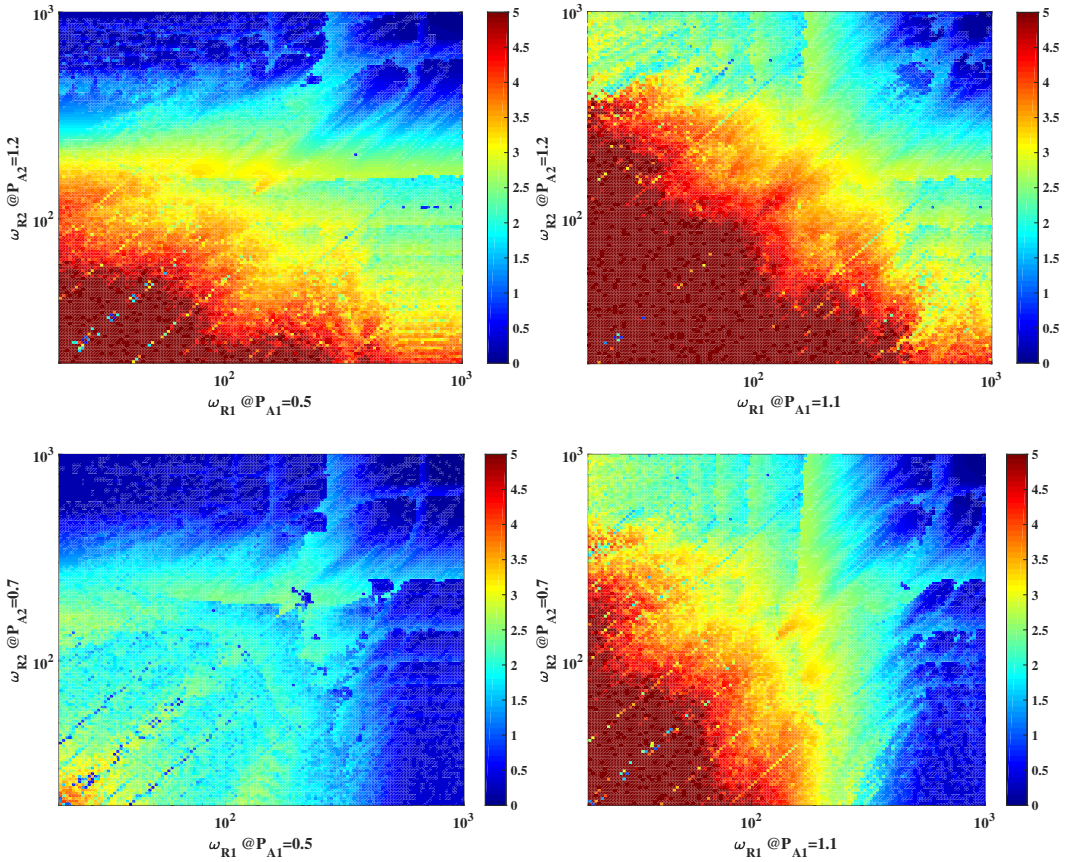

Figure 13: Four dimensional parameter scan of the relative expansion of an oscillating single spherical bubble.

13.4 Tutorial 5: impact dynamics

The fifth tutorial example describes the behaviour of a pressure relief valve which can exhibit impact dynamics. The dimensionless governing equations are

$$\dot{y}_1 = y_2, \tag{28}$$
$$\dot{y}_2 = -\kappa y_2 - (y_1 + \delta) + y_3, \tag{29}$$
$$\dot{y}_3 = \beta(q - y_1\sqrt{y_3}), \tag{30}$$

where y_1 and y_2 are the displacement and velocity of the valve body, respectively. y_3 is the pressure inside the reservoir chamber to where the pressure relief valve is connected. The fixed parameters in the system are as follows: $\kappa = 1.25$ is the damping coefficient, $\delta = 10$ is the precompression parameter and $\beta = 20$ is the compressibility parameter. The control parameter during the simulations is the dimensionless flow rate q.

In Eqs (28)-(30), 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

$$y_1^+ = y_1^- = 0, \tag{31}$$
$$y_2^+ = -r y_2^-, \tag{32}$$
$$y_3^+ = y_3^- \tag{33}$$

is applied. The Newtonian coefficient of restitution $r = 0.8$ approximates the loss of energy of the impact. It shall be shown that by applying multiple event handling together with a special "action function" call at the impact detection, system (28)-(30) can be handled very efficiently.

13.4.1 The objectives

The objective of this tutorial example is very simple, generate a bifurcation diagram, where the maximum and the minimum displacement of the valve body y_1^{max} and y_1^{min} is plotted as a function of the low rate q. Since the system is autonomous, an event handling is necessary to detect the Poincaré section defined as the local maxima of y_1 at $y_2^+ = 0$. Moreover, another event handling is necessary to handle the impact at $y_1 = 0$. One integration phase means the integration of the systems from Poincaré section to Poincaré section. The sole control parameter is the flow rate q varied between 0.2 and 10 with a resolution of 30720. In case of impact, the minimum value of the displacement will contain the value of $y_1^{min} = 0$. 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. The first, second and third columns in the file are related to the control parameter q, y_1^{max} and y_1^{min}, respectively.

13.4.2 Configuration of the Solver Object

The configuration of the Solver Object is as follows:

```
// Solver Configuration
#define SOLVER RKCK45 // RK4, RKCK45
const int NT   = 30720; // NumberOfThreads
const int SD   = 3;     // SystemDimension
const int NCP  = 1;    // NumberOfControlParameters
const int NSP  = 4;     // NumberOfSharedParameters
const int NISP = 0;     // NumberOfIntegerSharedParameters
const int NE   = 2;     // NumberOfEvents
const int NA   = 2;     // NumberOfAccessories
const int NIA  = 0;     // NumberOfIntegerAccessories
const int NDO  = 0;     // NumberOfPointsOfDenseOutput
```

```
int main()
{
   int NumberOfFlowRates = NT;
   ...
   ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double> ScanPressureReliefValve(
       SelectedDevice);
   ...
```

Here the total number of problems and the number of threads in the Solver Object are equal (the resolution of the control parameter q). Therefore, there is only one simulation launch. The system dimension is $N_{SD} = 3$ according to the model of the pressure relief valve. The number of the control parameters is only $N_{CP} = 1$ (flow rate q). However, there are 4 shared parameters: κ, δ, β and r. For the computations, two event functions are needed. One for defining a Poincaré section and one for detecting the impact. Finally, two user programmable parameters (`Accessories`) are used to store the values of y_1^{max} and y_1^{min}.

13.4.3 The pre-declared user functions of the system

In this section, only some of the pre-declared user functions are shown, which provides new insight into the implementation possibilities. The complete system definition file can be found in the corresponding directory of the tutorial.

The implemented right-hand side is as follows

```
__device__ void PerThread_OdeFunction(int tid, int NT, double* F, double* X, double T,
    double* cPAR, double* sPAR, int* sPARi, double* ACC, int* ACCi)
{
   F[0] = X[1];
   F[1] = -sPAR[0]*X[1] - (X[0]+sPAR[1]) + X[2];
   F[2] = sPAR[2]*( cPAR[0] - X[0]*sqrt(X[2]) );
}
```

Observe that 3 out of the 4 parameters are loaded from the Shared Memory.

The definition of the two event functions are very straightforward:

```
__device__ void PerThread_EventFunction(int tid, int NT, double* EF, double* X, double T,
     double* cPAR, double* sPAR, int* sPARi, double* ACC, int* ACCi)
{
   EF[0] = X[1]; // Poincare section
   EF[1] = X[0]; // Impact detection
}
```

The direction of both event detection is -1 and for the Poincaré section, we have to stop the simulation, see the system definition file.

In case of impact detection, one has to define a proper "action" (applying the impact law) after every successful event detection. The corresponding pre-declared user function is written as

```
__device__ void PerThread_ActionAfterEventDetection(int tid, int NT, int IDX, int CNT,
    double &T, double &dT, double* TD, double* X, double* cPAR, double* sPAR, int* sPARi,
     double* ACC, int* ACCi)
{
   if ( IDX == 1 )
      X[1] = -sPAR[3] * X[1];
}
```

Observe how the velocity y_2 is reversed by applying the impact law Eq. (32). The coefficient of restitution is load from the Shared Memory.

For the details of the rest of the pre-declared user functions, the reader is referred to the system definition file of this tutorial example.

13.4.4 Results

The maxima y_1^{max} (black dots) and minima y_1^{min} (red dots) of the displacement of the valve body is depicted in Fig. 14 as a function of the dimensionless flow rate q spanned between 0.2 and 10. The maxima are simply the Poincaré sections. The minima, however, are good indicators to show the range of parameters (approximately between $q = 0.2$ and $q = 7.5$) where impact dynamics occur ($y_1^{min} = 0$). The total simulation time is approximately 9.17 s (writing the data into the text file is again included), while the trasient iterations needed 7.11 s. The employed numerical scheme was the Runge–Kutta–Kash–Carp. Both the absolute and relative tolerances were 10^{-10}.

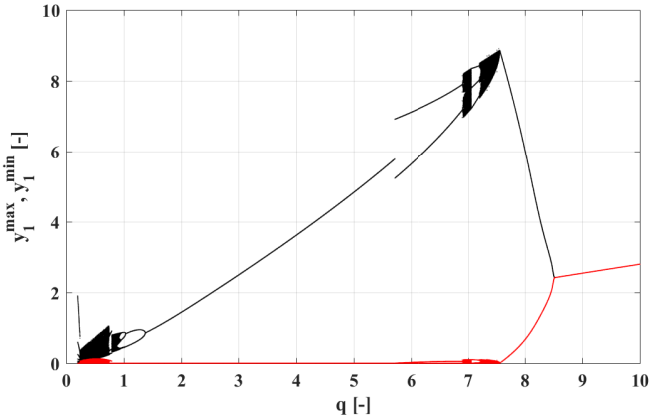

Figure 14: The maximum (black) and minimum (red) values of the valve position y_1 as a function of the dimensionless flow rate q for the pressure relief valve. The damping coefficient is $\kappa = 1.25$, the precompression parameter is $\delta = 10$ and the compressibility parameter is $\beta = 20$. The number of the used threads and the resolution of the control parameter is $N_T = 30720$.

13.5 Tutorial 6: overlap CPU and GPU computations (double buffering)

As it is mentioned in Sec. 9, CUDA offer asynchronous behaviour of the GPU related operations (run solver and/or memory copy between GPU and CPU) with respect to the CPU. This means that after dispatching a certain task to the GPU, the control immediately returns back to the CPU and it is free to do any other task (performing its own computations or dispatching another task to a GPU). In order to fully control the progress of the CPU threads and the GPU working queues, synchronisation possibilities are incorporated to the Solver Object. All these operations are managed by the Solver Object with a handful of member functions. Thus, the user do not have to perform GPU programming and do not need detailed knowledge of the GPU architecture.

Nevertheless, in order to clearly understand the behaviour of the asynchronous instructions, some brief introduction to CUDA streams is necessary. All the GPU related tasks e.g., the kernel launches (running the solver) or the memory copy operations are queued into a so-called CUDA stream. In a single stream, the associated tasks are executed one after another. Therefore, using only one stream, GPU related tasks cannot be overlapped, e.g., running the solver while copy back data to the CPU side of a previous simulation. In case of asynchronous dispatch of the tasks, the CPU can fill up a CUDA stream even with hundreds of tasks that will be completed by the GPU one after another. In this sense, a CUDA stream has synchronous behaviour with respect to the GPU, but asynchronous behaviour with respect to the CPU (unless explicit synchronisation is placed somewhere in the code, see again Sec. 9).

In a single GPU, there can be multiple active CUDA streams (up to 16 in the Kepler architecture), each can be filled up with tasks in any order by the CPU. From compute capabilities CC 3.5 (Kepler architecture) or higher, the tasks in different CUDA streams can be executed concurently if there are enough hardware resources. Nevertheless, the tasks are started to be executed in the order they are dispached by the CPU to a GPU regardless of the coresponding CUDA stream.

To each Solver Object, a device number is immediately associated during its declaration, see Sec. 9. Similarly, a CUDA stream is also created and associated to a Solver Object inside its constructor. Any GPU related actions via its member function will use its own CUDA stream to queue the tasks to the associated GPU.

13.5.1 The objectives and the workflow

The main objective of this tutorial example is to revise the reference example and try to overlap as many computations between the CPU and GPU as possible. Keep in mind that the total number of problems need to be solved is 46080 which was divided into two launches with $N_T = 23040$ using a single GPU and a single Solver Object. Here, two Solver Objects are employed for overlapping computations. The fill up procedure is wrapped into tutorial specific functions, for details see the code in `DoubleBuffering.cu` in the folder of the sixth tutorial example. Since the system definition is identical with the reference case, its discussion is omitted here.

First, the two Solver Objects are created as follows:

```
// Solver Configuration
#define SOLVER RKCK45 // RK4, RKCK45
const int NT   = 46080/2; // NumberOfThreads
const int SD   = 2;       // SystemDimension
const int NCP  = 1;       // NumberOfControlParameters
const int NSP  = 1;       // NumberOfSharedParameters
const int NISP = 0;       // NumberOfIntegerSharedParameters
const int NE   = 2;       // NumberOfEvents
const int NA   = 3;       // NumberOfAccessories
const int NIA  = 0;       // NumberOfIntegerAccessories
const int NDO  = 200;   // NumberOfPointsOfDenseOutput

int main()
{
   int NumberOfProblems = NT*2;
   int NumberOfThreads  = NT;
```

```
    int MajorRevision  = 3;
    int MinorRevision  = 5;
    int SelectedDevice = SelectDeviceByClosestRevision(MajorRevision, MinorRevision);

    ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double> ScanDuffing1(
        SelectedDevice);
    ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double> ScanDuffing2(
        SelectedDevice);
    ...
}
```

Observe that the same GPU (closet to CC 3.5) is accosiated to the Solver Objects.

The main part of the control flow of the program is to fill up the Solver Object with the rest of the data (time domain, initial conditions, control parameters and the initial values of the accessories), perform 1024 transient iterations (integration phases) and finally iterate further 32 times and save some data to disc. Meanwhile try to overlap as much computation as possible. This part of the code is listed below.

```
int NumberOfSimulationLaunches = NumberOfProblems / NumberOfThreads / 2;

int FirstProblemNumber;
for (int LaunchCounter=0; LaunchCounter<NumberOfSimulationLaunches; LaunchCounter++)
{
    FirstProblemNumber = LaunchCounter * (2*NumberOfThreads);

    FillSolverObjects(ScanDuffing1, Parameters_k_Values, Parameters_B,
        InitialConditions_X1, InitialConditions_X2, FirstProblemNumber, NumberOfThreads);
    ScanDuffing1.SynchroniseFromHostToDevice(All);
    ScanDuffing1.Solve();
    ScanDuffing1.InsertSynchronisationPoint();

    FirstProblemNumber = FirstProblemNumber + NumberOfThreads;

    FillSolverObjects(ScanDuffing2, Parameters_k_Values, Parameters_B,
        InitialConditions_X1, InitialConditions_X2, FirstProblemNumber, NumberOfThreads);
    ScanDuffing2.SynchroniseFromHostToDevice(All);
    ScanDuffing2.Solve();
    ScanDuffing2.InsertSynchronisationPoint();

    for (int i=0; i<1024; i++)
    {
        ScanDuffing1.SynchroniseSolver();
        ScanDuffing1.Solve();
        ScanDuffing1.InsertSynchronisationPoint();

        ScanDuffing2.SynchroniseSolver();
        ScanDuffing2.Solve();
        ScanDuffing2.InsertSynchronisationPoint();
    }

    for (int i=0; i<31; i++)
    {
        ScanDuffing1.SynchroniseSolver();
        ScanDuffing1.SynchroniseFromDeviceToHost(All);
        ScanDuffing1.Solve();
        ScanDuffing1.InsertSynchronisationPoint();
        SaveData(ScanDuffing1, DataFile, NumberOfThreads);

        ScanDuffing2.SynchroniseSolver();
        ScanDuffing2.SynchroniseFromDeviceToHost(All);
        ScanDuffing2.Solve();
        ScanDuffing2.InsertSynchronisationPoint();
        SaveData(ScanDuffing2, DataFile, NumberOfThreads);
    }

    ScanDuffing1.SynchroniseSolver();
```

```
    ScanDuffing1.SynchroniseFromDeviceToHost(All);
    SaveData(ScanDuffing1, DataFile, NumberOfThreads);

    ScanDuffing2.SynchroniseSolver();
    ScanDuffing2.SynchroniseFromDeviceToHost(All);
    SaveData(ScanDuffing2, DataFile, NumberOfThreads);
}
```

Observe that compared to the reference case, the number of simulation launches are halved due to the usage of two Solver Objects. More precisely, only a single launch is enough, as each Solver Object is responsible for a half of the total number of problems (instances of the ODE). The tutorial specific function called `FillControlParameters()` fills up the passed Solver Object with the aforementioned data. After finishing this task on the first Solver Object (`ScanDuffing1`), the CPU immediately put the following tasks to the associated CUDA stream asynchronously: a) copy the data from the System Memory to the Device Memory, b) perform an integration phase and c) insert a synchronisation point to be able to check whether the simulation of the first integration phase is completed or not. Since all these operations are asynchronous, while the GPU is working on the above initiated tasks, the CPU can proceed further and fill up the second Solver Object (`ScanDuffing2`) with the rest of the data. Moreover, after that, the CPU can again immediately initiate the copy process, the integration phase and can insert a synchronisation point.

The next loop iterates over the initial transient performing integration phases one after another. The technique to dispatch the tasks are very similar. The difference is that data copy between the host side and the device side is not necessary as they are not used during the transient phase. Moreover, synchronisation takes place before the initiation of another integration phase, and a synchronisation point is inserted right after the initiation of the integration (to be able to synchronise it in the next cycle). This is done to avoid overfilling of the CUDA streams of the Solver Objects. Observe that inside the loop there are only asynchronous GPU related function calls. Therefore, after every function call the CPU immediate calls the next one almost without no delay. Without synchronisation, this would result in CUDA streams each filled immediately with 1024 number of kernel launches (integration phases)[10].

Now the control flow inside the last loop should be self-explanatory. Keep in mind, however, that the copy process is initiated via the function call `SynchroniseFromDeviceToHostAsync()` before initiating another integration phase. After that, with the tutorial specific function `SaveData()`, some of the calculated data are stored to disc. It is important to note that the first saving process is performed on the data corresponding to the last integration phase inside the previous transient loop since all the preceding GPU related function calls are asynchronous. That is, the first integration phase inside the loop is initiated only, and the `SaveData()` function is immediately called without waiting for the computation to be completed. This is the reason that the second loop has an iteration number of 31 instead of 32.

Finally, the last integration phase of both Solver Object have to be synchronised, the data have to copied back to the host side and saved to disc.

13.5.2 Results

The results of this tutorial example is exactly the same as the reference example discussed thoroughly in the main part of this manual. Only the total runtime is different. Using the Nvidia GeForce GTX Titan Black graphics card, the total simulation of the reference case is 11.72 s. This simulation time is reduced to 11.09 s by employing the double buffering technique introduced in this tutorial example. The small performance increase is due to much higher required time to save data to disc than to perform an integration phase. Therefore, only a fraction of the saving process can be overlapped. However, this is a perfect example how to overlap CPU and GPU computations.

[10] Personally, I could not find any information how much tasks can be queued into a CUDA stream. But to avoid possible runtime errors, the synchronisation points are inserted, even if they would not strictly be necessary.

13.6 Tutorial 7: using multiple GPUs in a single node

The main purpose of this seventh tutorial example is to introduce the usage of multiple GPUs in a single node. That is, the GPUs must reside physically inside a single machine. But, there is no restriction for the number of the GPUs. During this section, the reference example is revised using two GPUs (Nvidia Tesla K20m) and using two Solver Objects each is associated to different GPUs, see the code snippet below

```
// Solver Configuration
#define SOLVER RKCK45 // RK4, RKCK45
const int NT    = 23040; // NumberOfThreads
const int SD    = 2;     // SystemDimension
const int NCP   = 1;     // NumberOfControlParameters
const int NSP   = 1;     // NumberOfSharedParameters
const int NISP  = 0;     // NumberOfIntegerSharedParameters
const int NE    = 2;     // NumberOfEvents
const int NA    = 3;     // NumberOfAccessories
const int NIA   = 0;     // NumberOfIntegerAccessories
const int NDO   = 200;   // NumberOfPointsOfDenseOutput

int main()
{
   int NumberOfProblems = NT*2;
   int NumberOfThreads  = NT;

   ListCUDADevices();
   int SelectedDevice1 = 0; // According to the output of ListCUDADevices();
   int SelectedDevice2 = 2; // THEY MUST BE SET ACCORDING TO YOUR CURRENT CONFIGURATION
       !!!

   ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double> ScanDuffing1(
       SelectedDevice1);
   ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,double> ScanDuffing2(
       SelectedDevice2);
   ...
}
```

The different device numbers are specified manually according to the output of the function `ListCUDA Devices()` called in advance solely.

During the computation similar techniques have to be used already introduced in the sixth tutorial example. Therefore, if the reader is not familiar with the asynchronous behaviour of the member functions of the Solver Objects, He/She is referred back to the sixth tutorial example discussed in Sec. 13.5. The control flow is also very similar to the one shown there, see the following listing

```
int NumberOfSimulationLaunches = NumberOfProblems / NumberOfThreads / 2;

int FirstProblemNumber;
for (int LaunchCounter=0; LaunchCounter<NumberOfSimulationLaunches; LaunchCounter++)
{
   FirstProblemNumber = LaunchCounter * (2*NumberOfThreads);

   FillSolverObjects(ScanDuffing1, Parameters_k_Values, Parameters_B,
       InitialConditions_X1, InitialConditions_X2, FirstProblemNumber, NumberOfThreads);
   ScanDuffing1.SynchroniseFromHostToDevice(All);
   ScanDuffing1.Solve();
   ScanDuffing1.InsertSynchronisationPoint();

   FirstProblemNumber = FirstProblemNumber + NumberOfThreads;

   FillSolverObjects(ScanDuffing2, Parameters_k_Values, Parameters_B,
       InitialConditions_X1, InitialConditions_X2, FirstProblemNumber, NumberOfThreads);
   ScanDuffing2.SynchroniseFromHostToDevice(All);
   ScanDuffing2.Solve();
   ScanDuffing2.InsertSynchronisationPoint();

   for (int i=0; i<1023; i++)
```

```
    {
        ScanDuffing1.SynchroniseSolver();
        ScanDuffing1.Solve();
        ScanDuffing1.InsertSynchronisationPoint();

        ScanDuffing2.SynchroniseSolver();
        ScanDuffing2.Solve();
        ScanDuffing2.InsertSynchronisationPoint();
    }

    ScanDuffing1.SynchroniseSolver();
    ScanDuffing2.SynchroniseSolver();
    for (int i=0; i<32; i++)
    {
        ScanDuffing1.Solve();
        ScanDuffing1.SynchroniseFromDeviceToHost(All);
        ScanDuffing1.InsertSynchronisationPoint();

        ScanDuffing2.Solve();
        ScanDuffing2.SynchroniseFromDeviceToHost(All);
        ScanDuffing2.InsertSynchronisationPoint();

        ScanDuffing1.SynchroniseSolver();
        SaveData(ScanDuffing1, DataFile, NumberOfThreads);

        ScanDuffing2.SynchroniseSolver();
        SaveData(ScanDuffing2, DataFile, NumberOfThreads);
    }
}
```

The main difference is that here there is no overlapping computations with the CPU. After dispatching integration phases to both GPUs, they are synchronised (both) before proceeding further with any other tasks. Therefore, the main gain here is the parallel computation of each half of the total number of problems distributed to different GPUs. In order to overlap CPU-GPU computations, at least two Solver Objects have to be associated to each GPU similarly as in case of the previous tutorial example. This means at least 4 Solver Objects in this specific example. It is left to the reader to assemble a control flow for such a problem. However, keep in mind that the total number of problems has to be divided up to 4 different parts.

13.6.1 Results

Again the results of this tutorial example is exactly the same as the reference example. Therefore, only the runtimes are discussed here. Using a single Nvidia Tesla K20m graphics card for the reference case, the total simulation time is 14.50 s while the time needs for the transients is 5.77 s. The runtimes are reduced to 11.70 s and 3.02 s using two Nvidia Tesla K20m graphics cards. Observe that the transient part scales almost linearly with the number of the GPUs.