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

Massively Parallel GPU-ODE Solver (MPGOS)

MODULES
Single Sytem Per-Thread v3.1
Coupled Systems Per-Block v1.0

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

Ferenc Hegedűs

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

September 18, 2020

Contents

Chapter 1

Introduction and Basics

Program code MPGOS is written in C++ and 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 are

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

The directory `Doc/` contains the documentation of the program package. In the directory `SourceCodes/`, there are the source codes of the solver including the interface one needs to be included for the usage of the program package. Inside the folders `Tutorials_SingleSystem_PerThread/` and `Tutorials_Coupled Systems_PerBlock/`, there are collections of tutorial examples help the user getting started with the modules of MPGOS.

For a minimalist project, the user has to prepare three files:

- `MainCode.cu`
- `SystemDefinition.cuh`
- `makefile` (for Linux) or `make.bat` (for Windows)

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 classes 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. Their name and definition are pre-declared and should NOT be modified, and they must be visible in the main C++ program. However, these functions can be collected in a header file having an arbitrary name, e.g., `SystemDefinition.cuh`, which must be included in the main C++ file. Alternatively, these device functions can be copied immediately into the main C++

file omitting the inclusion via a header file. The best practice is to take such a file from one of the tutorial examples, rename the file to a desired name and reimplement the function bodies inside. **Again, 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 for managing the compilation process (including, e.g., the specification of the path of the source codes). Every tutorial examples have such a file that can compile the corresponding problem by simply type *make* into the command line (as long as the required compiler is installed). The command *make clean* clears the directory from the files generated during the compilation process and by the program run (executables and text files). Thus use it with caution. The proposed `makefile`s are very simple examples using only the basics of the makefile environment. For more details, the interested user is referred to the following site:

https://www.gnu.org/software/make/manual/

For Windows users, MPGOS provides a batch file called `make.bat` for every tutorial examples to mimic the `makefile` environment. The only difference is that the compilation process is via the command *make.bat*, and there is no cleaning feature (only the already existing executable is overwritten).

To put everything together, to be able to use program package MPGOS, 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 "SystemDefinition.cuh"
#include "SingleSystem_PerThread_Interface.cuh"
```

The first inclusion is the system definition file mentioned previously. The second header file is the interface of the module used (Single System Per-Thread here), which can be found in the folder `SourceCodes/`. That is, 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 set-up a proper path of the `SourceCodes/` folder in the `makefile` or `make.bat`, see the corresponding examples in the tutorials. The file `SingleSystem_PerThread_Interface.cuh` itself includes many other header files from the `SourceCodes/` folder, thus, the whole content must be visible.

The list of the available modules is summarised in Tab. 1.1. The first, second and third columns are the type of the system that can be handled, the main parallelisation strategy and the name of the header file of the interface, respectively. For the details of the capabilities of the different modules, see Sec. 1.4.

Table 1.1: Summary of the available MPGOS modules

system type	parallelisation	header file
Single System	Per-Thread	`SingleSystem_PerThread_Interface.cuh`
Coupled Systems	Per-Block	`CoupledSystems_PerBlock_Interface.cuh`

1.1 Requirements

1.1.1 Hardware requirements

The program package MPGOS is heavily relying on the CUDA C software environment. Thus, the first step should be a quick check for the available CUDA-capable GPUs in your system. If the graphics card(s) is listed in

http://developer.nvidia.com/cuda-gpus,

the GPU is CUDA-capable. In the case of computer clusters (HPC/GPU clusters), please consult with the corresponding operator.

1.1.2 Software requirements (Windows)

- **Microsoft Visual Studio**. CUDA C/C++ provides only a set of extensions to the standard programming language C or C++. Therefore, as a first step, a proper version of the Microsoft Visual Studio is necessary. For the list of compatible versions, see the link below.

  https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html

  Important: Visual Studio does NOT install C++ by default. Thus do not choose the express install option. If you already have a version without C++, rerun the setup, select modify and then tick programming language C++ on. As MPGOS uses features of C++14, a Visual Studio version with capabilities of **C++14 features is necessary**.

  The **Visual Studio Community 2015** edition can be a perfect free choice in case of no subscription available.

- **NVIDIA CUDA Toolkit 9.0** or higher. MPGOS uses C++14 features that are available only in version 9.0 or higher. The CUDA Toolkit installs the compiler **nvcc**, the device driver for the GPU and many other packages (e.g., profiler or sample programs). The toolkit can be downloaded from

  https://developer.nvidia.com/cuda-downloads

- In order to check the proper installation of the above-mention software packages, and to check the proper working of the available GPU devices, at least two tests should be compiled and run from the sample programs NVIDIA provided by the toolkit. Assuming default installation directory and version 10.2, the sample programs can be found in

  ```
  C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2
  ```

  By double-clicking on a Microsoft Visual Studio solution file, it is opened with the corresponding IDE (Integrated Development Environment). With the keyboard combinations `Ctrl+F5`, the program code is built-up and run. The codes worth to check is the `deviceQuery` (lists the CUDA-capable devices and some of their properties) and `bandwidthTest` (check that the system and the CUDA-capable device can communicate correctly). The directories of both sample programs are located in the `1_Utilities` subdirectory. For more details, the interested reader is referred to the official installation guide:

  https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html

- It can happen that the executable `cl.exe` (need by **nvcc**) is not included into the **PATH environment variables**. For Windows 10: Control Panel → System and Security → System → Advanced system settings → Environment Variables. Here, the following items to the environmental variable PATH have to be added (assuming Visual Studio Community 2015, and a 64-bit system; otherwise, the proper paths can be different, but similar):

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64\cl.exe
```

- **Important:** one of the side effect of using windows is that the graphical user interface of the GPU becomes unresponsive, very slow or even crash by executing a code that fully utilizes the graphics card. The reason is that Windows has a feature called **Timeout Detection and Recovery (TDR)** built into the Windows Display Driver Model (WDDM) that watches out for that sort of situation and resets the graphics driver if it happens. The remedy is to increase the length of time TDR kicks in or turn TDR entirely off. Both can be done via registry entries, for the technical details see:

  https://www.pugetsystems.com/labs/hpc/Working-around-TDR-in-Windows-for-a-better-GPU-computing-experience-777/

1.1.3 Software requirements (Linux)

For Linux systems, it is crucial to follow precisely the steps provided by the installation guides of the CUDA Toolkit documentation:

https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html

Here we summarise the main steps only briefly for Ubuntu. Especially for HPC/GPU clusters, please ask the corresponding IT manager for further help.

- **Supported Linux Distribution**. As the CUDA development environment relies on tight integration with the host development environment, including the host compiler and C runtime libraries, only specific versions of Linux distributions are supported. For a complete list, see the provided link above.
- **C++14 Compiler**. CUDA C/C++ provides only a set of extensions to the standard programming language C or C++. In addition, MPGOS uses features of C++14. Therefore, the installation of a C++14 capable compiler is necessary (e.g., **gcc 6.1 or above**). The installation of the latest version of gcc on Ubuntu system can be done by the following command (superuser privileges is usually necessary)

  ```
  sudo apt install g++
  ```

- **Lunux Kernel Headers**. Before the CUDA Toolkit installation, the kernel headers and the corresponding development packages for the running version of the Linux kernel must be installed. This can be done on Ubuntu by the command

  ```
  sudo apt-get install linux-headers-\$(uname-r)
  ```

- **NVIDIA CUDA Toolkit 9.0** or higher. MPGOS uses C++14 features that are available only in version 9.0 or higher. The CUDA Toolkit installs the compiler **nvcc**, the device driver for the GPU and many other packages (e.g., profiler or sample programs). The toolkit can be downloaded from

  https://developer.nvidia.com/cuda-downloads

  After the selection of the proper architecture, distribution and version of the system, the further installation instructions are also listed on the website.

- As a last action, the proper setup of the **PATH environment variables** are also advisable (e.g., in a *.bashrc* or *.profile*). The following listing shows an example for such a set up for CUDA Toolkit 9.0, which is a part of a *.bashrc* file (the `CUDA_HOME` folder can be queried via the command `which nvcc` if a proper CUDA Toolkit has already been installed):

  ```
  export CUDA_HOME=/usr/local/cuda-9.0
  export LD_LIBRARY_PATH=${CUDA_HOME}/lib64
  PATH=${CUDA_HOME}/bin:${PATH}
  export PATH
  ```

```
$ (do not bother this character! It is a LaTeX glich!)
```

1.1.4 Software requirements (Mac)

Unfortunately, for Mac OS system, we have no experience; therefore, please follow the steps provided by the official CUDA installation guide:

https://docs.nvidia.com/cuda/cuda-installation-guide-mac-os-x/index.html

1.1.5 Programming experinece

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 tutorial examples provided by the program package and 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. 4 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 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. 1.1.3. In this section, a BASH command shell is used. 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 installed:

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

It downloads the package to the current directory. Next, go, e.g., to the directory of the first tutorial example in which the files of 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
```

The command

```
make clean
```

erases the created executable (here Reference.exe) and all the files (*.txt) produced during program execution.

1.2.1 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. 1.1. 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.

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. 1.2.

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 → Editors menu, the user can specify and add editors (I prefer **Notepad++**), see Fig. 1.3. 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.

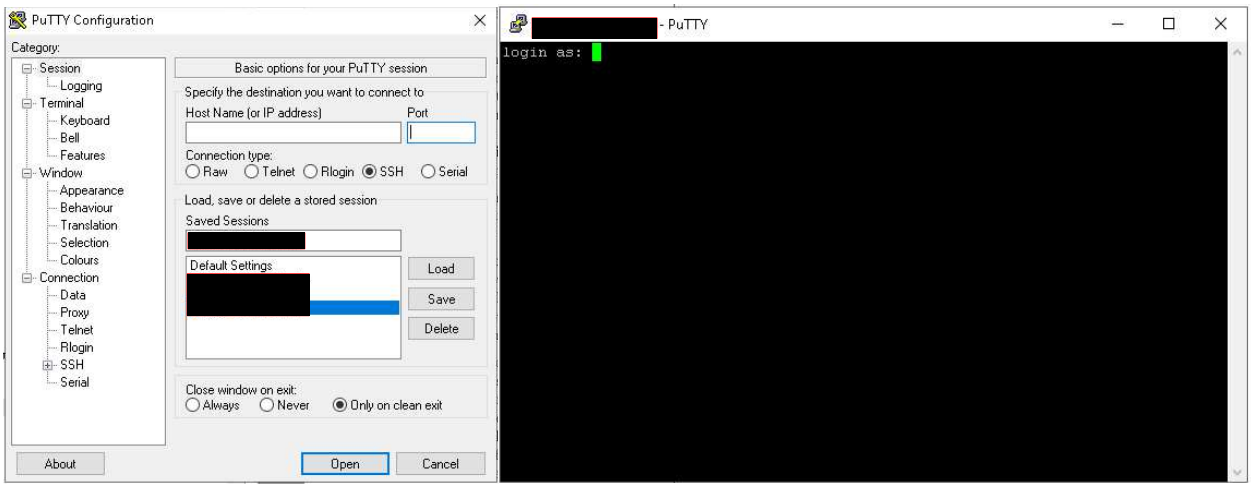

Figure 1.1: Login with the SSH client PuTTY.

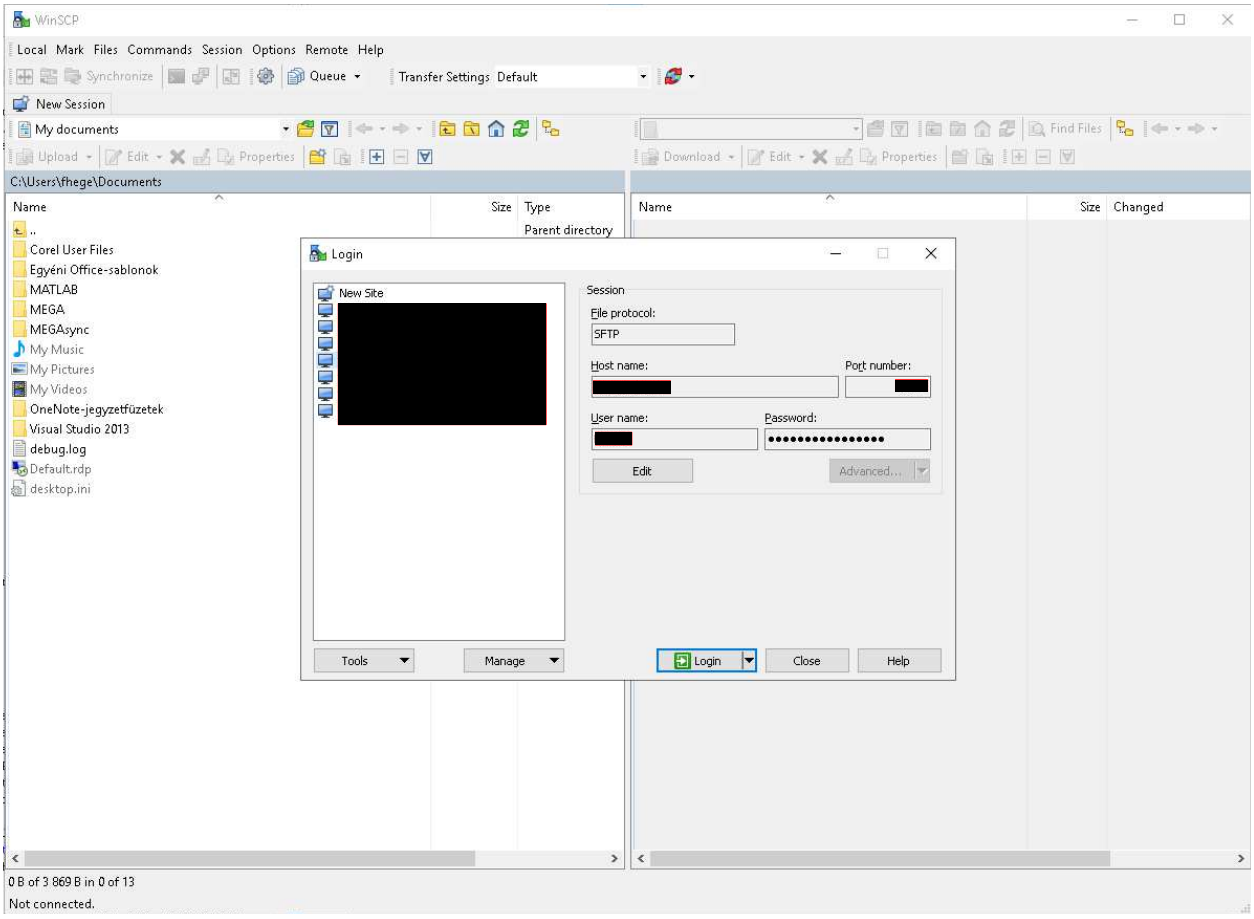

Figure 1.2: Login with the tool WinSCP for file exchange and editing files.

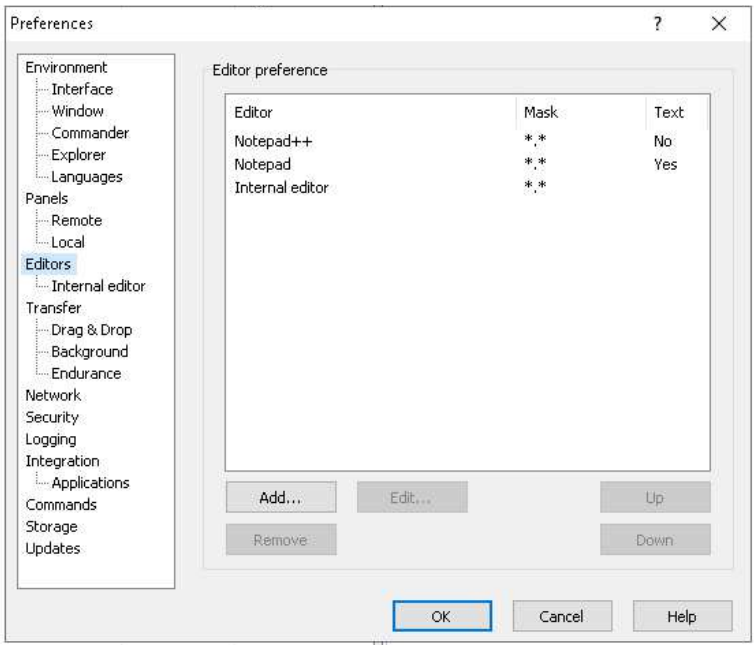

Figure 1.3: Changing the preferred editor in WinSCP under the menu Option → Preferences.

1.3 Quick start on Windows

Here it is assumed that the user has a properly-installed Visual Studio and CUDA Toolkit with a properly set **PATH** environment variables for the compilers, see Sec. 1.1.2. Although Visual Studio must already been installed, we are going to introduce a minimalist way to run the tutorial examples of MPGOS. The reason is that the proper set up of, e.g., the include libraries can be a cumbersome task, and it is left to the user to figure out these specifics for his/her preferred IDE.

As usual, 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. Next, go, e.g., to the directory of the first tutorial example in which the files for this reference test case reside:

```
cd D:\...\Massively-Parallel-GPU-ODE-Solver\Tutorials_SingleSystem_PerThread\
T1_Reference/
```

MPGOS offers a batch file for the compiling process that can be used similarly as the makefile environment under Linux. In order to create and run the executable simply type

```
make.bat
Reference.exe
```

As the folder of the downloaded material is under the control of the user, in the file **make.bat**, the proper path of the folder `SourceCodes/` has to be updated. The batch file make.bat overwrites the already existing executable. In contrast to the Linux makefile environment, here there is NO option to clear the directory through the make.bat file.

1.4 Overview of the capabilities of the modules

In this section, a brief description of the capabilities and purpose of the different MPGOS modules is provided. It serves to be able to select the proper module for a specific problem. For the details, e.g., the available solvers, the reader is referred to the corresponding section in Chap. 2. We shall see in the forthcoming Chapters that **MPGOS is an efficient tool to investigate dynamical systems rather than a "plain" ODE solver**.

In general, regardless of the module used, 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 a very similar 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 Chap. 4.

1.4.1 Module: Single System Per-Thread

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

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, t; \boldsymbol{p}) = \boldsymbol{f}(\boldsymbol{x}, t; \boldsymbol{p_{cp}}, \boldsymbol{p_{sp}}; \boldsymbol{p_{ac}}), \tag{1.1}$$

where $\boldsymbol{x}$ is the vector of the state variables, t is the time and $\boldsymbol{p}$ is the vector of the parameters. The dot stands for the derivative with respect to time. It is important that the vector of functions $\boldsymbol{f}$ 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 utilise 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 this MPGOS module can efficiently handle. The parallelisation strategy is very simple here: a single instance of Eq. (1.1) is solved by a single GPU thread.

It is important to note that parameters are divided into three subcategories called *control parameters* $\boldsymbol{p_{cp}}$, *shared parameters* $\boldsymbol{p_{sp}}$ and user programmable parameters called *accessories* $\boldsymbol{p_{ac}}$. The sets of *control parameters* are different from instance-to-instance. 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. (1.1); 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 Chap. 4. This can be extremely important in memory bandwidth limited applications. There are **two types** of *shared parameters*: **integer** (store, e.g., common indices) and **floating point** type (the precision is under the control of the user).

The last kind of parameters are called *accessories* which are multi-purpose (user programmable) parameters. They are not strictly a parameter of Eq. (1.1) 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. (1.1), 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 requirement for storing a dense output of every instances. This can be crucial for high performance. Throughout the tutorial examples in Chap. 5, the efficient use and the capabilities of the *accessories* are emphasized. There are **two types** of accessories as well: **integer** and **floating point**.

1.4.2 Module: Coupled Systems Per-Block

This module is designed to solve systems composed by subsystems (called units here) coupled globally or diffusively in a 1D grid. Each subsystem can have an arbitrary number of components (first-order ODEs), but

each component must have the same form. That is, the system is composed of several instances of the same subsystem but with different parameter sets and/or initial conditions. Therefore, huge **parameter studies**, the examination of **multistability** or the investigation of the **domain of attraction** of systems composed by coupled units are among the most suitable situations this MPGOS module can efficiently handle.

The structure of a coupling is as follows:

$$K_i^{(k)} = \sigma^{(k)} G_i^{(k)} \odot \left(A_{i,j}^{(k)} \, H_j^{(k)} \right), \tag{1.2}$$

where $A_{i,j}$ is the coupling matrix, H_j is the vector of the coupling terms, G_i is the vector of the coupling factors, σ is the coupling strength (scalar) and K_i is the vector of the coupling values. The $\odot$ stands for the element-wise (Hadamard or Schur) product of two vectors. The index k denotes the serial number of a coupling (there is no restriction for the number of couplings). After performing the matrix-vector and vector-vector operations in Eq. (1.2), the coupling values K_i are added to a user-specified component of the subsystems.

The general form of a system reads as

$$\begin{aligned}
\text{unit 1} \begin{cases} \dot{x}_{1,1} &= f_1(x_{1,j}, t; \boldsymbol{p}) &+K_1^{(1)} &+K_1^{(2)} &+\cdots \\ \dot{x}_{1,2} &= f_2(x_{1,j}, t; \boldsymbol{p}) &+K_1^{(m)} &+K_1^{(m+1)} &+\cdots \\ \cdots & & & & \\ \dot{x}_{1,n} &= f_n(x_{1,j}, t; \boldsymbol{p}) &+K_1^{(p)} &+K_1^{(p+1)} &+\cdots \end{cases} \\
\text{unit 2} \begin{cases} \dot{x}_{2,1} &= f_1(x_{2,j}, t; \boldsymbol{p}) &+K_2^{(1)} &+K_2^{(2)} &+\cdots \\ \dot{x}_{2,2} &= f_2(x_{2,j}, t; \boldsymbol{p}) &+K_2^{(m)} &+K_2^{(m+1)} &+\cdots \\ \cdots & & & & \\ \dot{x}_{2,n} &= f_n(x_{2,j}, t; \boldsymbol{p}) &+K_2^{(p)} &+K_2^{(p+1)} &+\cdots \end{cases} \\
\cdots \\
\text{unit N} \begin{cases} \dot{x}_{N,1} &= f_1(x_{N,j}, t; \boldsymbol{p}) &+K_N^{(1)} &+K_N^{(2)} &+\cdots \\ \dot{x}_{N,2} &= f_2(x_{N,j}, t; \boldsymbol{p}) &+K_N^{(m)} &+K_N^{(m+1)} &+\cdots \\ \cdots & & & & \\ \dot{x}_{N,n} &= f_n(x_{N,j}, t; \boldsymbol{p}) &+K_N^{(p)} &+K_N^{(p+1)} &+\cdots \end{cases}
\end{aligned} \tag{1.3}$$

where n is the number of components in a subsystem, and N is the number of units in the system. The vector $\boldsymbol{p}$ denotes the parameters of the system that have many subcategories, see the discussion below. The MPGOS module **Coupled Systems Per-Block solves many instances of the same system presented by Eq.** (1.3) **simultaneously**, but with different parameter sets or initial conditions. The basic idea of the parallelisation strategy is that a single GPU thread treats one unit in the system. The first terms in the right-hand side represented by the functions f_j are the *uncoupled parts* of the system. Thus, they can be computed independently by each thread. Observe how the functions f_j depends only on the state variables $x_{i,j}$ of the same unit, and how the same functions f_j are used for all the units. Here, $i = 1, \ldots, N$ and $j = 1, \ldots, n$ are the indices of the units and the components, respectively. The rest of the terms in the right-hand side are the *coupled parts* and their corresponding coupling values $K_i^{(k)}$. Note that how the coupling values of the same serial number k is added to the same components of each unit. **Keep in mind that the serial number of the couplings k is linear, and the assignment of k to a component is arbitrary. The coupling index k does NOT need to follow any kind of pattern in relation with the component index i; the pattern in Eq.** (1.3) **is artificial and only for demonstration purposes.**

The computation of the values of $K_i^{(k)}$ needs co-operation between the GPU threads. In order to hide this problem from the user, the following strategy is employed. As it is stated before, the uncoupled parts are computed independently, and they are implemented in the right-hand side (ODE) function of the system, for details see Chap. 3. The components of the vector of the coupling terms must also be dependent only on a specific unit:

$$H_i^{(k)} = h^{(k)}(x_{i,j}, t; \boldsymbol{p}). \tag{1.4}$$

Note that for fixed component i, the state variables $x_{i,j}$ belongs only to the i^{th} unit. Therefore, the values of $H_i^{(k)}$ itself can be computed independently by a single thread, and the functions $h^{(k)}$ can be (and have to be) implemented into the same ODE function as the uncoupled part. The number of the functions h that have to be specified is exactly the number of the couplings k. The computed $H_i^{(k)}$ values are stored automatically in the Shared Memory of the GPU. Thus, all the threads in a thread block have access to them, and a block of threads can cooperate to perform the matrix-vector multiplication in Eq. (1.2). It is done automatically after the evaluation of the ODE function and the synchronisation of the threads. **It is important to highlight again that a component $H_i^{(k)}$ must have NO interdependence between different units**. Otherwise, the couplings have to be transformed into the proper form; for an example, see the fist tutorial example of this module in Sec. 6.1.

The computation of a component of the coupling factors also depends only on a single unit by nature:

$$G_i^{(k)} = g^{(k)}(x_{i,j}, t; \boldsymbol{p}). \tag{1.5}$$

It can be implemented into the same ODE function as usual, and the functions $g^{(k)}$ can be evaluated together with the functions f_i and $h^{(k)}$. Thus, this part needs no further discussion. The multiplications with the coupling factor $G_i^{(k)}$ and with the coupling strength σ are also done automatically by the solver. **Note that the coupling between the different units in a system takes place "only" via the coupling matrix $A_{i,j}$ that reflects the topology of the connectivity**.

For additional details about the implementation, the parallelisation strategies and the optimization options of the coupling matrix, the reader is referred to Sec. 3.2, Sec. 4.2.2 and Sec. 4.7, respectively.

The vector of parameters $\boldsymbol{p}$ are divided into five subcategories called *unit parameters* $\boldsymbol{p_{up}}$, *system parameters* $\boldsymbol{p_{sp}}$, *global parameters* $\boldsymbol{p_{gp}}$, user programmable parameters called *unit accessories* $\boldsymbol{p_{uac}}$ and user programmable parameters called *system accessories* $\boldsymbol{p_{sac}}$.

The sets of *unit parameters* are different from instance-to-instance and unit-to-unit of the system. That is, every independent instance and unit (subsystem) have their own set of *unit parameters*. The *system parameters* are different from the instance-to-instance but shared by all the units within a system. The *global parameters* are shared by all the instances and units. That is why they are called global. There are **two types** of global parameters: **integer** (store, e.g., common indices) and **floating point** types (the precision is under the control of the user).

Similarly, as in the case of the Per-Thread module, the last kind of parameters are called *accessories*, which are multi-purpose (user-programmable) parameters. They are not strictly a parameter of Eq. (1.3) 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* is independent of Eq. (1.3), 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 requirement for storing a dense output of every instance. This can be crucial for high performance. Throughout the tutorial examples in Chap. 6, the efficient usage and the capabilities of the *accessories* are emphasized. There are two subcategories of *accessories*: *unit accessories* that are different from the instance-to-instance and unit-to-unit; and *system accessories* that are different from instance-to-instance but shared by all the units within a system. Finally, both subcategories have **two types**: **integer** and **floating point**.

1.5 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. 1.2.

Table 1.2: Summary of the frequently user terms and definitions

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	Parts 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.

1.6 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 prints 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 numbers: 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 Chap. 2. 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
```

1.7 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 instances of a system at different parameter sets need to be solved. In practice, these instances are not solved one at a time on a GPU. Usually, one creates smaller chunks of problems and integrate only a moderate number of instances 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 (for every modules).

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. 1.4. The collection of template parameters are necessary for the memory allocations, for details see Chap. 2. Some tutorials in Chap. 5 provides examples for using more than one SolverObjects, e.g., to overlap CPU-GPU computations.

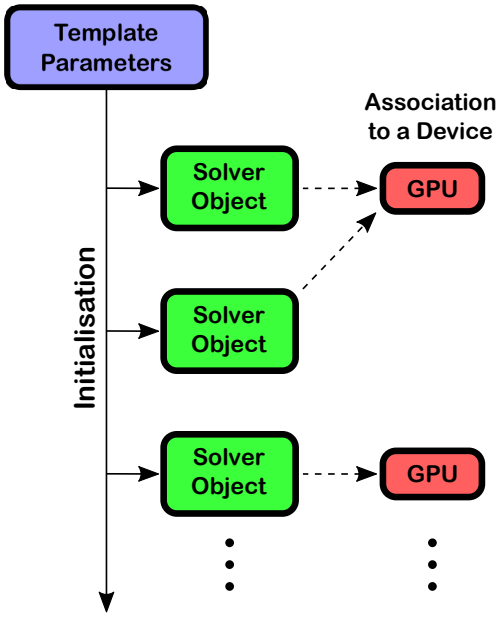

Figure 1.4: Initialisation of SolverObjects on a machine. The collection of template parameters are necessary information for the memory allocations.

The computational workflow can be briefly summarised by the following steps using a single SolverObject and a single GPU (see also Fig. 1.5):

1. Collect all the necessary information for memory allocations via constant template parameters.
2. Create an instance (a SolverObject) of the `ProblemSolver`.
3. Configure the SolverObject with its option possibilities.
4. Fill the data structure of the SolverObject (e.g., time domain, initial conditions and parameters).

5. Synchronise the data from the Host (System Memory) to the Device (Global Memory).
6. Perform integration.
7. Synchronise the data back from the Device (Global Memory) to the Host (System Memory).
8. Record and/or manipulate the retrieved data.
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). That is, points 6, 7 and 8 are usually performed iteratively.
10. Repeat phases between points 4 and 8; for instance, to process other portion of the problems.

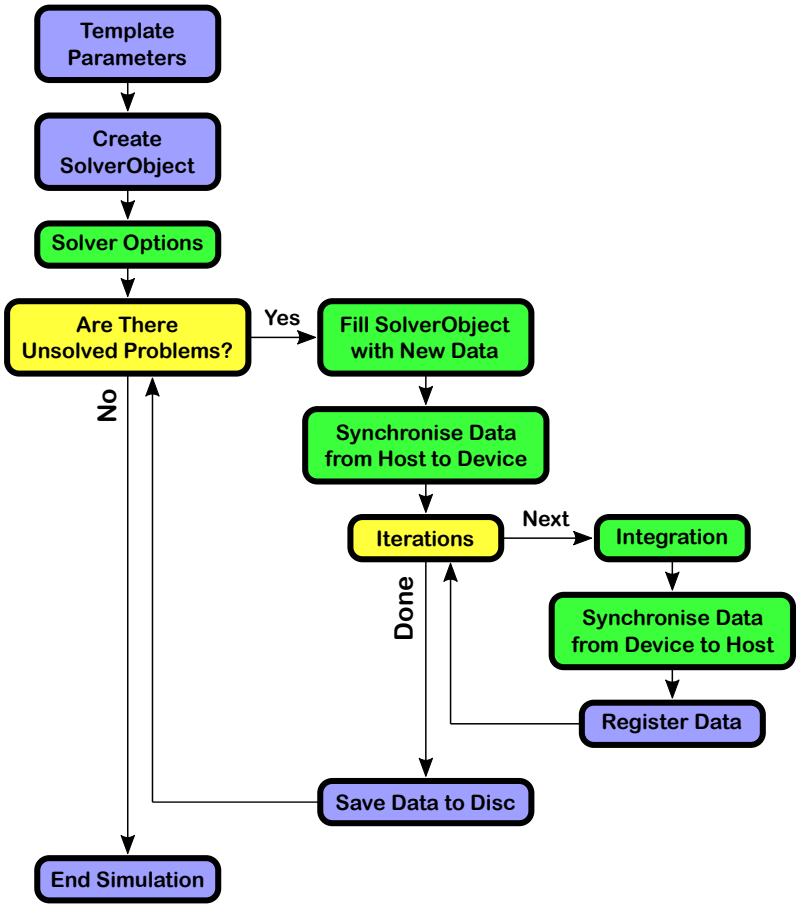

Figure 1.5: Typical workflow using a single SolverObject and a single GPU.

To be able to perform the integration phase, the 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 already discussed at the beginning of Chap. 1. For a thorough discussion about the proper definition of a system, see Chap. 3.

It must be emphasised that the basic workflow introduced here uses a single SolverObject 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 available in a single node/machine. They need more complicated control logic. These features of MPGOS are introduced via tutorial examples, see, e.g., Sec. 5.7 for an example to overlap CPU and GPU computations, or Sec. 5.8 for an example of multi-GPU usage.

Chapter 2

Description of the Interfaces

This chapter describes the SolverObject. Its interface is responsible for a general setup (e.g., specifying the tolerances), for data management between the CPU and the GPU, and the synchronisation of the control flow between the CPU and the GPU.

2.1 Templatisation and creation of a SolverObject

In order to initialise a SolverObject, it needs to know information on how much System Memory and Global Memory has to be allocated. This depends on the various size parameters. Moreover, to be able to produce highly optimised code, these numbers should be passed as template arguments to the SolverObject. In this way, the program package can perform optimisations during compile time. The consequence is that these values must be known at compile time. Therefore, they are allocated as global `const` variables or defined by the `#define` directive in the tutorial examples. The different modules of MPGOS need a different number of template parameters described in the following subsections.

2.1.1 Module: Single System Per-Thread

An example for the definition of the necessary template parameters and the creation of an instance of the `class ProblemSolver` (simply called `Solver` here) is listed below. **The interface of the module have to be included.** It is important to note that a proper device must already be selected via the input argument `DeviceNumber`. The reason is that during the initialisation of the SolverObject, memory allocations are already performed on a corresponding Device (GPU). In case of multiple GPUs and multiple SolverObjects, the serial number of the devices must properly be associated to every SolverObject. The list of the devices can be acquired via the function call `ListCUDADevices()`, see the discussion again in Sec. 1.6. 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 size-related parameters, and their explanation is given in Tab. 2.1. For their more detailed description and usage, the reader is referred to the tutorial examples of this module in Chap. 5. The details of the last two template parameters (the employed numerical algorithm and the data type) are summarized in Tab. 2.2 and Tab. 2.3. Keep in mind that the order of the specified template parameters in the angled brackets <> is fixed. **Important: only the interface of a single module should be included at once due to possible name conflicts. In later versions, this issue will be resolved by putting the different modules into different namespaces.**

```
...
#include "SingleSystem_PerThread_Interface.cuh"
...
#define SOLVER RKCK45     // RK4, RKCK45
#define PRECISION double  // float, double
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()
{
    ...
    ProblemSolver<NT,SD,NCP,NSP,NISP,NE,NA,NIA,NDO,SOLVER,PRECISION> Solver(DeviceNumber);
    ...
}
```

Table 2.1: Description of the size-related template parameters for module **Single System Per-Thread**. The first column is the serial number of the template parameters. The second column shows their notation throughout the manual.

no.	notation	description
1	NT	The number of GPU threads during an integration. This is also the number of instances of the system solved in parallel.
2	SD	System dimension; that is, the number of the *state variables* x_i of the system. The length of vector $\boldsymbol{x}$ in Eq. (1.1).
3	NCP	Number of the *control parameters* (floating point type). Parameters that are different from intance-to-instance.
4	NSP	Number of the *shared parameters* (floating point type). *Shared parameters* are automatically loaded into the fast on-chip Shared Memory of the GPU; thus, it can be accessed by all threads in a thread block very fast with low latency. If a parameter is common to all instances of the system, it is advised to define them as *shared parameters* parameters. For instance, the differentiation matrix is a good candidate if the system is originated from the semi-discretisation of a partial differential equation (PDE).
5	$NISP$	Number of the *shared parameters* of integer type. The same roles apply as for NSP. It is useful when a complicated indexing technique is necessary during the evaluation of the pre-declared user-defined functions.
6	NE	Number of the events/event functions. The solver tries to locate special points of the trajectory with a prescribed tolerance at the zero values of every implicitly defined event function, for details see Sec. 3.1.2 and Sec. 3.1.3.
7	NA	Number of the user-programmable *accessories* (floating point type). It is under the control of the user independently from the actual number of the parameters of a given system. They can be used to store special properties of the trajectories; for instance, a global maximum of a state variable or its value at a detected event. In these cases, the storage of dense output is not necessary, which minimises the slow Global Memory and PCI-E memory transactions.
8	NIA	Number of the user-programmable *accessories* of integer type. The same roles apply as for NA. It is useful, e.g., for storing counters. For instance, how many times an event is detected or the number of the successful time steps, to name a few.
9	NDO	The maximum number of time steps stored for each trajectory into a dense output. Zero value means no dense output storage. If the number of stored points reaches this value, the storing of the trajectory is stopped. Adjustable storage length of the dense output will not be supported in the near future for performance reasons. **The usage of dense output is advisable ONLY for testing purposes.** Reaching the storage limit of the Global Memory, the user might need to reduce the number of the residing threads (NT) and thus the available parallelism. This might result in a significant decrease in performance. To store special properties of the trajectories, use the functionalities of the *accessories*, see the discussion of the pre-declared user-defined device functions in Sec. 3.1 and the tutorials in Chap.5.

Table 2.2: Summary of the available numerical algorithms for module **Single System Per-Thread** (10^{th} template parameter of the SolverObject).

option	description
`RK4`	Classic fourth-order Runge–Kutta solver with fixed time step.
`RKCK45`	Adaptive Runge–Kutta–Cash–Karp method with embedded error estimation of orders 4 and 5.

Table 2.3: Summary of the available data types for module **Single System Per-Thread** (11^{th} template parameter of the SolverObject).

option	description
`float`	Single precision floating point data type.
`double`	Double precision floating point data type.

2.1.2 Module: Coupled Systems Per-Block

An example for the definition of the necessary template parameters and the creation of an instance of the `class ProblemSolver` (simply called `Solver` here) is listed below. **The interface of the module have to be included.** It is important to note that a proper device must already be selected via the input argument `DeviceNumber`. The reason is that during the initialisation of the SolverObject, memory allocations are already performed on a corresponding Device (GPU). In case of multiple GPUs and multiple SolverObjects, the serial number of the devices must properly be associated to every SolverObject. The list of the devices can be acquired via the function call `ListCUDADevices()`, see the discussion again in Sec. 1.6. The parameters between the angled brackets <> are the template parameters, and thus they must be known at compile time. The first 18 template parameters are the size-related parameters, and their explanation is given Tab. 2.4 and Tab. 2.5. **It is highly recommended to read the related parts of Chap. 4 as some of the template parameters can have a significant impact on the performance.** For their more detailed description and usage, the reader is referred to the tutorial examples of this module in Chap. 6. The details of the last two template parameters (the employed numerical algorithm and the data type) are summarized in Tab. 2.6 and Tab. 2.7. Keep in mind that the order of the specified template parameters in the angled brackets <> is fixed. **Important: only the interface of a single module should be included at once due to possible name conflicts. In later versions, this issue will be resolved by putting the different modules into different namespaces.**

```
...
#include "CoupledSystems_PerBlock_Interface.cuh"
...
#define SOLVER RKCK45     // RK4, RKCK45
#define PRECISION double  // float, double
const int NS   = 25;      // Total number of solves systems
const int UPS  = 14;      // Number of units in a system
const int UD   = 2;       // Dimension of a unit (number of components)
const int TPB  = 32;      // Number of threads per block
const int SPB  = 1;       // Number of solved systems per thread block
const int NC   = 2;       // Number of couplings
const int CBW  = 1;       // Coupling bandwidth radius (0: full coupling matrix)
const int CCI  = 1;       // Coupling matrix circularity (0: non-circular, 1: circular)
const int NUP  = 3;       // Number of unit parameters
const int NSP  = 0;       // Number of system parameters
const int NGP  = 1;       // Number of global parameters
const int NiGP = 0;       // Number of integer global parameters
const int NUA  = 3;       // Number of unit accessories
const int NiUA = 1;       // Number of integer unit accessories
```

```
const int NSA   = 2;       // Number of system accessories
const int NiSA  = 1;       // Number of integer system accessories
const int NE    = 1;       // Number of events (per units)
const int NDO   = 50;      // Number of the points of the dense output (per units)
...
int main()
{
    ...
    ProblemSolver<NS,UPS,UD,TPB,SPB,NC,CBW,CCI,NUP,NSP,NGP,NiGP,NUA,NiUA,NSA,NiSA,NE,NDO,
        SOLVER,PRECISION> Solver(DeviceNumber);
    ...
}
```

Table 2.4: Description of the size-related template parameters for module **Coupled Systems Per-Block**. The first column is the serial number of the template parameters. The second column shows their notation throughout the manual.

no.	notation	description
1	NS	The total number of instances of the coupled systems to be solved.
2	UPS	Units per system; that is, the number of the subsystems coupled into a large system.
3	UD	Unit dimension; that is, the number of the *state variables* $x_{i,j}$ (with constant unit index i) of a unit. For an example, see Eq. (1.3).
4	TPB	Number of threads in a thread block. A single unit in an instance of a coupled system is assigned to a single GPU thread. A single GPU thread; however, can deal with multiple units (that might be related to a different instance) depending on the value of UPS and SPB. More precisely, when $TPB < UPS \times SPB$, a single thread have to handle multiple units. The number of units that a single thread handles is called *block launches*. Multiple block launches complicate the control flow of the program; therefore, it should be avoided if possible. This template parameter value should be an integer multiple of the warp size (32 of the present architectures).
5	SPB	Systems per block; that is, the number of solved instances of the coupled system per a block of threads. One instance is assigned to a single block of threads, but according to this template parameter value, a single block of threads can solve multiple instances.
6	NC	Number of couplings according to the linear coupling index k, see Sec. 1.4.2 for more details.
7	CBW	For global coupling between the units, the coupling matrix is full. However, for diffusively coupled units with a coupling radius R, the coupling matrix is a band matrix. With this template parameter, the bandwidth of the coupling matrix can be specified. This results in a more compact storage of the matrix and computations are performed only on the elements within the bandwidth, decreasing the computational costs as well. The compact storage of the matrix allows to store it in the fast on-chip shared memory of the GPU even for a larger number of units per systems. The value of $CWB = 0$ means full matrix. When $CWB > 0$, the matrix is diagonally stored in a circular fashion, see Sec. 4.7.
8	CCI	The compactness of the coupling matrix can be further increased if it is a circular matrix. If $CCI = 1$, only a single value of a diagonal is stored. Keep in mind that the diagonals of a matrix with a certain bandwidth can be "collapsed" into a single value if the matrix is also circular. It is possible that a full matrix is circular; for instance, the differentiation matrix of Fourier basis. In this case, even global coupling can be treated efficiently. For details, see again Sec. 4.7.
9	NUP	Number of the *unit parameters* (floating point type). They are different form instance-to-instance of the system, and different from unit-to-unit within the system.

Table 2.5: Continuation of Tab. 2.4.

no.	notation	description
10	NSP	Number of the *system parameters* (floating point type). They are different form instance-to-instance of the system but shared by all units within the system. They are automatically loaded into the fast on-chip Shared Memory of the GPU.
11	NGP	Number of the *global parameters* (floating point type). They are shared by all the units in all the instances of the system. The user can manage whether it is stored in the Shared Memory or in the Global Memory (e.g., to avoid the reduction of the residing blocks in a streaming multiprocessor) of the GPU.
12	$NiGP$	Number of the *global parameters* of integer type. The same role applies as for NGP. It is useful, e.g., if complicated indexing has to be carried out during the evaluation of the pre-declared user-defined functions.
13	NUA	Number of the user-programmable *unit accessories* (floating point type). They are different form instance-to-instance of the system, and different from unit-to-unit within a system. They can be used to store special properties of the trajectories. For instance, a global maximum of a state variable or its value at a detected event. In these cases, the storage of dense output is not necessary minimising the slow PCI-E memory transactions.
14	$NiUA$	Number of the user-programmable *unit accessories* of integer type. The same roles apply as for NUA. It is useful, e.g., for storing counters. For instance, how many times an event is detected or the number of the successful time steps.
15	NSA	Number of the user-programmable *system accessories* (floating point type). They are different form instance-to-instance of the system but shared by all units within the system. They are automatically loaded into the fast on-chip Shared Memory of the GPU. The same roles apply as for NUA but on a system level instead of on a unit level.
16	$NiSA$	Number of the user-programmable *system accessories* of integer type. The same roles apply as for NSA. It is useful, e.g., for storing counters on a system level instead of on a unit level.
17	NE	Number of the events/event functions on a unit level. The solver tries to locate special points of the trajectory of a unit with a prescribed tolerance at the zero values of every implicitly defined event function, for details see Sec. 3.2.2 and Sec. 3.2.3. It is not possible to detect an event on a system level.
18	NDO	The maximum number of time steps stored for each trajectory. Zero value means no dense output storage. If the number of stored points reaches this value, the storing of the trajectory is stopped. Adjustable storage length of the dense output will not be supported in the near future for performance reasons. **The usage of dense output is advisable ONLY for testing purposes.** Reaching the storage limit of the Global Memory, the user might need to reduce the number of the residing systems (NS) and thus the available parallelism. This might result in a significant decrease in performance. To store special properties of the trajectories, use the functionalities of the *accessories*, see the discussion of the pre-declared user-defined device functions in Sec. 3.2 and the tutorials in Chap.6.

Table 2.6: Summary of the available numerical algorithms for module **Coupled Systems Per-Block** (19^{th} template parameter of the SolverObject).

option	description
`RK4`	Classic fourth-order Runge–Kutta solver with fixed time step.
`RKCK45`	Adaptive Runge–Kutta–Cash–Karp method with embedded error estimation of orders 4 and 5.

Table 2.7: Summary of the available data types for module **Coupled Systems Per-Block** (20^{th} template parameter of the SolverObject).

option	description
`float`	Single precision floating point data type.
`double`	Double precision floating point data type.

2.2 Setup of the SolverObject (options)

Each SolverObject can be customised by its overloaded member function `SolverOption()`

```
Solver.SolverOption( Option, Value);
Solver.SolverOption( Option, SerialNumber, Value);
```

where the proper version is selected according to the argument list. The possible options and their descriptions are summarised in Sec. 2.2.1 and Sec. 2.2.2 for the different modules separately via an **example with their default values**. Although there are many common options for the different modules, we devoted a subsection for each module (and thus repeating many options) so that the user does NOT need to collect information from separate subsections for a specific module. All options have a default value. Therefore, an option needs to be specified only if the default value is not satisfactory. In general, the data type (floating point or integer) of the argument `Value` is option specific. In case of specifying a wrong type, the SolverObject will convert the given value to the proper type automatically. **However, it is advised to pass the argument with the correct data type to avoid unpredictable behaviour.**

2.2.1 Module: Single System Per-Thread

ThreadsPerBlock:

```
Solver.SolverOption(ThreadsPerBlock, WarpSize);
```

The number of threads residing in a block. It can have an impact on the performance, for details see Chap. 4. It is advised to set as an integer multiple of the warp size (32 in all the present CUDA architectures). The upper limit of this value is architecture-specific that can be checked by the MPGOS functions described in Sec. 1.6. The SolverObject queries the warp size of the actual Device and sets the default value accordingly. The data type of `Value` is **integer**.

InitialTimeStep:

```
Solver.SolverOption(InitialTimeStep, 1e-2);
```

The initial time step of the integration phases. This is the employed time step for algorithms with fixed time steps. The data type of `Value` is **floating-point**.

ActiveNumberOfThreads:

```
Solver.SolverOption(ActiveNumberOfThreads, NT);
```

Integration for problem number greater then `ActiveNumberOfThreads` will NOT be performed. It is an important set-up if the total number of problems solved simultaneously on a GPU is smaller than the total number of threads given by the template parameter NT (e.g., during the integration of the last chunk of problems from a larger problem set). In this case, before the last simulation launch, the user can modify this option to specify how many threads should perform the integration that fits into the remaining number of problems. The `Value` must be within $[0, NT]$. The default value is NT. The data type of the `Value` is **integer**.

MaximumTimeStep:

```
Solver.SolverOption(MaximumTimeStep, 1e6);
```

The allowed maximum time step during an integration phase. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

MinimumTimeStep:

```
Solver.SolverOption(MinimumTimeStep, 1e-12);
```

The allowed minimum time step during an integration phase. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

TimeStepGrowLimit:

```
Solver.SolverOption(TimeStepGrowLimit, 5.0);
```

The allowed maximum growth rate of the time step in case of an accepted step. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

TimeStepShrinkLimit:

```
Solver.SolverOption(TimeStepShrinkLimit, 0.1);
```

The allowed minimum shrink rate of the time step in case of a rejected step. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

RelativeTolerance:

```
Solver.SolverOption(RelativeTolerance, SerialNumber, 1e-8);
```

Relative tolerance for the components of the system. The component is specified via the `SerialNumber` argument and must be within $[0, SD-1]$, where SD is the system dimension template parameter. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

AbsoluteTolerance:

```
Solver.SolverOption(AbsoluteTolerance, SerialNumber, 1e-8);
```

Absolute tolerance for the components of the system. The component is specified via the `SerialNumber` argument and must be within $[0, SD-1]$, where SD is the system dimension template parameter. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

EventTolerance:

```
Solver.SolverOption(EventTolerance, SerialNumber, 1e-6);
```

Absolute tolerance for detecting special points by events. The serial number of the event is specified via the `SerialNumber` argument and must be within $[0, NE-1]$, where NE is the template parameter to give the number of events. The data type of `Value` is **floating-point**.

EventDirection:

```
Solver.SolverOption(EventDirection, SerialNumber, 0);
```

Detection of events with positive (1) or negative (-1) tangent of the event function. The value 0 means detection in both directions. The serial number of the event is specified via the `SerialNumber` argument and must be within $[0, NE-1]$, where NE is the template parameter to give the number of events. The data type of `Value` is **integer**.

DenseOutputMinimumTimeStep:

```
Solver.SolverOption(DenseOutputMinimumTimeStep, 0.0);
```

By default, the dense output is stored after every successful time step until the stored number of time steps reached the value of the template parameter NDO. In the present version, there is NO option for storing the steps on an equidistant spacing in time for performance reasons. However, it is possible that in some parts of the integration, the size of the time steps must be decreased significantly to keep the prescribed tolerances. With this option, the minimum elapsed time between the stored steps can be prescribed to prevent the fill-up of the storage capacities. Obviously, the 0.0 default value means that every successful time step is stored. The data type of `Value` is **floating-point**.

DenseOutputSaveFrequency:

```
Solver.SolverOption(DenseOutputSaveFrequency, 1);
```

This is another option to prevent the fill-up of the storage capacities. Time steps are stored only by the prescribed frequency. For instance, if the value is 2 or 3, only every second or third successful time steps are stored, respectively. The data type of `Value` is **integer**.

PreferSharedMemory:

```
Solver.SolverOption(PreferSharedMemory, 1);
```

Manages the storage of *shared parameters* (both floating-point and integer types) in Shared Memory (0: stored in Global Memory, 1: stored in Shared Memory). Keep in mind that storing a variable in Shared Memory can significantly decrease the access time of that variable. However, Shared Memory is a scarce resource; overusing it can significantly reduce the residing number of blocks on a streaming multiprocessor. This can reduce the available "parallelism", which might decrease the performance. Thus, using Shared Memory is not trivial and problem-dependent. Make your own experience. The data type of `Value` is **integer**.

2.2.2 Module: Coupled Systems Per-Block

InitialTimeStep:

```
Solver.SolverOption(InitialTimeStep, 1e-2);
```

The initial time step of the integration phases. This is the employed time step for algorithms with fixed time steps. The data type of `Value` is **floating-point**.

ActiveSystems:

```
Solver.SolverOption(ActiveSystems, NS);
```

Integration for problem number (number of instances of the system) greater then `ActiveSystems` will NOT be performed. It is an important set-up if the total number of problems solved simultaneously on a GPU is smaller than the total number of instances specified by the template parameter NS. This option is useful, e.g., during the integration of the last chunk of problems from a larger problem set. In this case, before the last simulation launch, the user can modify this option to specify how many instances should participate in the integration process. The `Value` must be within $[0, NS]$. The default value is NS. The data type of `Value` is **integer**.

MaximumTimeStep:

```
Solver.SolverOption(MaximumTimeStep, 1e6);
```

The allowed maximum time step during an integration phase. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

MinimumTimeStep:

```
Solver.SolverOption(MinimumTimeStep, 1e-12);
```

The allowed minimum time step during an integration phase. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

TimeStepGrowLimit:

```
Solver.SolverOption(TimeStepGrowLimit, 5.0);
```

The allowed maximum growth rate of the time step in case of an accepted step. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

TimeStepShrinkLimit:

```
Solver.SolverOption(TimeStepShrinkLimit, 0.1);
```

The allowed minimum shrink rate of the time step in case of a rejected step. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

RelativeTolerance:

```
Solver.SolverOption(RelativeTolerance, SerialNumber, 1e-8);
```

Relative tolerance for the components of the units in the system. The component is specified via the `SerialNumber` argument and must be within $[0, UD - 1]$, where UD is the unit dimension template parameter. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

AbsoluteTolerance:

```
Solver.SolverOption(AbsoluteTolerance, SerialNumber, 1e-8);
```

Absolute tolerance for the components of the units of the system. The component is specified via the `SerialNumber` argument and must be within $[0, UD - 1]$, where UD is the unit dimension template parameter. It is used only by adaptive solvers. The data type of `Value` is **floating-point**.

EventTolerance:

```
Solver.SolverOption(EventTolerance, SerialNumber, 1e-6);
```

Absolute tolerance for detecting special points by events. The serial number of the event is specified via the `SerialNumber` argument and must be within $[0, NE-1]$, where NE is the template parameter to give the number of events. Keep in mind that events can be specified only on a unit level. There is NO possibility to detect events of interacting units. The data type of `Value` is **floating-point**.

EventDirection:

```
Solver.SolverOption(EventDirection, SerialNumber, 0);
```

Detection of events with positive (1) or negative (-1) tangent of the event function. The value 0 means detection in both directions. The serial number of the event is specified via the `SerialNumber` argument and must be within $[0, NE - 1]$, where NE is the template parameter to give the number of events. The data type of `Value` is **integer**.

DenseOutputMinimumTimeStep:

```
Solver.SolverOption(DenseOutputMinimumTimeStep, 0.0);
```

By default, the dense output is stored after every successful time step until the stored number of time steps reached the value of the template parameter NDO. In the present version, there is NO option for storing the steps on an equidistant spacing in time for performance reasons. However, it is possible that in some parts of the integration, the size of the time steps must be decreased significantly to keep the prescribed tolerances. With this option, the minimum elapsed time between the stored steps can be prescribed to prevent the fill-up of the storage capacities. Obviously, the 0.0 default value means that every successful time step is stored. The data type of `Value` is **floating-point**.

DenseOutputSaveFrequency:

```
Solver.SolverOption(DenseOutputSaveFrequency, 1);
```

This is another option to prevent the fill-up of the storage capacities. Time steps are stored only by the prescribed frequency. For instance, if the value is 2 or 3, only every second or third successful time steps are stored, respectively. The data type of `Value` is **integer**.

SharedGlobalVariables:

```
Solver.SolverOption(SharedGlobalVariables, 1);
```

Manages the storage of textitglobal parameters (both floating-point and integer types) in Shared Memory (0: stored in Global Memory, 1: stored in Shared Memory). Keep in mind that storing a variable in Shared Memory can significantly decrease the access time of that variable. However, Shared Memory is a scarce resource; overusing it can significantly reduce the residing number of blocks on a streaming multiprocessor. This can reduce the available "parallelism", which might decrease the performance. Thus, using Shared Memory is not trivial and problem-dependent. Make your own experience. The data type of `Value` is **integer**.

SharedCouplingMatrices:

```
Solver.SolverOption(SharedCouplingMatrices, 0);
```

Manages the storage of coupling matrices in Shared Memory (0: stored in Global Memory, 1: stored in Shared Memory). **If the matrices are circular ($CCI = 1$), they are always stored in the Shared Memory**. Keep in mind that storing a variable in Shared Memory can significantly decrease the access time of that variable. However, Shared Memory is a scarce resource; overusing it can significantly reduce the residing number of blocks on a streaming multiprocessor. This can reduce the available "parallelism", which might decrease the performance. Thus, using Shared Memory is not trivial and problem-dependent. Make your own experience. The data type of `Value` is **integer**.

2.3 Data management

Data management is an important part of the interface of the program package MPGOS. It has three major components. The first one is the interface to set up the data structure in the SolverObject. This can be done by the overloaded member function

```
Solver.SetHost( ...args... );
```

where the proper version of the member function is selected according to the argument list. The argument list required depends on the variable about to set. As the name of the member function implies, it sets the value of a variable only in the Host (CPU) side. The second component is the synchronisation of the data between the Host (CPU) and the Device (GPU):

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

where the sole argument `Variable` selects the variable about to synchronise. Before performing any integration, the synchronisation of the necessary data between the Host and the Device is necessary. At some point, after performing one or more integration procedures, synchronisation of the newly calculated data from the Device to the Host is usually required. To retrieve data from the internal storages of the SolverObject, the third component of the interface can be used. This is again an overloaded member function, where the argument list depends on the actual variable:

```
Solver.GetHost<RETURNTYPE>( ...args... );
```

The template argument `RETURNTYPE` is necessary, and it specifies the requested return type. Keep in mind that any kind of variable (e.g., a floating-point number of the state variable) can be retrieved as any tpye of data (`int`, `float` or `double`). The `GetHost()` member function automatically performs the type conversion. **Thus, the return tpye has to be carfully prescribed.**

Since there are many versions of the overloaded member functions `SetHost()` and `GetHost()`, an example is presented for all the possible variables is Sec. 2.3.1 and Sec. 2.3.2 for the different modules separately (to keep the discussion clear).

2.3.1 SetHost and GetHost member functions for the module: Single System Per-Thread

TimeDomain:

```
Solver.SetHost(Instance, TimeDomain, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, TimeDomain, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The `Component` has two possible options (integer): 0 is the lower bound and 1 is the uper bound. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). The upper bound should be larger than the lower bound (integration backward in time is not tested in MPGOS). The integration is automatically stopped reaching the upper bound of the time domain. **Specifying this variable is mandatory.**

ActualState:

```
Solver.SetHost(Instance, ActualState, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, ActualState, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The `Component` is the serial number of the component of the state space; its data type is **integer** and its range is $[0, SD-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). **The set-up of the `ActualState` is the initial condition of the instances, and specifying it is mandatory.**

ControlParameters:

```
Solver.SetHost(Instance, ControlParameters, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, ControlParameters, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The `Component` is the serial number of the *control parameter*; its data type is **integer** and its range is $[0, NCP-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`).

Accessories:

```
Solver.SetHost(Instance, Accessories, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, Accessories, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The `Component` is the serial number of the *accessories*; its data type is **integer** and its range is $[0, NA-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`).

IntegerAccessories:

```
Solver.SetHost(Instance, IntegerAccessories, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, IntegerAccessories, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The `Component` is the serial number of the *accessories*; its data type is **integer** and its range is $[0, NIA-1]$. The data type of the `Value` is **integer** (it can be changed by the `RETURNTYPE`).

DenseTime:

```
Solver.SetHost(Instance, DenseTime, TimeStep, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, DenseTime, TimeStep);
```

It is possible to pre-fill some portion or all of the states in the dense output. It is a series of points (t^k, x_i^k), where t^k is the time instance at time step number k and x_i^k is the state of an instance at time step k. The variable `DenseTime` sets the value of the time instances t^k. The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The `TimeStep` is the serial number k of the time step in the dense output; its data type is **integer** and its range is $[0, NDO-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). This feature will be important in a later version of MPGOS, when the solution of delay differential equations (DDEs) will be supported. In this case, an initial function has to be given, representing the history of the trajectory in the past.

ActualTime:

```
Solver.SetHost(Instance, ActualTime, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, ActualTime);
```

For flexibility reasons, an integration phase does NOT start form the lower bound of the `TimeDomain`, but from a user-specified time instance called `ActualTime`. Thus, `ActualTime` and `TimeDomain` is completely independent from each other. The variable `ActualTime` always stores the actual progress of an integration in time. Thus, it makes possible to retrieve the value of the time instance at a termination that is different from the upper bound of the `TimeDomain` (e.g., due to stop by event). As its value does NOT change in the Global Memory of the GPU after termination, a subsequent integration phase will continue the integration automatically from the actual value of the `ActualTime`. This would NOT be possible when the solver automatically resets the initial time of the integration to the lower bound of the `TimeDomain`. The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). **Specifying this variable is mandatory.**

DenseIndex:

```
Solver.SetHost(Instance, DenseIndex, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, DenseIndex);
```

It is possible to pre-fill some portion or all of the states in the dense output. It is a series of points (t^k, x_i^k), where t^k is the time instance at time step number k and x_i^k is the state of an instance at time step k. The variable `DenseIndex` specifies the index in the dense output data structure from which the solver will start to store the computed states. Initially, it is presumably equal to the number of the pre-stored points in the dense output (as the indexing starts from zero). Specifying a lower value, the solver will overwrite some portion of the user-defined values. Specifying a larger value will leave a gap in the dense output data structure. If no pre-filling of the dense output is necessary, its value should be 0. The variable `DenseIndex` is continuously updated by the solver, and at the end of an integration phase, it points to the last valid element in the dense output. The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The data type of the `Value` is **integer** (it can be changed by the `RETURNTYPE`); and its range is $[0, NDO-1]$. This feature will be important in a later version of MPGOS, when the solution of delay differential equations (DDEs) will be supported. In this case, an initial function has to be given, representing the history of the trajectory in the past. **Specifying this variable is mandatory if** $NDO > 0$**.**

SharedParameters:

```
Solver.SetHost(SharedParameters, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(SharedParameters, Component);
```

The `Component` is the serial number of the *shared parameter*; its data type is **integer** and its range is $[0, NSP-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). Observe that NO serial number for the instance is necessary as this parameter is shared among all the instances.

IntegerSharedParameters:

```
Solver.SetHost(IntegerSharedParameters, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(IntegerSharedParameters, Component);
```

The `Component` is the serial number of the *shared parameter* (of type integer); its data type is **integer** and its range is $[0, NISP-1]$. The data type of the `Value` is **integer** (it can be changed by the `RETURNTYPE`). Observe that NO serial number for the instance is necessary as this parameter is shared among all the instances.

DenseState:

```
Solver.SetHost(Instance, DenseState, Component, TimeStep, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, DenseState, Component, TimeStep);
```

It is possible to pre-fill some portion or all of the states in the dense output. It is a series of points (t^k, x_i^k), where t^k is the time instance at time step number k and x_i^k is the state of an instance at time step k. The variable `DenseState` sets the value of the states x_i^k. The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NT-1]$. The `Component` is the serial number i of the component of the state space; its data type is **integer** and its range is $[0, SD-1]$. The `TimeStep` is the serial number k of the time step in the dense output; its data type is **integer** and its range is $[0, NDO-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). This feature will be important in a later version of MPGOS, when the solution of delay differential equations (DDEs) will be supported. In this case, an initial function has to be given, representing the history of the trajectory in the past.

2.3.2 SetHost and GetHost member functions for the module: Coupled Systems Per-Block

ActualState:

```
Solver.SetHost(Instance, Unit, ActualState, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, Unit, ActualState, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Unit` is the serial number of the unit in the system; its data type is **integer** and its range is $[0, UPS-1]$. The `Component` is the serial number of the component of the state space of a unit; its data type is **integer** and its range is $[0, UD-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). **The set-up of the `ActualState` is the initial condition of the instances, and specifying it is mandatory.**

UnitParameters:

```
Solver.SetHost(Instance, Unit, UnitParameters, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, Unit, UnitParameters, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Unit` is the serial number of the unit in the system; its data type is **integer** and its range is $[0, UPS-1]$. The `Component` is the serial number of the component of the *unit parameters* of a unit; its data type is **integer** and its range is $[0, NUP-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`).

UnitAccessories:

```
Solver.SetHost(Instance, Unit, UnitAccessories, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, Unit, UnitAccessories, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Unit` is the serial number of the unit in the system; its data type is **integer** and its range is $[0, UPS-1]$. The `Component` is the serial number of the component of the *unit accessories* of a unit; its data type is **integer** and its range is $[0, NUA-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`).

IntegerUnitAccessories:

```
Solver.SetHost(Instance, Unit, IntegerUnitAccessories, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, Unit, IntegerUnitAccessories, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Unit` is the serial number of the unit in the system; its data type is **integer** and its range is $[0, UPS-1]$. The `Component` is the serial number of the component of the *unit accessories* (of integer type) of a unit; its data type is **integer** and its range is $[0, NiUA-1]$. The data type of the `Value` is **integer** (it can be changed by the `RETURNTYPE`).

TimeDomain:

```
Solver.SetHost(Instance, TimeDomain, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, TimeDomain, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Component` has two possible options (integer): 0 is the lower bound and 1 is the uper bound. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). The upper bound should be larger than the lower bound (integration backward in time is not tested in MPGOS). The integration is automatically stopped reaching the upper bound of the time domain. **Specifying this variable is mandatory.** Observe that NO unit number is necessary as the time domian is shared among all the units.

SystemParameters:

```
Solver.SetHost(Instance, SystemParameters, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, SystemParameters, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Component` is the serial number of the component of the *system parameters* of an instance; its data type is **integer** and its range is $[0, NSP-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). Observe that NO unit number is necessary as this parameter is shared among all the units.

SystemAccessories:

```
Solver.SetHost(Instance, SystemAccessories, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, SystemAccessories, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Component` is the serial number of the component of the *system accessories* of an instance; its data type is **integer** and its range is $[0, NSA-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). Observe that NO unit number is necessary as this accessory is shared among all the units.

IntegerSystemAccessories:

```
Solver.SetHost(Instance, IntegerSystemAccessories, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, IntegerSystemAccessories, Component);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Component` is the serial number of the component of the *system accessories* (of integer type) of an instance; its data type is **integer** and its range is $[0, NiSA-1]$. The data type of the `Value` is **integer** (it can be changed by the `RETURNTYPE`). Observe that NO unit number is necessary as this accessory is shared among all the units.

DenseTime:

```
Solver.SetHost(Instance, DenseTime, TimeStep, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, DenseTime, TimeStep);
```

It is possible to pre-fill some portion or all of the states in the dense output. It is a series of points $(t^k, x^k_{i,j})$, where t^k is the time instance at time step number k and $x^k_{i,j}$ is the state of an instance at time step k of unit i and component j. The variable `DenseTime` sets the value of the time instances t^k. The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `TimeStep` is the serial number k of the time step in the dense output; its data type is **integer** and its range is $[0, NDO-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). This feature will be important in a later version of MPGOS, when the solution of delay differential equations (DDEs) will be supported. In this case, an initial function has to be given, representing the history of the trajectory in the past.

CouplingStrength:

```
Solver.SetHost(Instance, CouplingStrength, SerialNumber, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, CouplingStrength, SerialNumber);
```

The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `SerialNumber` is the serial number of the coupling; its data type is **integer** and its range is $[0, NC-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). Observe that NO unit number is necessary as the coupling strength is shared among all the units.

GlobalParameters:

```
Solver.SetHost(GlobalParameters, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(GlobalParameters, Component);
```

The `Component` is the serial number of the component of the *global parameters* of the system; its data type is **integer** and its range is $[0, NGP-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). Observe that NO instance and unit number is necessary as the global parameter is shared among all the instances.

IntegerGlobalParameters:

```
Solver.SetHost(IntegerGlobalParameters, Component, Value);
Value = Solver.GetHost<RETURNTYPE>(IntegerGlobalParameters, Component);
```

The `Component` is the serial number of the component of the *global parameters* (of integer type) of the system; its data type is **integer** and its range is $[0, NiGP-1]$. The data type of the `Value` is **integer** (it can be changed by the `RETURNTYPE`). Observe that NO instance and unit number is necessary as the global parameter is shared among all the instances.

CouplingIndex:

```
Solver.SetHost(CouplingIndex, SerialNumber, Value);
Value = Solver.GetHost<RETURNTYPE>(CouplingIndex, SerialNumber);
```

In Sec. 1.4.2, the general form of a coupling is discussed, see also Eq. (1.2). The computed coupling values of a coupling with serial number k are added to the right-hand side of the units to a specific unit component i. The variable `CouplingIndex` serves to associate a certain coupling serial number to a specific unit component i. The `SerialNumber` is the serial number of the coupling (k); its data type is **integer** and its range is $[0, NC-1]$. The data type of the `Value` is **integer** and it is the component i of the units to which the couling values of coupling k will be added (the type of `Value` can be changed by the `RETURNTYPE`); its range is $[0, UD-1]$.

CouplingMatrix:

```
Solver.SetHost(SerialNumber, CouplingMatrix, Row, Col, Value);
Value = Solver.GetHost<RETURNTYPE>(SerialNumber, CouplingMatrix, Row, Col);
```

It must be emphasised that the coupling matrices must be filled as if they were "ordinary" full matrices in terms of the specified row and column indexing. When the user specifies a certain bandwidth (CBW) or circularity (CCI), the unnecessary elements will not be stored; moreover, the `SetHost()` and `GetHost()` member functions will throw an error if the user, e.g., uses invalid `Row` and `Col` indices that are outside of the bandwidth. Keep in mind again that the diagonals with finite bandwidth are stored with their circular extension; therefore, proper values for these circular extensions must also be provided even if they are zero (for safety reason), for details see Sec. 4.7. The `SerialNumber` is the serial number of the coupling (k); its data type is **integer** and its range is $[0, NC-1]$. **Keep in mind also that a coupling matrix must be specified for each coupling even if the matrices are the same.** The input arguments `Row` and `Col` are the row and column indices, respectively. Their range is $[0, UPS-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`).

ActualTime:

```
Solver.SetHost(Instance, ActualTime, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, ActualTime);
```

For flexibility reasons, an integration phase does NOT start form the lower bound of the `TimeDomain`, but from a user-specified time instance called `ActualTime`. Thus, `ActualTime` and `TimeDomain` is completely independent from each other. The variable `ActualTime` always stores the actual progress of an integration in time. Thus, it makes possible to retrieve the value of the time instance at a termination that is different from the upper bound of the `TimeDomain` (e.g., due to stop by event). As its value does NOT change in the Global Memory of the GPU after termination, a subsequent integration phase will continue

the integration automatically from the actual value of the `ActualTime`. This would NOT be possible when the solver automatically resets the initial time of the integration to the lower bound of the `TimeDomain`. The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). **Specifying this variable is mandatory.** Observe that NO unit number is necessary as the actual time is shared among all the units.

DenseIndex:

```
Solver.SetHost(Instance, DenseIndex, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, DenseIndex);
```

It is possible to pre-fill some portion or all of the states in the dense output. It is a series of points $(t^k, x^k_{i,j})$, where t^k is the time instance at time step number k and $x^k_{i,j}$ is the state of an instance at time step k of unit i and component j. The variable `DenseIndex` specifies the index in the dense output data structure from which the solver will start to store the computed states. Initially, it should be equal to the number of the pre-stored points in the dense output (as the indexing starts from zero). Specifying a lower value, the solver will overwrite some portion of the user-defined values. Specifying a larger value will leave a gap in the dense output data structure. If no pre-filling of the dense output is necessary, its value should be 0. The variable `DenseIndex` is continuously updated by the solver, and at the end of an integration phase, it points to the last valid element in the dense output. The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The data type of the `Value` is **integer** (it can be changed by the `RETURNTYPE`); and its range is $[0, NDO-1]$. This feature will be important in a later version of MPGOS, when the solution of delay differential equations (DDEs) will be supported. In this case, an initial function has to be given, representing the history of the trajectory in the past. **Specifying this variable is mandatory if** $NDO > 0$**.**

DenseState:

```
Solver.SetHost(Instance, Unit, DenseState, Component, TimeStep, Value);
Value = Solver.GetHost<RETURNTYPE>(Instance, Unit, DenseState, Component, TimeStep);
```

It is possible to pre-fill some portion or all of the states in the dense output. It is a series of points $(t^k, x^k_{i,j})$, where t^k is the time instance at time step number k and $x^k_{i,j}$ is the state of an instance at time step k of unit i and component j. The variable `DenseState` sets the values of $x^k_{i,j}$). The argument `Instance` is the serial number of the instance of the ODE system; its data type is **integer** and its range is $[0, NS-1]$. The `Unit` is the serial number i of the unit in the system; its data type is **integer** and its range is $[0, UPS-1]$. The `Component` is the serial number of the component j of the state space of a unit; its data type is **integer** and its range is $[0, UD-1]$. The `TimeStep` is the serial number k of the time step in the dense output; its data type is **integer** and its range is $[0, NDO-1]$. The data type of the `Value` is **floating-point** (it can be changed by the `RETURNTYPE`). This feature will be important in a later version of MPGOS, when the solution of delay differential equations (DDEs) will be supported. In this case, an initial function has to be given, representing the history of the trajectory in the past.

2.3.3 Synchronise data between the Host and the Device

As it is already given at the beginning of Sec. 2.3, the synchronisation of the data between the Host (CPU) and Device (GPU) side can be performed via the member functions

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

for all modules. The only difference between the modules is the available options for the argument `Variable`. The list of options for the different modules are summarised in Tab. 2.8 and Tab. 2.9. The `DenseOutput` option synchronises every dense output related data (`DenseIndex`, `DenseTime` and `DenseState`). The option `All` synchronises everything.

Table 2.8: Summary of the possible options for `Variable` of the synchronisation process. Module: **Single System Per-Thread**.

argument values
`TimeDomain`
`ActualState`
`ActualTime`
`ControlParameters`
`SharedParameters`
`IntegerSharedParameters`
`Accessories`
`IntegerAccessories`
`DenseOutput`
`All`

Table 2.9: Summary of the possible options for `Variable` of the synchronisation process. Module: **Coupled Systems Per-Block**.

argument values
`TimeDomain`
`ActualState`
`ActualTime`
`UnitParameters`
`SystemParameters`
`GlobalParameters`
`IntegerGlobalParameters`
`UnitAccessories`
`IntegerUnitAccessories`
`SystemAccessories`
`IntegerSystemAccessories`
`CouplingMatrix`
`CouplingStrength`
`CouplingIndex`
`DenseOutput`
`All`

2.4 Integration and control flow management of CPU and GPU operations

After the proper data management operations, the integration can be performed simply by calling the member function (for all modules)

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

which integrates every instances of the ODE system given by Eq. (1.1) or Eq. (1.3) simultaneously from the specified `ActualTime` to the upper limit of the `TimeDomain` (unless the simulation is finished earlier due to user specified termination). It is important to emphasize 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 SolverObjects. 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 SolverObject. 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 SolverObject reaches the inserted synchronisation point. That is, the CPU thread is synchronised with respect to the SolverObject. Such a very simple combination of function calls works perfectly using a single GPU and a single SolverObject, for more complicated synchronisation patterns the reader is referred to the tutorial examples in Sec. 5.7 and 5.8.

The last option

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

synchronises the CPU with respect to a Device the calling SolverObject is assigned to. That is, even if there are multiple SolverObjects 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 SolverObject was the caller).

2.5 Quick print the content of SolverObject to file

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

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

where the second one is used exclusively for printing out the dense output. The name of the generated text file is `<Variable>.txt`, except for the dense output: `<Variable>_<SystemNumber>.txt`. The list of the options for the `Variable` of the different modules are summarised in Tab. 2.10 and Tab. 2.11. Keep in mind the for the option `DenseOutput`, the serial number of the instance of the system must be specified as well.

Table 2.10: Summary of the possible options for `Variable` of the printing process. Module: **Single System Per-Thread**.

argument values
TimeDomain
ActualState
ActualTime
ControlParameters
SharedParameters
IntegerSharedParameters
Accessories
IntegerAccessories
DenseOutput

Table 2.11: Summary of the possible options for `Variable` of the printing process. Module: **Coupled Systems Per-Block**.

argument values
TimeDomain
ActualState
ActualTime
UnitParameters
SystemParameters
GlobalParameters
IntegerGlobalParameters
UnitAccessories
IntegerUnitAccessories
SystemAccessories
IntegerSystemAccessories
CouplingMatrix
CouplingStrength
CouplingIndex
DenseOutput

Chapter 3

Details of the pre-declared user-defined device functions

The last step to set up a problem is to define the pre-declared user-defined device functions (PDUDDF); that is, to define the system itself. An example of a pre-declared user-defined device function is the right-hand side of the system. However, we shall see that there are many other similar functions that affect the behaviour of the integrator. This section is dedicated to describing them. It is important to emphasize that these functions can be implemented very similarly, as in the case of MATLAB. To fully grasp the ideas behind these functions, the internal structure of the `Solver()` member function has to be discussed in more details. Figure 3.1 shows its workflow for adaptive solvers with event handling (for all modules). This is the most complex flow structure. Using solvers with fixed time steps or without event handling, the structure can be simplified by eliminating the corresponding elements in the flow structure. The dashed lines indicate function calls to a pre-declared user-defined device function. The description of the workflow of the `Solver()` member function is as follows

- Due to the template parameters, all the required data structures for the solver for a GPU thread can be allocated in the Registers and/or in the Shared Memory. This has the advantages that the indexing of the proper data becomes much more simple, saving a tremendous amount of integers instructions. Most of the data structures stored in a linearly allocated memeory segment for all the threads. Thus, the complex computation (linear combination) of the memory addresses have to be calculated only once when they are loaded into the Register variables or into the Shared Memory.Naturally, register spilling can happen for complex systems when the variables do not fit into the available registers per threads; however, these variables have to be loaded regularly from Global Memory anyway.
- Next, during an initialisation procedure, the `Initialisation()` pre-declared user-defined 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 procedure. 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 the requirements.
- The integration can be stopped in two ways: reaching the upper limit of the prescribed `TimeDomain`, or by user-defined termination (see the discussion below).
- If the integration is not finished, then a time step is performed. The first task is to calculate the new state of a time step and estimate the corresponding error (in case of adaptive solver). If the step is rejected, a new corrected time step is estimated, and the stepping is repeated with the new, smaller time step. If the step is accepted, a new time step is estimated for the next step, and a series of instructions are performed discussed below in details. Using solvers with fixed time steps, the step is

always accepted and the time step estimation is completely ignored. This phase successively calls the right-hand side of the ODE system called `OdeFunction()` that is also a pre-declared user-defined device function.

- The next step is the evaluation of the event function via another pre-declared user-defined device function `EventFunction()`. If an event is overstepped, the actual step is rejected, and a new time step is estimated to detect the event withing the prescribed absolute tolerance.
- 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 no overstepping of an event occurred, the actual state and actual time are updated with the newly calculated step. Keep in mind that the step is accepted. After that, the pre-declared user-defined device function `ActionAfterSuccessfulTimeStep()` is called, in which the user has the possibility to manipulate the trajectories or store/accumulate any kind of properties of the solution into the *accessories*.
- When the correction of the time step (in the previous step) was successful to detect an event, the pre-declared user-defined device function `ActionAfterEventDetection()` is called. Similarly, as in the case of the previous point, the user has the possibility to manipulate the trajectories or store/accumulate any kind of properties of the solution into the *accessories*. The trajectory manipulation can be important, e.g., in systems that can exhibit impact dynamics. The impact law can be applied here immediately without the requirement of any special treatment. Due to the possibility of trajectory manipulation by the user, the event functions corresponding to the new step is re-evaluated. In addition, inside the functions `ActionAfterSuccessfulTimeStep()` and `ActionAfterEventDetection()` the user has the possibility to terminate the simulation. In this way, the simulation can be stopped before reaching the upper limit of the time domain.
- As the last process inside a time step, the control flow checks for user terminations, and the actual time for reaching the upper limit of the time domain. For stopping the simulation precisely at the end of the time domain, a final correction in the time step might be necessary.
- The last type of a pre-declared user-defined device function `Finalisation()` is called before finishing the integration. Again, it is a user-manageable function that can manipulate the trajectories or store special properties at the end of an integration phase.

In Fig. 3.1, 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, the required quantity can be computed

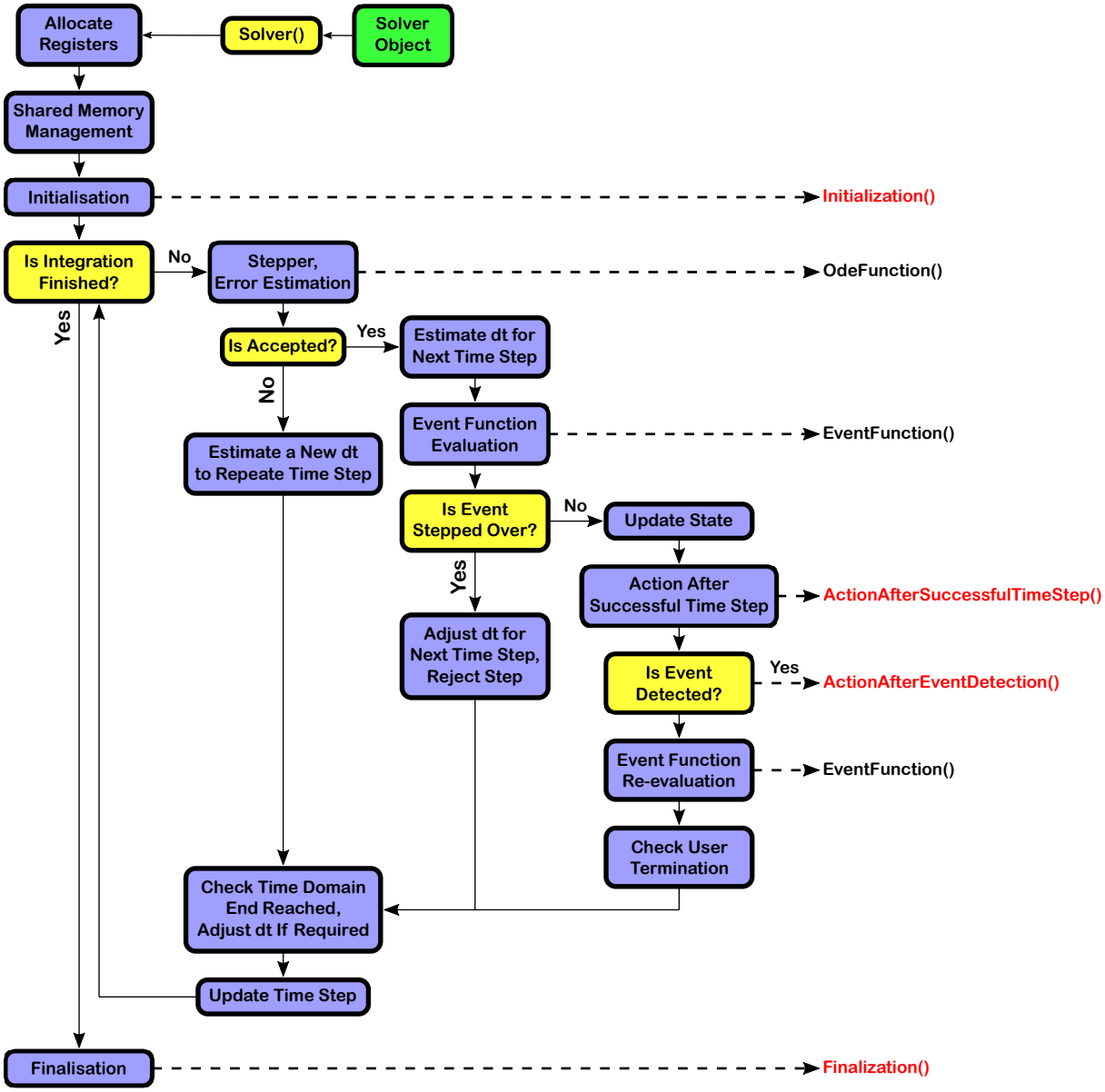

Figure 3.1: The general, internal structure of the `Solver()` member function (for all modules).

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 `ActionAfterSuccessful TimeStep()`. 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 seeking algorithm to every trajectories.

Thus, again a tremendous amount of overhead is eliminated, since everything is computed immediately on 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 4) 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, see the beginning of Chap. 1. 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 header file in the same directory and include this new system definition file to the main program code. Observe also that the system definition 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.

3.1 Module: Single System Per-Thread

Before proceeding further into the details of the pre-declared user-defined function, the abbreviated names of the used arguments and their descriptions are summarised in Tab. 3.1. **The details of the functions are introduced via the first tutorial example, see Sec. 5.1.**

Table 3.1: Summary of the input arguments of the pre-declared user-defined device functions for module **Single System Per-Thread**

Argument	Description
tid	The serial number of the threads indexed from 0 to $NT-1$. Since one thread solves one system in the SolverObject, this is also the serial number of the systems.
NT	The number of the threads NT. It is also the number of the systems solved simultaneously.
F	The array to where the right-hand side of the system is evaluated. Its length is the system dimension SD.
X	The actual state of the systems. Its length is the system dimension SD.
T	The actual time of the systems. **In case of the successive call of the `Solve()` member function, the actual time is NOT automatically reseted to the lower bound of the `TimeDomain` (TD).** If necessary, it can be done, e.g., in the `Initialisation()` or `Finalisation()` functions.
dT	The actual time step.
TD	The time domain of the integration. There are two values for each system; $TD[0]$ (lower bound) and $TD[1]$ (upper bound). Therefore, its total length is 2.
cPAR	The *control parameters* of the systems. Its total length is NCP.
sPAR	The *shared parameters* of the system. Its total length is NSP. If the `PreferSharedMemory` option is 1, it is stored only once in the Shared Memoryas as it is shared among all the threads/instances.
sPARi	The same as sPAR but for type `int`. Its total length is $NISP$.
ACC	The *accesories* of the systems. Its total length is NA.
ACCi	The same as the *accesories* but for type `int`. Its total length is NIA.
EF	The array to where the event functions are evaluated. Its total length is NE.
IDX	The serial number of the event function in case of succesful event detection.
UDT	User-defined termination. Setting this value to 1 will terminate the integration.
DOIDX	Dense output index (the starting index of the dense output storage). **In case of the successive call of the `Solve()` member function, the dense output index is not automatically reseted to the starting point.** If necessary, it can be done, e.g., in the `Initialisation()` or `Finalisation()` functions.

3.1.1 The right-hand side of the system

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

```
template <class Precision>
__forceinline__ __device__ void PerThread_OdeFunction(\
          int tid, int NT, \
          Precision* F, Precision* X, Precision T, \
          Precision* cPAR, Precision* sPAR, int* sPARi, Precision* 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, note again that the indexing starts from 0 in C++, and for matrix indexing, C++ uses square brackets.

Keep in mind that there is NO bound check for performance reasons. Therefore, double check the indexing and the sizes of each variables. If the `PreferSharedMemory` option is 1, the *shared parameters* (`sPAR` and `sPARi`) are immediately loaded into the Shared Memory that is a really fast on-chip memory type of the 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 bandwidth. However, the excessive usage of Shared Memory is not trivial and can be counterproductive, for details see also Sec. 2.2.1. Observe that via the user programmable *accessories* ACC, the user has great flexibility during the evaluation of the right-hand side. The template parameter `Precision` is prescribed during the creation fo the SolverObject, see Sec. 2.1.1.

3.1.2 The event functions

In the first tutorial example, two events are specified as follows

```
template <class Precision>
__forceinline__ __device__ void PerThread_EventFunction(\
          int tid, int NT, Precision* EF, \
          Precision T, Precision dT, Precision* TD, Precision* X, \
          Precision* cPAR, Precision* sPAR, int* sPARi, Precision* 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. 5.1. The second event function is $F_{E2} = x_1$ 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. 3.1 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 `ACCi`, then it can be used to specify an event corresponding to a maximum number of time steps.

3.1.3 Action after every succesful event detection

After every successful event detection, a special pre-defined user-defined function is called. For the first tutorial example it reads as

```
template <class Precision>
__forceinline__ __device__ void PerThread_ActionAfterEventDetection(\
          int tid, int NT, int IDX, int& UDT, \
          Precision &T, Precision &dT, Precision* TD, Precision* X, \
          Precision* cPAR, Precision* sPAR, int* sPARi, Precision* ACC, int* ACCi)
{
   if ( X[0] > ACC[0] )
      ACC[0] = X[0];

   if ( IDX == 1 )
      ACCi[0]++;

   if ( (IDX ==1 ) && ( ACCi[0] == 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 *accessories*. In case of the example above, the local maxima (local, as it is detected via an event) of the first component of the state variable x_1 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 middle part of the code snippet, a counter for the second event function ($IDX = 1$) is stored into an integer accesory. Next, this counter is used as a condition to store the value of x_2 (index 1) into the second accessory (index 1) at the second successful event detection of the second event function. This example has no particular physical importance, it is just an example to demonstrate the flexibility and strengths of the implemented event detection mechanism. **By setting the variable `UDT` to 1, the integration of the corresponding thread can be terminated.**

An exmaple to demonstrate the very powerful feature of this pre-declared user-defined function relates to impact dynamics. 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 the corresponding tutorial example in Sec. 5.6. 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.

3.1.4 Action after every successful time step

Similarly to Sec. 3.1.3, an action implemented by a control flow inside the following function

```
template <class Precision>
__forceinline__ __device__ void PerThread_ActionAfterSuccessfulTimeStep(\
          int tid, int NT, int& UDT, \
          Precision& T, Precision& dT, Precision* TD, Precision* X, \
          Precision* cPAR, Precision* sPAR, int* sPARi, Precision* 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 3.1 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 is overwritten at every time step even if x_1 increases monotonically. **By setting the variable `UDT` to 1, the integration of the corresponding thread can be terminated.**

3.1.5 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

```
template <class Precision>
__forceinline__ __device__ void PerThread_Initialization(\
          int tid, int NT, int& DOIDX, \
          Precision& T, Precision& dT, Precision* TD, Precision* X, \
          Precision* cPAR, Precision* sPAR, int* sPARi, Precision* ACC, int* ACCi)
{
    T      = TD[0]; // Start the integration from the lower limit of the time domain
    DOIDX  = 0;     // Filling the dense output from the beggining

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

    ACCi[0] = 0; // Event counter of the second event function
}
```

This function is called only once at the beginning of each integration phase. Here, the first line sets the initial time of the integration to the lower limit of the time domain; the second line sets the dense output index to the beginning of the dense output storage. When the `Solver()` member function is called successively one after another, these variables are NOT reset automatically at the end of the integration phases. Therefore, they must be done here if necessary. Next, the first three *accessories* are initialised according to the initial state of the system (in order: the *accessories* store the local maxima of x_1, the x_2 value detected via the second event function, and finally the global maxima of x_1). The last line sets the integer accessory for the event counter to zero.

3.1.6 Finalisation after every integration phase

The function shown below is called after the end of every integration phase. Its purpose and usage is similar as in the case of `PerThread_Initialization()`. 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.

```
template <class Precision>
__forceinline__ __device__ void PerThread_Finalization(\
          int tid, int NT, int& DOIDX, \
          Precision& T, Precision& dT, Precision* TD, Precision* X, \
          Precision* cPAR, Precision* sPAR, int* sPARi, Precision* ACC, int* ACCi)
{
}
```

3.2 Module: Coupled Systems Per-Block

Before proceeding further into the details of the pre-declared user-defined function, the abbreviated names of the used arguments and their descriptions are summarised in Tab. 3.2. **The details of the functions are introduced via the first tutorial example, see Sec. 6.1.**

Table 3.2: Summary of the input arguments of the pre-declared user-defined device functions for module **Single System Per-Thread**

argument	description
sid	The serial number of the instance of the system (indexed from 0 to $NS-1$).
uid	The serial number of the units in an instance (indexed from 0 to $UPS-1$.
F	The array to where the **uncoupled part of the right-hand side** of the system is evaluated, see Sec. 1.4.2 for the details. Its length is the unit dimension UD. Keep in mind that a single unit is assigned to a GPU thread, instead of an instance of the system.
X	The actual state of the unit. Its length is the unit dimension UD.
T	The actual time of the systems. In case of the successive call of the `Solve()` member function, the actual time is NOT automatically reseted. If necessary, it can be done, e.g., in the `Initialisation()` or `Finalisation()` functions.
dT	The actual time step.
TD	The time domain of the integration. There are two values for each system; $TD[0]$ (lower bound) and $TD[1]$ (upper bound). Therefore, its total length is 2.
uPAR	The *unit parameters* of the systems. Its total length is NUP.
sPAR	The *system parameters* of the system. Its total length is NSP. It is automatically stored in the fast on-chip Shared Memory.
gPAR	The *global parameters* of the systems. Its total length is NGP. If the `SharedGlobalVariables` option is 1, it is automatically stored in the fast on-chip Shared Memory.
igPAR	The *integer global parameters* of the systems. Its total length is $NiGP$. If the `SharedGlobalVariables` option is 1, it is automatically stored in the fast on-chip Shared Memory.
uACC	The *unit accesories* of the systems. Its total length is NUA.
iuACC	The *integer unit accesories*. Its total length is $NiUA$.
sACC	The *system accesories* of the systems. Its total length is NSA. It is automatically stored in the fast on-chip Shared Memory.
isACC	The *integer system accesories*. Its total length is $NiSA$. It is automatically stored in the fast on-chip Shared Memory.
EF	The array to where the event functions are evaluated. Its total length is NE.
IDX	The serial number of the event function in case of succesful event detection.
UDT	User-defined termination. Setting this value to 1 will terminate the integration.
DOIDX	Dense output index (the starting index of the dense output storage). In case of the successive call of the `Solve()` member function, the dense output index is not automatically reseted to the starting point. If necessary, it can be done, e.g., in the `Initialisation()` or `Finalisation()` functions.
CPT	The coupling terms H_i, see Sec. 1.4.2. Its total length is NC.
CPF	The coupling factors G_i, see Sec. 1.4.2. Its total length is NC.

3.2.1 The uncoupled part of the right-hand side, the coupling terms and the coupling factors of the system

The uncoupled part of the righ-hand side of the system, the coupling terms and the coupling factors have to be defined in the pre-declared user-defined device function `CoupledSystems_PerBlock_OdeFunction()`. For the first tutorial case, its code snippet is as follows

```
template <class Precision>
__forceinline__ __device__ void CoupledSystems_PerBlock_OdeFunction(\
          int sid, int uid, \
          Precision* F, Precision* X, Precision T, \
          Precision* uPAR, Precision* sPAR, Precision* gPAR, int* igPAR, \
          Precision* uACC, int* iuACC, Precision* sACC, int* isACC, \
          Precision* CPT, Precision* CPF)
{
    F[0] = X[1] - gPAR[0]*X[0];
    F[1] = -uPAR[1]*X[1] + X[0] - X[0]*X[0]*X[0] + uPAR[2]*cos(uPAR[0]*T) - gPAR[0]*X[1];

    // i=0...NC
    CPT[0] = X[0];
    CPF[0] = 1.0;

    CPT[1] = X[1];
    CPF[1] = 1.0;
}
```

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 functions is very similar as in the case of MATLAB. But still, note again that the indexing starts from 0 in C++, and for matrix indexing, C++ uses square brackets. Moreover, the coupling terms H_i and factors G_i have to be computed as well, see Sec. 1.4.2 for details and reasons. Comparing this pre-declared user-defined device function with the system example shown in Sec. 1.4.2, the values of F, CPT and CPF correspond to the functions f, h and g, respectively.

Keep in mind that there is NO bound check for performance reasons. Therefore, double-check the indexing and the sizes of each variable. The *system parameters* (`sPAR`, `sACC`, `isACC`) are immediately loaded into the Shared Memory. The *global parameters* (`gPAR`, `igPAR`) are loaded into the Shared Memory only if the `SharedGlobalVariables` option is set to 1. Again, Shared Memory is a really fast on-chip memory type of the GPUs (similarly to CPUs, these memories are caches of GPUs). A single request from a shared memory can be "broadcasted" to every thread further reducing the pressure on memory bandwidth. However, the excessive usage of Shared Memory is not trivial and can be counterproductive, for details see also Sec. 2.2.2. Observe that via the user-programmable *accessories* `uACC`, `iuACC`, `sACC` and `isACC`, the user has great flexibility during the evaluation of the functions f, h and g. The template parameter `Precision` is prescribed during the creation fo the SolverObject, see Sec. 2.1.1.

3.2.2 The event functions

In the first tutorial example, a single event is specified as follows

```
template <class Precision>
__forceinline__ __device__ void CoupledSystems_PerBlock_EventFunction(\
          int sid, int uid, Precision* EF, \
          Precision     T, Precision     dT, Precision*   TD, Precision*     X, \
          Precision* uPAR, Precision*  sPAR, Precision* gPAR,       int* igPAR, \
          Precision* uACC,       int* iuACC, Precision* sACC,       int* isACC)
{
    EF[0] = X[1];
}
```

There are only unit scope event functions; that is, an event specified according to the trajectory of a unit. System-wide detection of a special event is not supported in the present version. The event function is simply $F_{E1} = x_{i,2} = y_i$ evaluated into the variable $EF[0]$ meaning that a special point is detected if $x_{i,2} = y_i = 0$. Since $\dot{x}_{i,2} = x_{i,1}$ ($\dot{y}_i = x_i$), this means the local maxima or minima of the variable $x_{i,1}$, see Sec. 6.1. Here i is the serial number of the units. For simplicity, in the first tutorial example in Sec. 6.1, the first and second component of a unit is denoted by x and y, respectively. the Observe that most of the possible arguments listed in Tab. 3.1 are passed to this function providing a great flexibility to define an event function. For instance, the number of the successful time steps can be registered in a user-programmable accessory variable `iuACC`; then it can be used to specify an event corresponding to a maximum number of time steps.

3.2.3 Action after every succesful event detection

After every successful event detection, a special pre-defined user-defined device function is called. For the first tutorial example it reads as

```
template <class Precision>
__forceinline__ __device__ void CoupledSystems_PerBlock_ActionAfterEventDetection(\
          int sid, int uid, int IDX, int& UDT, \
          Precision&    T, Precision&    dT, Precision*   TD, Precision*     X, \
          Precision* uPAR, Precision*  sPAR, Precision* gPAR,       int* igPAR, \
          Precision* uACC,       int* iuACC, Precision* sACC,       int* isACC)
{
   iuACC[0]++;

   if ( iuACC[0] == 2 )
   {
      UDT = 1;
      uACC[2] = X[0];
   }
}
```

Inside this function, many special operations can be performed. For instance, the trajectories can be manipulated, or some properties related to an event can be stored into *accessories*. In the case of the example above, an integer unit accessory is used to accumulate the number of successful event detection (first line). The control flow checks whether the second event is detected or not. If detected, the integration is stopped by setting the user-defined termination to $UDT = 1$, and store the first component of the units of the system into a unit accessory. As the event function represents a local maximum (also specified by the event direction solver option, see the details in the tutorial example) of the first component of a unit $x_{i,1} = x_i$, the integration of the whole system is stopped when a single unit first reaches the second local maximum (even if the other units have not been reached their second local maximum).

An example to demonstrate the very powerful feature of this pre-declared user-defined device function relates to impact dynamics. 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 a tutorial example in Sec. 5.6 for uncoupled systems. Since any kind of control flow algorithm can be implemented inside the function body, usefulness of this function depends on the imagination of the user.

3.2.4 Action after every successful time step

Similarly to the Sec. 3.2.3, an action implemented by a control flow inside the following function

```
template <class Precision>
__forceinline__ __device__ void CoupledSystems_PerBlock_ActionAfterSuccessfulTimeStep(\
          int sid, int uid, int& UDT, \
          Precision&    T, Precision&    dT, Precision*   TD, Precision*     X, \
          Precision* uPAR, Precision*  sPAR, Precision* gPAR,       int* igPAR, \
          Precision* uACC,       int* iuACC, Precision* sACC,       int* isACC)
{
    if ( uid == 0 )
    {
        if ( sACC[0] > dT ) // Update minimum dT
            sACC[0] = dT;
        if ( sACC[1] < dT ) // Update maximum dT
            sACC[1] = dT;

        isACC[0]++;          // Accumulate number of time steps
    }
}
```

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 3.1 and the corresponding discussion. The only difference is that properties related to event handling are not passed as arguments. The above example stores the maximum and minimum values of the time steps, and the number of required time steps. Observe how the uint ID *uid* is used to avoid writing the same values of dT to the corresponding *accessories* multiple times, and to avoid the update of the number of time step accumulator more times than necessary.

3.2.5 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

```
template <class Precision>
__forceinline__ __device__ void CoupledSystems_PerBlock_Initialization(\
          int sid, int uid, int& DOIDX, \
          Precision&    T, Precision&    dT, Precision*   TD, Precision*     X, \
          Precision* uPAR, Precision*  sPAR, Precision* gPAR,       int* igPAR, \
          Precision* uACC,       int* iuACC, Precision* sACC,       int* isACC)
{
    if ( uid == 0 )
    {
        sACC[0]   = dT; // Minimum dT
        sACC[1]   = dT; // Maximum dT
        isACC[0] = 0;  // Number of time steps
        T = TD[0];  // Reset the starting point of the simulation from the lower limit of
            the time domain
        DOIDX = 0; // Reset the start of the filling of dense output from the beggining
    }

    uACC[2]   = X[0];   // End state of the unit first reach the second local maximum;
        initial state otherwise
    iuACC[0] = 0;       // Event counter
}
```

This function is called only once at the beginning of each integration phase. Inside the conditional, the *accessories* are initialised used inside the pre-declared user-defined devices function discussed in Sec. 3.2.4. Next, the initial time of the integration to the lower bound of the time domain; and the dense output index to the beginning of the dense output storage is properly set. When the `Solver()` member function is called successively one after another, these variables are not reset at the end of the integration phases automatically. Therefore, they must be done here if necessary. Next, the accessories to store the second local maximum and the event counter is initialised.

3.2.6 Finalisation after every integration phase

The function shown below is called after the end of every integration phases. Its purpose and usage is similar as in the case of `CoupledSystems_PerBlock_Initialization()`. 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.

```
template <class Precision>
__forceinline__ __device__ void CoupledSystems_PerBlock_Finalization(\
          int sid, int uid, int& DOIDX, \
          Precision&    T, Precision&    dT, Precision*   TD, Precision*     X, \
          Precision* uPAR, Precision*  sPAR, Precision* gPAR,       int* igPAR, \
          Precision* uACC,       int* iuACC, Precision* sACC,       int* isACC)
{
}
```

3.2.7 Data race condition and atomic operations

Keep in mind that the pre-declared user-defined device functions are evaluated by every unit (GPU thread) in the system simultaneously. Therefore, writing a value into a *system parameter* or *global parameter* can lead to a data race condition where the order of the writing is undefined and can lead to unexpected results. The reason is that only a single instance of these variables is stored, and they are shared by all the threads. Reading such a memory location is not a problem. Moreover, if the variable is in the Shared Memory, the value is broadcasted to every thread that makes the reads very efficient. In contrast, the Global Memory read from a single location by many threads leads to very inefficient utilisation of the memory subsystem, see Sec. 4.6.1.

One solution for the writing problem is to select only a specific thread to perform the writing operations. The integer function argument *uid* (unit ID) can be used to select a single unit to update a *system parameter* or a *global parameter*. Examples are already provoded in Sec. 3.2.3, Sec. 3.2.4 and Sec. 3.2.5. For completeness, the system ID *sid* is also passed as a function argument into the pre-declared user-defined device functions.

Unfortunately, the above technique cannot be used in every situation. For instance, during the calculation of the sum, minimum, maximum or other reduction operations on the state variable across the units. Since, all the threads of the corresponding units have to take part in the accumulation process. In this case, atomic operations need to be performed. Calling an atomic function by a thread, the code will block the usage of that particular memory location by another thread. This also means that the atomic operations are serialised; thus, the user can experience huge performance loss if many atomic operations are used. The list of the available atomic functions supported by CUDA is listed here:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Unfortunately, the supported atomic functions are architecture-dependent. For instance `atomicAdd()` function for data type `double` is supported only compute capability 6.0 (CC6.0) or above. In this specific case, CUDA documentation provides a function to workaround this problem with the help of `atomicCAS()` function; however, the many type conversions makes the code extremely inefficient. Therefore, I suggest using only hardware supported atomic functions, and try to find another kind of workaround of the problem if necessary.

Chapter 4

Performance considerations

In case of the module **Single System Per-Thread**, for small systems ($SD \approx 10 - 20$), with the "default" setup introduced via the first tutorial example, the performance will likely suffer **NO** harm. The only issue the user has to take care of is to set a sufficiently large number of threads in order to fully utilise the GPU's arithmetic processing units (approximately $NT > 5000 - 10000$). For larger systems, the improper setup can have a significant impact on the performance. In such systems, the available register memory (fastest memory type on the GPU) for a single problem/thread is not enough to store all the required variables. Thus, it is very likely that some variables are "spilt" back into the slowest Global Memory resulted in the requirement of slow memory transactions. The larger the number of variables needs to be loaded periodically from the Global Memory, the higher the chance that the application becomes bounded by the memory bandwidth instead of by the by peak processing power.

For large systems, if possible, use the module **Coupled Systems Per-Block**. For example, when the system comes from the semi-discretisation of a partial differential equation, it can usually be interpreted as a large system of spatially coupled subsystems. The advantage is that the solution of the complete system is distributed to many threads increasing the total available register memory for a single problem. However, if the coupling matrix is full and large, the application can be still memory bandwidth limited due to the matrix-vector multiplication. Moreover, for coupled systems, some amount of performance can be lost because of the possible idle threads.

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-defined device functions (e.g., the right-hand side) efficiently.

4.1 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 are organised into thread blocks. The template variable `TPB` (coupled systems) and the solver option `ThreadsPerBlock` (single system) introduced in Sec. 2 defines the number of threads that can reside in a single block. In case of the first tutorial example of the module **Single System Per-Thread**, the number of threads is $NT = 46080$, and the block size is 64; that is, the number of the blocks is $46080/64 = 720$.

4.2 The parallelisation strategy

The different MPGOS modules use very different parallelization strategies (distribution of work to GPU threads). For some modules, it is crucial to understand the details for efficient application—this section devoted to their description.

4.2.1 Module: Single System Per-Thread

This module uses a very simple parallelisation technique. A single instance of the ODE system is assigned to a single GPU thread. That is, the different threads work on a different set of data (parameters, initial conditions, time domains or accessories). We call this technique per-thread approach also emphasised by the name of the module. The technique is depicted in Fig. 4.1. In this module, no special care has to be taken for workload distribution. The only thing the user has to keep in mind is that the option `ThreadsPerBlock` (block size) must be integer multiple of the warp size (32 in the present CUDA architectures), and to launch a sufficiently large number of threads NT to fully utilise the GPU, see some more details in the forthcoming sections.

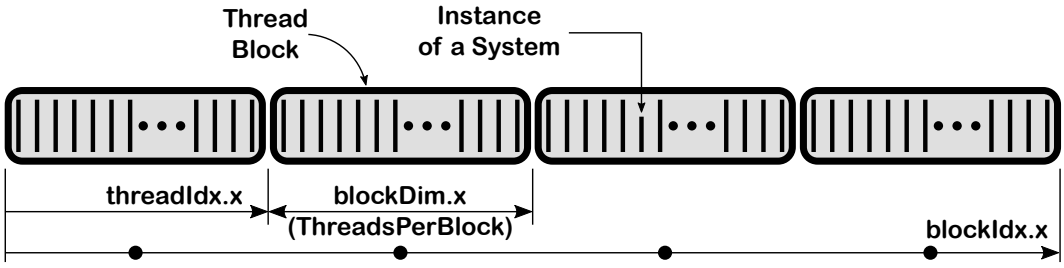

Figure 4.1: Parallelisation strategy for module **Single System Per-Thread**.

4.2.2 Module: Coupled Systems Per-Block

The basic principle of the parallelisation strategy of this module is that one or more instances (green rectangles) are assigned to a single thread block (red rectangles), see the Fig. 4.2. One unit (s subsystem) in an instance is assigned to a single GPU thread (denoted by the black thin lines). If the multiply of the number of instances associated to a thread block and the number of the units in a system is greater than the number of threads in a block ($SPB \times UPS > TPB$), one thread has to deal with a multiple number of units. This results in the so-called multiple block launches, shown by the block with the white background in Fig. 4.2. The number of the required block launches is calculated automatically by the solver according to the template parameter values of SPB, UPS and TPB. The number of the block launches defines the total

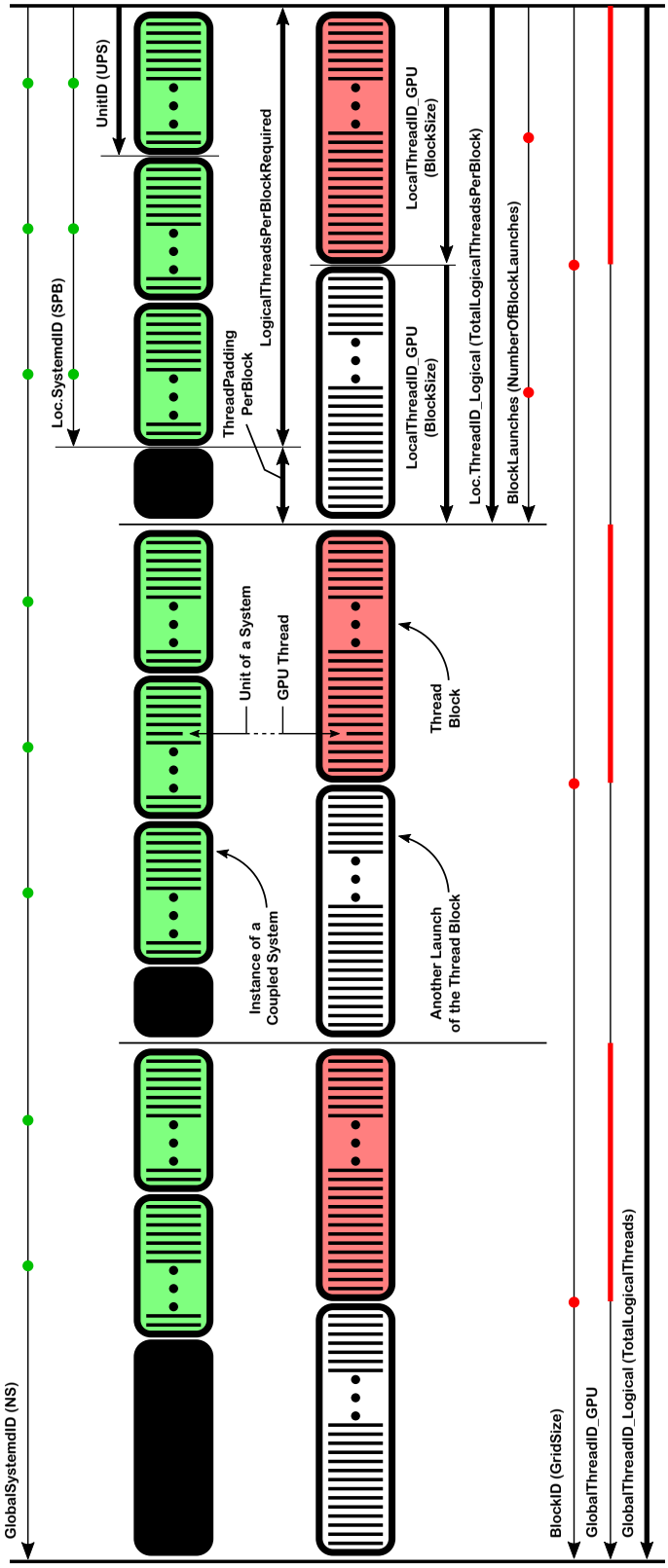

Figure 4.2: Parallelisation strategy for module **Coupled Systems Per-Block**.

logical threads of a block. Naturally, the total number of units associated with a thread block and the total number of logical threads are different (in general). The difference is called thread padding representing idle threads during a simulation, which causes a certain amount of performance loss just because no work is distributed to these threads. Such portion of the logical threads is depicted by the black rectangles in Fig. 4.2. One important performance consideration from the user is the minimisation of the black portions by properly adjust the values of SPB, UPS and TPB.

4.3 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 Sec. 4.5 and Sec. 4.8. 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. 1.6), 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. 1.6 on how to query this information. This means that a total number of blocks of, e.g., 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 instance of a system 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 first tutorial example of the module **Single System Per-Thread** in Sec. 5.1, the total number of the blocks during one simulation launch is $23040/64 = 360$, where 64 is the number of threads in a block (the block size). 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.

4.4 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.

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

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 Chap. 3.

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

4.5 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. 1.6. The only exception is the number of blocks that can simultaneously reside in an SM discussed already in Sec. 4.3. The information has to be obtained from the CUDA Programming Guide:

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

4.6 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 been 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 cache memories (and the Shared

Memory) per threads 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 a highly efficiency code. This means mainly an implementation of an efficient right-hand side.

4.6.1 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 [2]. 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 the 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.

4.6.2 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. 1.6. 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. 4.3. Thus, one cannot rely on the cooperation between threads residing in different blocks. In case of newer architectures, this is eased by the introduction of co-operative groups; however, this is irrelevant for the program package MPGOS.

The program package MPGOS offers the possibility to exploit the advantage of Shared Memory by putting variables common to all threads/systems into the Shared Memory, for the technical details see Chap. 2. 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 some tutorial example, this technique does not have a significant impact as the number of the shareable parameters are limited, 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. (1.1) 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 instances. 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. (1.1). 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**

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

stored as a linear sequence of shared parameters. This needs special care of indexing by the user in the pre-declared user-defined device functions.

Another example of the possible use of Shared Memory is the case of coupled systems where the **coupling marix** is the same for all systems, see Sec. 1.4.2. Storing the coupling matrix in the Shared Memory can significantly ease the pressure on the Global Memory. Keep in mind that the matrix-vector multiplications are usually memory bandwidth limited operations. For the module **Coupled System Per-Block**, the user has the opportunity to load the coupling matrix into the Shared Memory, see also Sec. 4.7.

Although the use of Shared Memory can be extremely powerful, its overusing can lead to performance decrease. If the thread blocks need a large amount of Shared Memory, the residing number of blocks can be decreased significantly greatly limiting the available "parallelism" on a Streaming Multiprocessor. One possible compensation is to use a large number of threads in a single block; however, in case of the coupled systems, the required synchronisation procedures can block the progress of certain systems. **It is advised that the user experience the Shared Memory related solver options according to the specific problem.**

4.6.3 Registers

First of all, every arithmetic operations can be done only on data residing in the Registers; otherwise, they must be loaded from the Shared or Global Memory first. 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. 1.6. 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-defined device 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 also scarce resources. However, it is much higher per a thread compared to CPUs. Allocating large arrays or structures inside a pre-declared user-defined device function (e.g., in the right-hand side evaluation), the compiler will automatically place them into the slow Global Memory if there are not enough available registers. This phenomenon is called register spilling and it is 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, the total number of threads residing in an SM can be significantly reduced, for details see Sec. 4.8. 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. 4.6.

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 [3]; 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
```

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 own 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 computationally more intensive systems, the number of registres should be 128 or above.

Another important rule-of-thumb is to minimise the memory allocations inside the pre-declared user-defined device functions. For instance, do not allocate big arrays rarely used just because it provides a more clear code structure. Try to perform given computations with minimal register usage. Otherwise, to be able to perform new computations, some older intermediate results have to be spilt into the slow Global Memory. When these data are required later, they must be loaded again from the Global Memory even though they were already in the registers before.

As final advice, try to exploit temporal locality during the evaluation of the pre-declared user-defined device functions. That is, try to use variables close to each other to prevent their reload from Global Memory. If a variable is used seldomly in scattered parts of the code, it is very likely that it is spilt back to the Global Memory regularly (and loaded back regularly) if the number of the Registers are not enough. In addition, try to reuse allocated arrays for different computational purposes.

4.7 Storage techniques of the coupling matrices

In general, the option for storing the coupling matrices in Shared Memory is always available for the user, see Sec. 2.2.2 for details. However, for a large number of coupled systems, and for full-matrices, the available amount Shared Memory might be not enough. In this case, there is nothing to do; the matrices have to be accessed through the slow Global Memory and hope that the memory bandwidth limited application still performs well.

For structured matrices, MPGOS offers the possibility for efficient storage. Consider a band matrix with a certain bandwidth radius, see the left-hand side of Fig. 4.3. In this case, only the diagonals of the matrix are stored in a fashion depicted in the middle panel of Fig. 4.3. Observe that the diagonals stored circularly, even if the matrix itself is not circular. **The user have to fill the unused elements by zero.** If the matrix is a band matrix, circular and having the same values in each diagonal, further storage capacities can be saved. In this case, only a single number has to be stored for each column of the middle part of Fig. 4.3. The results are depicted in the right-hand side of the figure. These kind of coupling matrices are always stored in the Shared Memory regardless of the actual solver option. The condensed representations of such

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

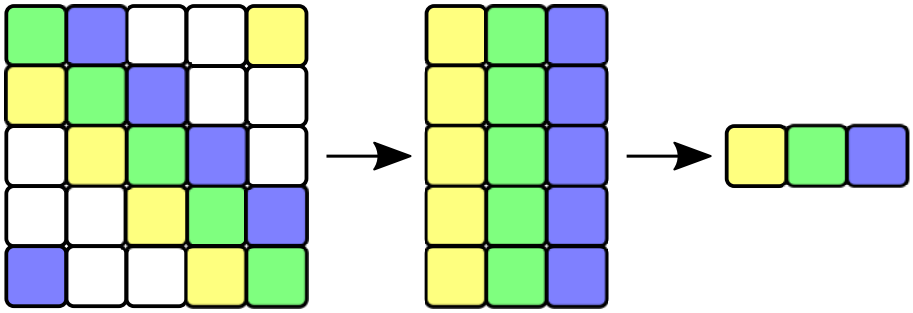

Figure 4.3: Coupling matrix storage techniques for module **Coupled Systems Per-Block**.

a matrices can significantly increase the performance as the coupling matrix is much likely to be fit into the fast Shared Memory. Moreover, the number of arithmetic computations are also decreased significantly. It is especially true for large matrices with small bandwidth radius.

4.8 Resource limitations and occupancy

The number of simultaneously residing threads and blocks in an SM is not limited only by the hardware restrictions (Sec, 4.5) 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 if $R_T = 64$ (compiler option); the maximum number of residing threads in this case are $N_{TSM} = 1024$. The hardware limitation is 2024 (see Sec. 1.6); 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" [4] 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

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

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 or a warning is printed.

4.9 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 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[5]. 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.

4.10 Profiling

The program package MPGOS provides a well prepared Linux shell script `Profile.sh` 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

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

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 first tutorial example of modeule **Single System Per-Thread**. 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).
- 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*). This 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%

Chapter 5

MPGOS tutorial examples for module Single System Per-Thread

This section serves as an additional material to highlight the flexibility and features of the module **Single System Per-Thread** of the program package MPGOS. The list of the tutorial examples will be continuously extended.

During the tutorial examples, code snippets are NOT provided as all the source codes can be found in the corresponding directories. Thus only the detailed description of the problem, the main aims and task, and the results are discussed. Each tutorial example have a main `.cu` file and a system definition file with extension `.cuh`.

5.1 Tutorial 1: The Duffing equation as a first tutorial example

Throughout the first tutorial example, the features and capabilities of the program package is introduced by using the well-known Duffing oscillator written in a simplified form as

$$\dot{x}_1 = x_2, \tag{5.1}$$

$$\dot{x}_2 = x_1 - x^3 - kx_2 + B\cos(t), \tag{5.2}$$

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 $SD = 2$. 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 instance of the 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*. It could be hard-coded; however, we defined it as a shared parameter for demonstration purposes. In summary, there is 1 *control parameter* and 1 *shared parameter* (of floating-point type). The employed precision of type `double`. For the sake of simplicity, the angular frequency of the excitation is unity. Consequently, the period of the excitation is exactly $T = 2\pi$.

Objectives

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 endpoints of each integration phase. For a thorough discussion on how to define the right-hand side of the system and how to set up the adaptive time marching, see Sec. 3.1.1 and Sec. 2.2. Note that in the `PerThread_Initialization()` pre-declared user-defined device function, the proper reset of the initial time and the starting index of the dense output must be done.
- 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 again Sec. 2.2 and Sec. 3.1.1.
- 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* (floating point type) for each instance 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. 3.1.3.
- The global maxima of each integration phases are also stored. This needs an additional, third *accessory* (floating point type). The process is discussed in Sec. 3.1.4.
- Finally, $N_{DO} = 200$ points are stored for each trajectory as a dense output. Only the results of the last integration phase can be accessed.

Explanation of the source codes

The majority of the above points can be set up via the pre-declared user-defined device functions, which have an important part in the definition of the system and its behaviour. However, the whole computation must be managed by the `main()` function. The code can be found in the directory of the tutorial example. Here, we will examine the code thoroughly, step-by-step (only for this first tutorial example).

- 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 Chap. 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 (in general, this name is arbitrary). The name of the second header file of the interface of the module **Single System Per-Thread** 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 specifying its namspece via the scope operator `std::`. The next define statement serves as a simplified modification of the algorithm template parameter (now `RKCK45`) and the precision of the floating point numbers (now `double`). Finally, size-related template parameters are declared as constant variables.

- The next three lines define the prototypes of auxiliary functions that are not part of the program package. However, they make the code more readable and its usage more easy. With them, e.g., the fill-up of the SolverObject and the save of the required results can be performed separately from the main control logic of the code.

- 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 $NP = 46080$ (`NumberOfProblems`), see the above discussion. Since the number of threads in the SolverObject is $NT = 23040$, two simulation launches will be necessary to process all the parameters ($NP/NT = 2$), see also the details below. The block size can be an important parameter for performance. Note again that it should be an integer multiple of 32 (the warp size), for the details see Chap. 4. If the user has no solid confidence, use the default value (warp size).

- 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*). Observe how the pre-defined `PRICISION` is used to declare the variables and set their floating-point precision.

- 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). Its type is the `vector` container of the Standard Template Library.

- Only one SolverObject 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 NP (total number of instances of the ODE system) and the used threads NT in the SolverObject. If there is a remainder during the division, the number of simulation launches is increased by 1 to process the remaining problems. In such cases, do not forget to adjust the option `ActiveNumberOfThreads` to the remainder before the last simulation launch; this is not the case in this 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 exactly $NP/NT = 2$).

- The first step for each simulation launch is to fill the SolverObject 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 tutorial example. 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 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 finishing the copy of 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 SolverObject to continue the integration phases if the time domain is set to $t_0 = 0$ and $t_1 = 2\pi$ for every instance of the ODE system. **Only the starting time of the integration needs to be reset via the `Initialisation()` pre-declared user-defined device function as it is already discussed in Chap. 3.** 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 or the previous call of the `Solve()` instruction, even though the CPU issued these tasks 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 1024 number of tasks immediately 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 them).

- The next 32 integration phases, organised in a loop, is served to save the data of the converged trajectories into a file in each phase. 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 this 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 and final integration phase 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 SolverObject with data. Observe that the difference between the variables `k_begin` and `k_end` is exactly the number of the used threads NT in

the SolverObject. The loop fills the data corresponding to the `TimeDomain` (fixed for each thread/instance), `ActualState` (initial condition, fixed also for each threads), `ActualTime` (for completeness, its value must be reseted via the pre-declared user-defined device functions), `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` (for completeness, initialised by zero, theirs values are also initialised at the beginning of each intgeration phase via the pre-declared user-defined device functions). Observe that the `SharedParameters` have to be set only once (outside the loop). In this tutorial example, there is only one *shared parameter* (excitation amplitude B); that is, there is only one corresponding `SetHost()` function call.

- Auxiliary function `SaveData()`: it save the required data to the specified file. It takes the SolverObject as an argument by reference (for performance reasons). Using the `GetHost<>()` member function, the sole `ControlParameters` and the `SharedParameters` are written into the first two columns of the file. 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 columns in the file, the three `Accessories` are stored.

Results

It can be seen that the main strength of the program package is that one needs only a two hundred lines long code block to perform a detailed parameter study of a dynamical system. The resulted bifurcation diagram is shown in Fig. 5.1, 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 purely to the computations, the trasient part of the simulation took only 4.9 s.

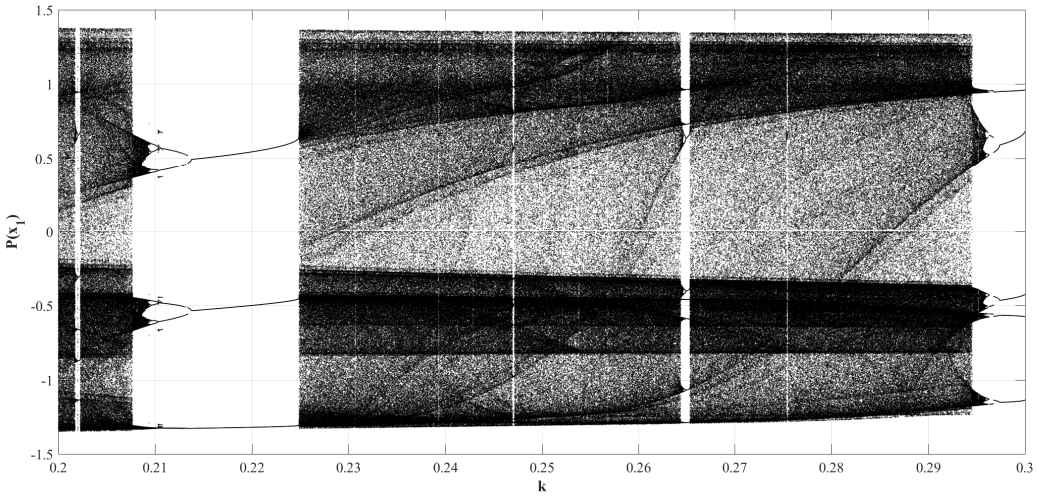

Figure 5.1: The bifurcation diagram of the Duffing oscillator; that is, the first component of the Poincaré section $P(x_1)$ as a function of the control parameter k. The control parameter is linearly distributed between 0.2 and 0.3 with a resolution of 46080.

5.2 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

$$\dot{y}_1 = 10.0\,(y_2 - y_1), \tag{5.3}$$
$$\dot{y}_2 = p\,y_1 - y_2 - y_1\,y_3, \tag{5.4}$$
$$\dot{y}_3 = y_1\,y_2 - 2.666\,y_3, \tag{5.5}$$

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. (5.4), 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.

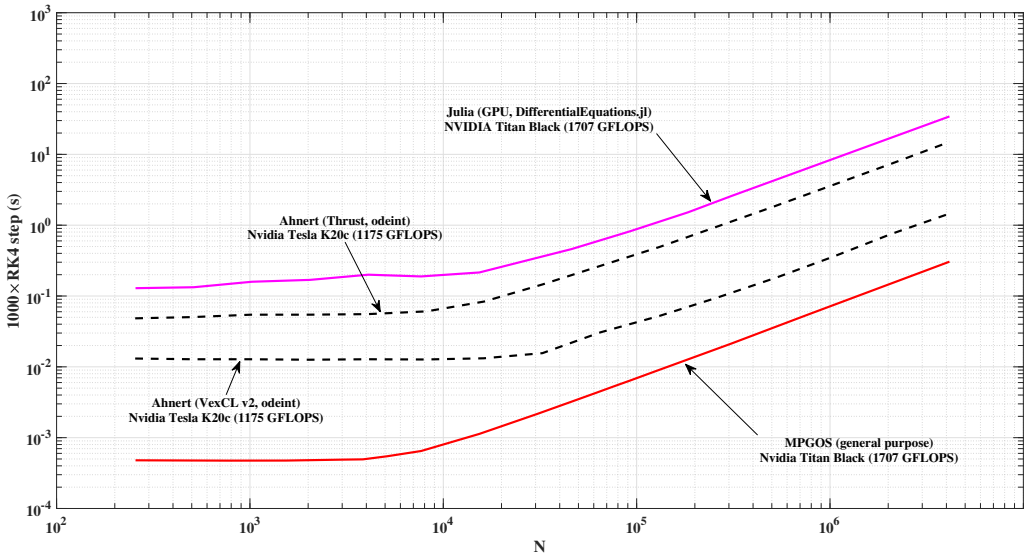

Figure 5.2: Comparison of the performance of different program packages; that is, the runtimes of 1000 steps of the solution of N Lorenz systems with the simple 4^{th} order Runge–Kutta scheme.

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[1] written in C++. The runtimes are digitalised from the publication of Ahnert et. al[2] from Fig. 7.1. The second program package is the DifferentialEquations.jl[3] written in the rapidly developing Julia language. The code of Julia is written by Dániel Nagy[4] according to the tutorial example available on GitHub[5]. Figure 5.2 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. A more detailed comparison with the packages odeint and Julia including CPU versions is available at a GitHub repository[6]. The codes of odeint is written by Lambert Plavecz[7].

[1] www.odeint.com
[2] 2014 Solving Ordinary Differential Equations on GPUs, Numerical Computations with GPUs pp. 125-157
[3] https://docs.juliadiffeq.org/stable/index.html
[4] n.nagyd@gmail.com
[5] https://github.com/JuliaDiffEq/DiffEqGPU.jl
[6] https://github.com/nnagyd/ode_solver_tests
[7] plaveczlambert@gmail.com

5.3 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 by comparing the first tutorial example with its simplified version where no event handling is used, and no special properties of the trajectories are stored. This third tutorial example is the simplified version of the first tutorial example. It keeps only the pre-declared member function of the right-hand side of the ODE. Therefore, only the runtimes of the computations are presented here: 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.

5.4 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

$$\left(1-\frac{\dot{R}}{c_L}\right)R\ddot{R}+\left(1-\frac{\dot{R}}{3c_L}\right)\frac{3}{2}\dot{R}^2=\left(1+\frac{\dot{R}}{c_L}+\frac{R}{c_L}\frac{d}{dt}\right)\frac{(p_L-p_\infty(t))}{\rho_L}, \tag{5.6}$$

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

$$p_\infty(t) = P_\infty + P_{A1}\sin(\omega_1 t) + P_{A2}\sin(\omega_2 t+\theta), \tag{5.7}$$

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

$$p_G + p_V = p_L + \frac{2\sigma}{R} + 4\mu_L\frac{\dot{R}}{R}, \tag{5.8}$$

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

$$p_G = \left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\left(\frac{R_E}{R}\right)^{3\gamma}, \tag{5.9}$$

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. (5.6)-(5.9) 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 attention have to be taken to handle the time domain during the simulations.

System (5.6)-(5.9) is rewritten into a dimensionless form defined as

$$\dot{y}_1 = y_2, \tag{5.10}$$

$$\dot{y}_2 = \frac{N_{\mathrm{KM}}}{D_{\mathrm{KM}}}, \tag{5.11}$$

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

$$\begin{aligned} N_{\mathrm{KM}} = (C_0+C_1y_2)\left(\frac{1}{y_1}\right)^{C_{10}} - C_2\left(1+C_9y_2\right) - C_3\frac{1}{y_1} - C_4\frac{y_2}{y_1} - \left(1-C_9\frac{y_2}{3}\right)\frac{3}{2}y_2^2 \\ - \left(C_5\sin(2\pi\tau)+C_6\sin(2\pi C_{11}\tau+C_{12})\right)\left(1+C_9y_2\right) \\ - y_1\left(C_7\cos(2\pi\tau)+C_8\cos(2\pi C_{11}\tau+C_{12})\right), \end{aligned} \tag{5.12}$$

and

$$D_{\mathrm{KM}} = y_1 - C_9 y_1 y_2 + C_4 C_9. \tag{5.13}$$

The coefficients are summarised as follows:

$$C_0 = \frac{1}{\rho_L}\left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{5.14}$$

$$C_1 = \frac{1-3\gamma}{\rho_L c_L}\left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\frac{2\pi}{R_E\omega_1}, \tag{5.15}$$

$$C_2 = \frac{P_\infty - p_V}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{5.16}$$

$$C_3 = \frac{2\sigma}{\rho_L R_E}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{5.17}$$

$$C_4 = \frac{4\mu_L}{\rho_L R_E^2}\frac{2\pi}{\omega_1}, \tag{5.18}$$

$$C_5 = \frac{P_{A1}}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{5.19}$$

$$C_6 = \frac{P_{A2}}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{5.20}$$

$$C_7 = R_E\frac{\omega_1 P_{A1}}{\rho_L c_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{5.21}$$

$$C_8 = R_E\frac{\omega_1 P_{A2}}{\rho_L c_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \tag{5.22}$$

$$C_9 = \frac{R_E\omega_1}{2\pi c_L}, \tag{5.23}$$

$$C_{10} = 3\gamma, \tag{5.24}$$

$$C_{11} = \frac{\omega_2}{\omega_1}, \tag{5.25}$$

$$C_{12} = \theta. \tag{5.26}$$

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 SolverObject.

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. 5.3, 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. 5.3. Observe that the time scales can be very different near the local maximum and the local minimum.

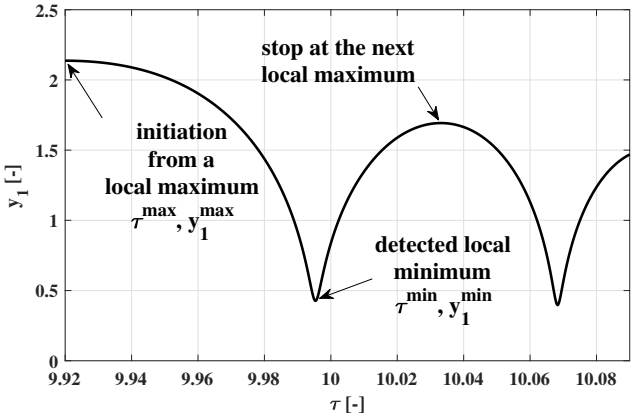

Figure 5.3: Demonstration of a bubble collapse via a dimensionless bubble radius vs. time curve. An integration start from a local maximum y_1^{max} at time instant τ^{max} and ends at the next local maximum. During the integration, the local minimum y_1^{min} and its time instant τ^{min} is also determined.

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 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. 5.1. 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 5.1: 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.

Remarks on the configuration of the SolverObject

The main idea behind the configuration of the SolverObject is as follows. 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 SolverObject 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 $SD = 2$ 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 $NCP = 21$, 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 the code more simple. Thus the number of the shared parameters is $NSP = 0$. This has no impact on the performance as our kernel function is still compute bound.

The number of events is $NE = 1$, only the local maxima of y_1 is detected. Finally, the number of the floating-point type user-programmable *accessories* is $NA = 4$: 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. Moreover, an *integer accessory* is also allocated to be able to register the number of the detected events and to be able to stop the simulation immediately at the first event by a user-defined termination.

Remarks on the pre-declared user-defined device functions

In the implementation of the `PerThread_OdeFunction()` only the first 13 `cPAR` variables are used. Observe also that the frequently used $1/x_1$ is pre-computed (`rx1`). This is extremely important as the division operation has very high latency and very low arithmetic throughput; thus, such operations can generate a long execution dependecy stall.

The accessories are initialised with the initial conditions at the beginning of every integration phase inside the function `PerThread_Initialization()`. The integer accessory to count the detected event is also reset here. Observe that here **the `ActualTime` should NOT be reset to zero**, as the integration has to be continued from where the previous integration phase has been stopped. This is due to the non-periodic nature of the external driving of the system. In general, it can be quasiperiodic in contrast to the Duffing oscillator introduced in the first tutorial example. Accordingly, the end of the time domain is set to a very high number; namely, 10^{10}, the simulation will be stopped by the event. In the function `PerThread_ActionAfterEventDetection()`, the sole *integer accessory* is incremented, and the simulation is stopped by setting the user-defined termination to $UDT = 1$ when the number of the detected events reached the value of 1.

Only the properties of 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.

Results

In Fig. 5.4, 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 f_1 and f_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. 5.4). The employed numerical scheme was the Runge–Kutta–Kash–Carp. Both the absolute and relative tolerances were 10^{-10}.

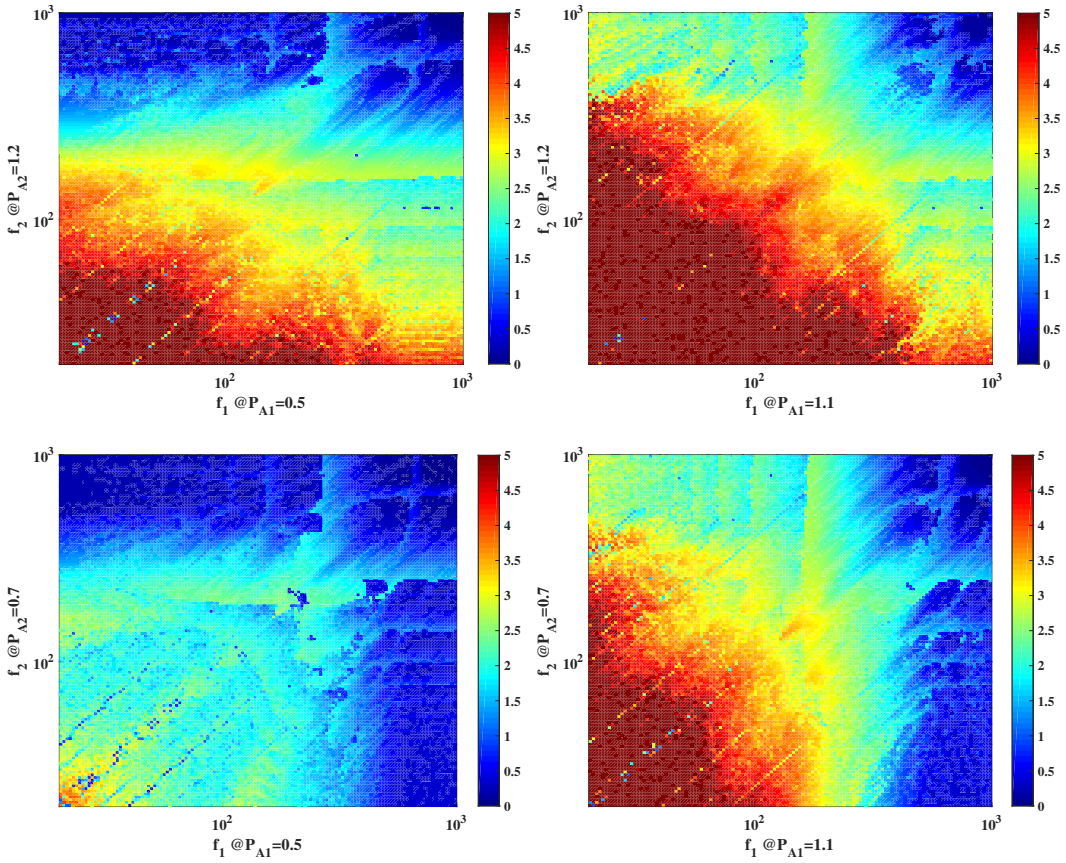

Figure 5.4: Four dimensional parameter scan of the relative expansion of an oscillating single spherical bubble.

5.5 Tutorial 5: large time scale differences

This tutorial example serves to demonstrate the effectiveness of MPGOS over other available program packages for a problem where the time scales of the solutions can vary between orders of magnitude. The ODE system is exactly the same as in case of the fourth tutorial example in Sec. 5.4 (the Keller–Miksis equation); thus, it is not repeated here. The main difference is the employed single frequency driving ($P_{A2} = 0$, $f_1 = f$). The objective is to construct a frequency response curve, where the maximum of the first component y_1^{max} of the converged solution is plotted as a function of the driving frequency f, see Fig. 5.5. The effect of the resolution N of the driving frequency on the runtime is the keen interest here. The value of N is also the number of the instances of the ODE system the GPU solved simultaneously ($N = NT$). The driving pressure amplitude $P_{A1} = P_A = 1.5\,\text{bar}$. In the dimensionless form of the single-frequency driven system, the period of the external forcing is $T = 1$ in the dimensionless time co-ordinate τ. Thus, the system became time-periodic similarly to the Duffing oscillator, see Sec. 5.1. The first 1024 integration phases are regarded as transient and discarded, and the maximum value of y_1 is determined from the next 64 integration phases. One integration phase means the integration of the system between $\tau \in (0, 1)$.

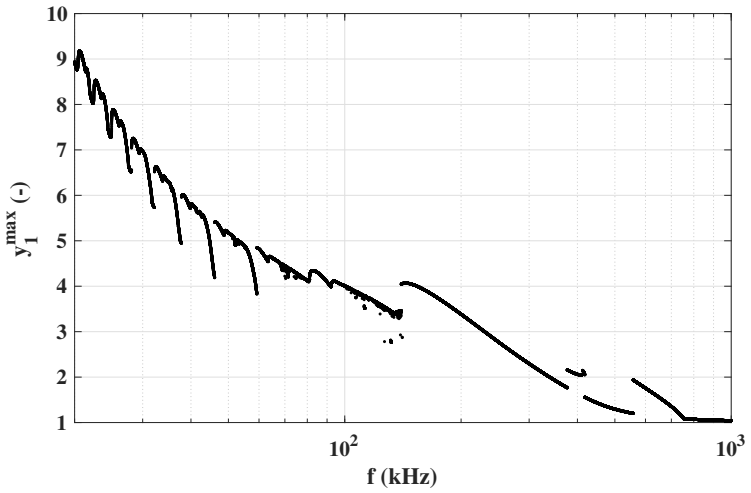

Figure 5.5: Frequency response curve of the Keller–Miksis equation, where the maximum of the dimensionless bubble radius y_1^{max} is presented as a function of the driving frequency f. The pressure amplitude was constant $P_A = 1.5\,\text{bar}$. The resolution of the frequency domain in this specific example is $N = NT = 46080$ that is also the number of the instances of the ODE system solved simultaneously.

The main challenge of this problem is that the solutions at different frequency values have time scale differences varied between orders of magnitude. This is demonstrated in Fig. 5.6, where the time series of the dimensionless bubble radius of 2 integration phases are plotted for three different frequency values. The required time steps for the frequency values $f = 20$, 100 and 500 kHz were 6820, 1851 and 320, respectively. In case of such systems, it is extremely important to be able to separate the time co-ordinate for each instance. Otherwise, the time step will always be determined by the smallest required time step in an ensemble of instances. That is, some solution will be solved with a number of time steps orders of magnitude larger than necessary. Actually, the situation is even more problematic as all the solutions will slow down in every case a single instance slows down. A GPU is working with thousands of threads (instances), if every thread (instance) slows down only once, but at different time instances, altogether thousands of slow down will happen for each thread (instances).

The program code is the simplified version of the one already discussed in Sec. 5.4. Therefore, such a description is omitted here; the interested reader is also referred to the available code of the corresponding tutorial folder in the GitHub repository.

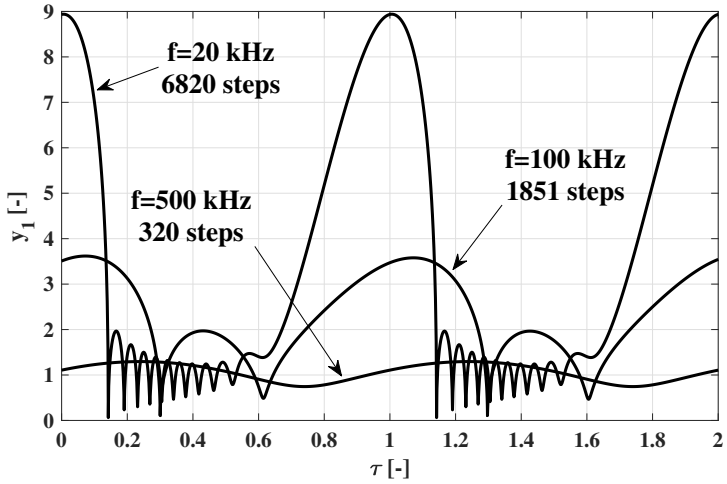

Figure 5.6: Time series of the dimensionless bubble radius at three different frequency values.

Performance comparisons

Figure 5.7 demonstrates the impact when the different time scales in an ensemble of oscillators are NOT handled properly. In the figure, the runtime of the total simulation time is depicted as a function of the resolution $N = NT$ of the parameter range. The red curve is the performance characteristics of the program package MPGOS. Its efficiency is quite clear; after saturating the GPU with enough parallel computations (threads), the curve shows a linear characteristic in the log-log diagram. The blue and black curves are related to the program packages odeint[8] written in C++, and DifferentialEquations.jl[9] written in the rapidly developing Julia language. The problem is implemented for both CPUs and GPUs. The codes of the odeint and the Julia packages are written by Lambert Plavecz[10] and Dániel Nagy[11], respectively. Their codes are also available in a GitHub repository[12]. The odeint CPU version has two variants: without the exploitation of the SIMD vector arithmetic capabilities of the CPU (using 4 doubles), and explicit vectorisation via the Vector Class Library[13] written by Agner Fog. In the case of Julia, explicit vectorisation can be done via the @avx macro. However, we did not experience any speed-up with the non-macro version. Thus, the compiler could not perform the vectorisation. The used software environment is as follows. The applications are tested under Ubuntu 20 LTS operating system. The C++ compiler is gcc 7.5.0, and the CUDA Toolkit version is 10.0. The versions of the program packages are as follows; ODEINT: v2, MPGOS: v3.1, Julia: v1.5.0 and DifferentialEquations.jl: v6.15.0. In case of Julia, many other modules are used: DiffEqGPU v1.6.0, SimpleDiffEq v1.2.1, LoopVectorization v0.8.24.

MPGOS have a solid $\times 130$ speed up compared to the CPU version of the odeint code (with VCL library). This is higher than the peak theoretical floating-point processing powers ($\times 56$) of the hardware used. In the case of Julia, the speedup is as high as $\times 300$, since we were unable to exploit the vectoristaion capabilities of the CPU. The GPU versions of both odeint and Julia performs quite poorly. The main reason is that these program packages cannot separate the time co-ordinates of the different (and independent) instances in the ensemble. That is, every independent instance can march only with the same time step; thus, the different instances of the Keller–Miksis equations slow down each other. It is hard to predict the performance differences between the other solutions as the runtimes were quite large. However, as these GPU versions perform quite poorly, it is not a priority here.

[8] www.odeint.com
[9] https://docs.juliadiffeq.org/stable/index.html
[10] plaveczlambert@gmail.com
[11] n.nagyd@gmail.com
[12] https://github.com/nnagyd/ode_solver_tests
[13] https://www.agner.org/optimize/vectorclass.pdf

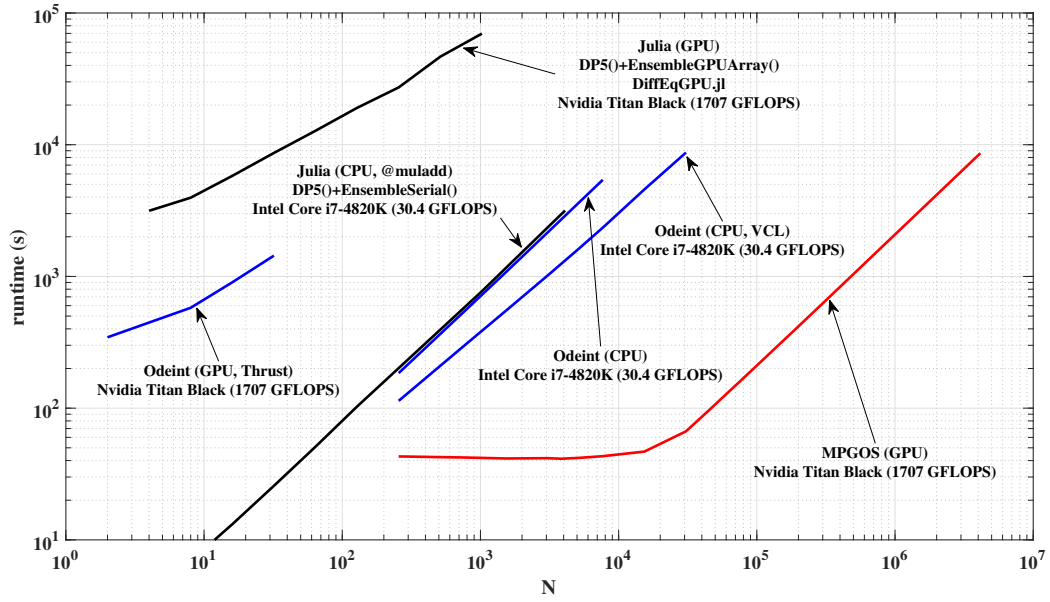

Figure 5.7: Performance comparison of different program packages for the problem of this tutorial.

5.6 Tutorial 6: impact dynamics

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

$$\dot{y}_1 = y_2, \tag{5.27}$$
$$\dot{y}_2 = -\kappa y_2 - (y_1 + \delta) + y_3, \tag{5.28}$$
$$\dot{y}_3 = \beta(q - y_1\sqrt{y_3}), \tag{5.29}$$

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 of 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 (5.27)-(5.29), 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

$$y_1^+ = y_1^- = 0, \tag{5.30}$$
$$y_2^+ = -r y_2^-, \tag{5.31}$$
$$y_3^+ = y_3^- \tag{5.32}$$

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 properly implemented pre-declared user-defined device function `PerThread_ActionAfterEventDetection` called at the impact detection, system (5.27)-(5.29) can be handled very efficiently.

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 $NT = 30720$. 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.

Remarks on the configuration of the SolverObject

In this tutorial example, the total number of problems and the number of threads NT in the SolverObject are equal (the resolution of the control parameter q). Therefore, there is only one simulation launch. The system dimension is $SD = 3$ according to the model of the pressure relief valve. The number of the *control parameters* is only $NCP = 1$ (flow rate q). However, there are 4 *shared parameters*: κ, δ, β and r. For the computations, two event functions are needed ($NE = 2$). 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}.

Remarks on the pre-declared user-defined device functions

Observe that in the function `PerThread_OdeFunction()`, 3 out of the 4 parameters are loaded from the Shared Memory. The definition of the two event functions is very straightforward. The direction of both events are -1, and for the Poincaré section, we have to stop the simulation; for details, 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. In the function `PerThread_ActionAfterEventDetection()`, the velocity y_2 is reversed by applying the impact law in Eq. (5.31). Here, the coefficient of restitution is loaded from the Shared Memory.

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. 5.8 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}. The number of the used threads and the resolution of the control parameter is $NT = 30720$.

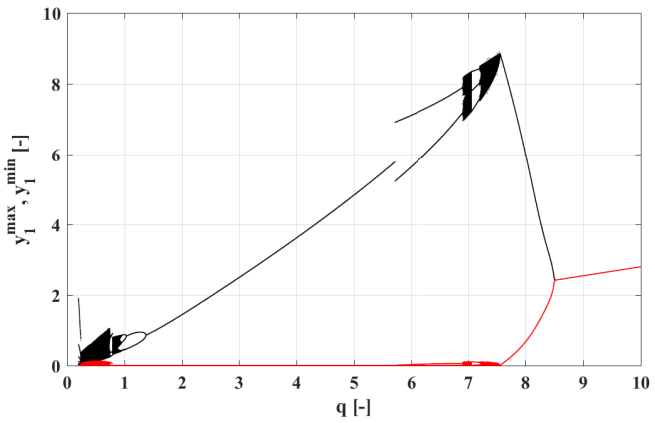

Figure 5.8: The maximum (black) and minimum (red) values of the valve position y_1 as a function of the dimensionless flow rate q for the pressure relief valve. The damping coefficient is $\kappa = 1.25$, the precompression parameter is $\delta = 10$ and the compressibility parameter is $\beta = 20$. The number of the used threads and the resolution of the control parameter is $NT = 30720$.

Performance comparison

Figure 5.9 demonstrates the impact on the performance when a special, non-smooth dynamical system has to be handled. In the figure, the runtime of the total simulation time is depicted as a function of the resolution $N = NT$ of the range of the parameter q. The red curve is the performance characteristics of the program package MPGOS. Its efficiency is quite clear; after saturating the GPU with enough parallel computations (threads), the curve shows a linear characteristic in the log-log diagram. The blue and black curves are related to the program packages odeint[14] written in C++, and DifferentialEquations.jl[15] written in the rapidly developing Julia language. The odeint CPU version exploits the SIMD vector arithmetic capabilities of the CPU (using 4 doubles) via the Vector Class Library[16] written by Agner Fog. In case of the Julia, we

[14] www.odeint.com
[15] https://docs.juliadiffeq.org/stable/index.html
[16] https://www.agner.org/optimize/vectorclass.pdf

did not manage to fuse the explicit vectorisation SIMD libraries with the package DifferentialEquations.jl due to compatibility issues. For this tutorial example, there are NO GPU versions of the codes corresponding to odeint and Julia, as the common time-stepping does not allow to implement a stop condition and the impact law properly at the different time instances. Thus, in these cases, the issue is not only performance-related, but the program packages are unsuitable for handling such a problem on GPUs. The codes of the odeint and the Julia packages are written by Lambert Plavecz[17] and Dániel Nagy[18], respectively. Their codes are also available in a GitHub repository[19].

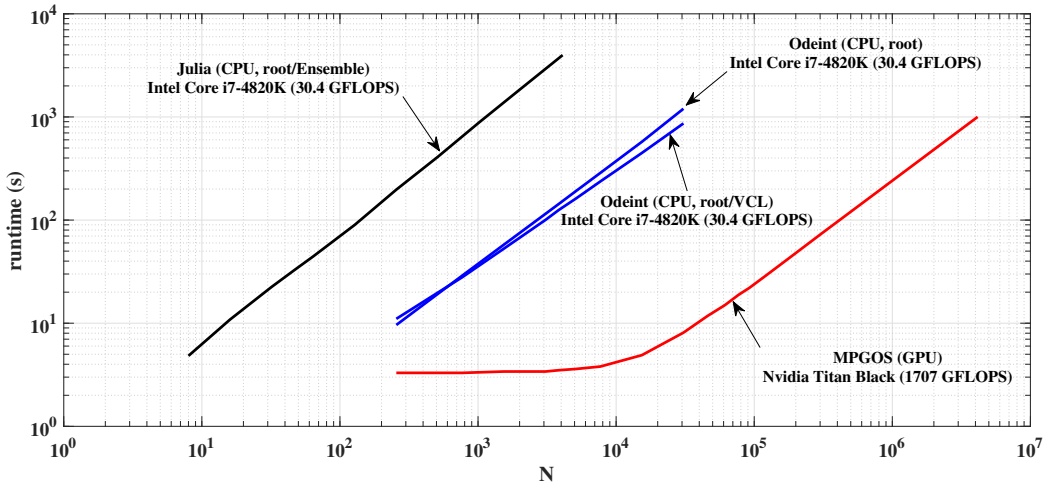

Figure 5.9: Performance comparison of different program packages for the problem of this tutorial.

MPGOS have a solid ×120 speed up compared to the CPU version of the odeint code. In the case of Julia, the speedup is as high as approximately ×3600, indicating a suboptimal usage of CPU resources.

[17] plaveczlambert@gmail.com
[18] n.nagyd@gmail.com
[19] https://github.com/nnagyd/ode_solver_tests

5.7 Tutorial 7: overlap CPU and GPU computations (double buffering)

As it is mentioned in Sec. 2.4, 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 SolverObject. All these operations are managed by the SolverObject 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. 2.4).

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 SolverObject, a device number is immediately associated during its declaration, see Sec. 2.1. Similarly, a CUDA stream is also created and associated to a SolverObject 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.

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 $NT = 23040$ using a single GPU and a single SolverObject. Here, two SolverObjects 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 seventh tutorial example. Since the system definition is identical with the reference case (first tutorial example), its discussion is omitted here.

Observe that one of the main differences from the reference case is the declaration of two SolverObjects, to which the same GPUs are associated (closest to CC 3.5). The main part of the control flow of the program is to fill up the SolverObjects one after another with 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 the last 32 integration phases with writing data onto a disc.

Compared to the reference case, the number of simulation launches are halved due to the usage of two SolverObjects. More precisely, only a single launch is enough, as each SolverObject is responsible for a half of the total number of problems (instances of the ODE). The tutorial specific function called `FillSolver Objects()` fills up the passed SolverObject with the aforementioned data. After finishing this task on the first SolverObject (`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 SolverObject (`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 SolverObjects. 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)[20].

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 just initiated, 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 SolverObjects have to be synchronised, the data have to copied back to the host side and saved to disc.

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.

[20] 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.

5.8 Tutorial 8: using multiple GPUs in a single node

The main purpose of this 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 SolverObjects each is associated to different GPUs. In the main code, the different device numbers are specified manually according to the output of the function `ListCUDADevices()` called in advance excusively.

During the computation, similar techniques have to be used already introduced in the seventh tutorial example. Therefore, if the reader is not familiar with the asynchronous behaviour of the member functions of the SolverObjects, He/She is referred back to the seventh tutorial example discussed in Sec. 5.7. The control flow is also very similar. 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 SolverObjects have to be associated to each GPU similarly as in case of the previous tutorial example. This means at least 4 SolverObjects 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.

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.

Chapter 6

MPGOS tutorial examples for module Coupled Systems Per-Block

This section serves as an additional material to highlight the flexibility and features of the module **Coupled Systems Per-Block** of the program package MPGOS. The list of the tutorial examples will be continuously extended.

During the tutorial examples, code snippets are NOT provided as all the source codes can be found in the corresponding directories. Thus only the detailed description of the problem, the main aims and task and the results are discussed. Each tutorial example have a main `.cu` file and a system definition file with extension `.cuh`.

6.1 Tutorial 1: Duffing equations connected in a ring topology

Throughout the first tutorial example, the features and capabilities of the program package are introduced by using the well-known Duffing oscillator coupled in a ring topology. The form of a single oscillator (a single Unit in the system) is

$$\dot{x}_i = y_i + \frac{\hat{\sigma}}{2R} \sum_{j=i-R}^{i+R} (x_j - x_i), \tag{6.1}$$

$$\dot{y}_i = -\gamma y_i + x_i - x_i^3 + A\cos(\omega t) + \frac{\hat{\sigma}}{2R} \sum_{j=i-R}^{i+R} (y_j - y_i), \tag{6.2}$$

As the dimension of the state space for a single oscillator (unit) is $UD = 2$ (unit dimension), we used the notation x and y for the components of a system to avoid the overuse of multi-indexing, and thus keep the discussion more clear. In this sense, the pair of (x_i, y_i) defines a point on the state space of a single oscillator, where $i = 1 \dots N$. The variable $N = UPS = 14$ (unit per system) is the number of the coupled oscillators. The parameter $\hat{\sigma} = 0.004$ is the coupling strength, and $R = CWB = 1$ is the coupling radius of the neighbouring units. The units are coupled in a ring topology; thus, the coupling matrix is circular (discussed below). The angular frequency of the driving is $\omega = 0.5\,\mathrm{rad/s}$, the damping of the system is $\gamma = 0.24$ and the driving amplitude is $A = 13.633$. Due to the periodic forcing, the state space is also periodic with period $T = 2\pi/\omega$ that is used to define the global Poincaré section. The initial conditions for each oscillators (units) are $(x_i^0, y_i^0) = (-0.5, -0.5)$ except the 12^{th} unit (x_{12}^0, y_{12}^0). The initial condition x_{12}^0 is varied between -8 and 8 with a resolution of 5; the initial condition y_{12}^0 is varied between -20 and 40 with a resolution of again 5. Both are distributed uniformly. Therefore, altogether $NS = 5 \times 5 = 25$ instance of the coupled ODE system are solved. They have the same parameter set; however, initialised from a different set of initial conditions. The notations UD, UPS, CWB and NS are template parameters, see Sec. 2.1.2 for details.

The coupling terms in Eqs. (6.1)-(6.2) reflect an important physical property; namely, in case of complete synchronisation of the oscillators, the values of the coupling terms vanish. However, the expression inside the sum $(x_j - x_i)$ is problematic from computational point of view as these terms do NOT have the general form MPGOS can handle, see Eq. (1.2) in Sec.1.4.2. Thus, the equation system needs to be rewritten as

$$\dot{x}_i = y_i - \hat{\sigma} x_i + \frac{\hat{\sigma}}{2R} \sum_{\substack{j=i-R \\ j\neq i}}^{i+R} x_j, \tag{6.3}$$

$$\dot{y}_i = -\gamma y_i + x_i - x_i^3 + A\cos(\omega t) - \hat{\sigma} y_i + \frac{\hat{\sigma}}{2R} \sum_{\substack{j=i-R \\ j\neq i}}^{i+R} y_j. \tag{6.4}$$

Since the Duffing oscillators are coupled in a ring topology, and the coupling radius is $R = 1$, the coupling values $K_i^{(k)}$ can be written in matrix forms as

$$\begin{pmatrix} K_1^{(0)} \\ K_2^{(0)} \\ \vdots \\ K_N^{(0)} \end{pmatrix} = \frac{\sigma}{2R} \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 1 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} \tag{6.5}$$

and

$$\begin{pmatrix} K_1^{(1)} \\ K_2^{(1)} \\ \vdots \\ K_N^{(1)} \end{pmatrix} = \frac{\sigma}{2R} \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 1 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \tag{6.6}$$

where $k = 0, 1$ is the serial number of the couplings. Comparing the matrix form of the couplins in Eq. (6.5) and in Eq. (6.6) with the general form of a coupling in Eq. (1.2), one can see the following. The coupling terms are $H_i^{(0)} = x_i$ and $H_i^{(1)} = y_i$; the coupling factors are identially one $G_i^{(k)} = 1$; The "effective" coupling strength is $\sigma^{(k)} = 2\hat{\sigma}/(2R)$; and finally, the two coupling matrices $A_{i,j}^{(k)}$ are identical and they are circular band matrices with a bandwidth of $R = 1$. Although the coupling matrices are identical, we need two specify both coupling matrices. Note that the serial number of the couplings are started from zero following a C++ indexing convention.

Due to the ring topology, the coupling matrices are circular; that is, $A_{N,1} = 1$ and $A_{1,N} = 1$. Therefore, such matrices can be stored very efficiently, by specifying the corresponding template parameter to $CCI = 1$, for details see Sec. 4.7. The coupling radius (bandwidth of the coupling matrices) is $R = CBW = 1$. Therefore, only 3 elements have to be stored for a single matrix (the values at each diagonal). The last template parameter related to the coupling is the number of couplings $NC = 2$.

The parameters γ, A and ω are stored in *unit parameters* that are different from instance to instance and unit to unit. Although they are constants and the same for all instances of the system, we wanted to keep the code general to be able to easily modify the problem to calculate instances having different parameter sets. For this purpose, 3 *unit parameters* are allocated ($NUP = 3$). The coupling factor σ is stored in a *global parameter* shared by all the instances and all the units ($NGP = 1$). The effective coupling factor $\hat{\sigma}/(2R)$—the coupling factor from the computational point of view—and the coupling matrix has to be specified during the fill-up of the SolverObject (similarly to the initial conditions, time domain, *parameters* or *accessories*).

The objectives

The objective of this tutorial example is very simple. Start the simulation from $t = 0$, and stop the simulation when a unit reaches a local maximum of its first component x_i. For this, an event function has to be defined as $F_{E1} = y_i$. Keep in mind that event can be specified only in a unit-level; that is, only in the (x_i, y_i, t) subspace. There is no possibility to define event on a system level that needs intercommunication between threads. To be able to stop the integration at a speecific number of detected event (it is 1 here), an *integer unit accessory* is allocated to each unit ($NiUA = 1$).

Secondary objectives are the storage of the minimum and the maximum values of the time step, and the total number of successful steps. This needs 2 *system accessories* ($NSA = 2$) and 1 *integer system accessory* ($NiSA = 1$). In addition, the component x_i of the end state of the unit that triggers the event is also stored in a *unit accessory*; the accessory of the other units remain the initial condition x_i^0. For convenience, the initial conditions x_i^0 and y_i^0 are also stored in *unit accessories* (it is done during the filling of the SolverObject). Therefore, altogether 3 *unit accessories* are allocated ($NUA = 3$).

Finally, the dense output of all the simulations are also stored and printed in different files. The maximum number of stored time steps are $NDO = 500$ that is more than enough for this problem.

Further remarks on the configuration of the SolverObject

Regardless of how the coupling matrix is stored internally in MPGOS, it must be specified by ordinary matrix indexing via the interface of the module. That is why in the function `FillCouplingMatrix()`, the access of the elements in the diagonals are recalculated to row and column indices.

In the last part of the `main()` function, the results are written into the file `Results.txt`. It contains the system number, the number of time steps and the final time instance, the minimum and maximum time steps, the initial conditions of the 12^{th} unit and the final state of the whole coupled systems.

As a last remark, observe that how the vaule of $\hat{\sigma}/(2R)$ is specified as a coupling strength (instead of simply $\hat{\sigma}$) in the function `FillSolverObject()`.

Remarks on the pre-declared user-defined device functions

Inspecting the `CoupledSystems_PerBlock_OdeFunction()` pre-declared user-defined device function, it is clear that the first two lines calculate the uncoupled part of a unit. The next two lines calculate the coupling term ($CPT[0] = X[0]$, $H_i^{(0)} = x_i$) and coupling factor ($CPF[0] = 1$, $G_i^{(0)} = 1$) of the first coupling. Finally, the last two lines calculate the coupling term ($CPT[1] = X[1]$, $H_i^{(1)} = y_i$) and coupling factor ($CPF[1] = 1$, $G_i^{(1)} = 1$) of the first coupling. Observe that only the uncoupled part of the system, the coupling terms and the coupling factors have to be implemented by the user, the rest is managed by the solver.

The event function is very simple, according to the function $F_{E1} = y_i$. The function `CoupledSystems_PerBlock_ActionAfterEventDetection()` manages the stop condition, when a thread (unit) triggers the event detection the second time, the simulation of the corresponding whole instance of the coupled system is stopped (regardless of the number of the detected events of the other units). The end state of the first component of the unit that triggered the termination is stored in an *accessory*.

The function `CoupledSystems_PerBlock_ActionAfterSuccessfulTimeStep()` simply updates the maximum and minimum time steps and accumulates the number of time steps. Each variable is stored in an *accessory*. During the initialisation of the integration phases, the initial values of the *accessories* have to be properly specified via the function `CoupledSystems_PerBlock_Initialization()`. Moreover, the reset of the `ActualTime` and the starting index of the `DenseOutput` has to be done here as well.

Results

As this tutorial example serves only to demonstrate the proper usage of most of the possibilities of this MPGOS module, no performance measurements are done. Moreover, this is an artificially constructed problem; thus, no particular simulation results are presented either in a visualised form. For such examples, the reader is referred to the subsequent tutorial examples.

6.2 Tutorial 2: Bubble ensembles

The second tutorial example calculates the dynamics of an ensemble of spherical bubbles. A single bubble has been examined in the tutorial example in Sec. 5.4. The mathematical model is very similar. The dynamics of the system of a single bubble (a single unit) is governed by the same Keller–Miksis equation extended with a coupling part that summarises the influence caused by the other bubbles. The interaction of the bubbles take place via their emitted pressure wave:

$$p_i(t) = \frac{\rho}{r}\left(R_i^2\ddot{R}_i + 2R_i\dot{R}_i^2\right), \tag{6.7}$$

where the subscript i is the serial number of a single bubble (unit), ρ is the density of the liquid, r is the distance from the bubble and $R(t)$ is the time-dependent bubble radius. The dot stands for the time derivative. Accordingly, the governing equation of the dynamics of a single bubble, that is a second-order ordinary differential equation, reads as

$$\left(1-\frac{\dot{R}_i}{c_L}\right)R_i\ddot{R}_i + \left(1-\frac{\dot{R}_i}{3c_L}\right)\frac{3}{2}\dot{R}_i^2 = \left(1+\frac{\dot{R}_i}{c_L}+\frac{R_i}{c_L}\frac{d}{dt}\right)\frac{(p_{L,i}-p_\infty(t))}{\rho_L} - \sum_{j=1}^{N}\frac{1}{d_{i,j}}\left(R_j^2\ddot{R}_j + 2R_j\dot{R}_j^2\right), \tag{6.8}$$

where the sum is the influence (pressure fluctuations) originated from the other bubbles. The distance between the i^{th} and j^{th} bubbles is $d_{i,j}$. The sound speed and the density of the liquid domain are $c_L = 1497.3\,\mathrm{m/s}$ and $\rho_L = 997.1\,\mathrm{kg/m^3}$, respectively. The pressure far away from the bubble, $p_\infty(t)$, is composed by static and periodic components

$$p_\infty(t) = P_\infty + P_{A1}\sin(\omega_1 t) + P_{A2}\sin(\omega_2 t + \theta), \tag{6.9}$$

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. For simplicity, all the bubbles in the ensemble "feel" the same ambient pressure and pressure amplitudes. Such a dual-frequency driven bubble cluster is important in the field of acoustic cavitation and sonochemistry.

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

$$p_{G,i} + p_V = p_{L,i} + \frac{2\sigma}{R_i} + 4\mu_L\frac{\dot{R}_i}{R_i}, \tag{6.10}$$

where the total pressure inside the bubble is the sum of the partial pressures of the non-condensable gas, $p_{G,i}$, 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

$$p_{G,i} = \left(P_\infty - p_V + \frac{2\sigma}{R_{E,i}}\right)\left(\frac{R_{E,i}}{R_i}\right)^{3\gamma}, \tag{6.11}$$

where the polytropic exponent $\gamma = 1.4$ (adiabatic behaviour) and the equilibrium radius of the i^{th} bubble is $R_{E,i}$. The values $R_{E,i}$ are constant during a simulation, but each bubble can have its own equilibrium size (its own size in the absence of external driving).

The detailed description and the physical interpretation of Eqs. (6.8)-(6.11) is omitted here. Although the external driving is written in a dual-frequency form due to compatibility reasons with the tutorial example in Sec. 5.4, only a single frequency is used here by setting $P_{A2} = 0$. Therefore, the external driving is always periodic. The main code in the present tutorial example generates a randomised structure of $N = 128$ bubbles. The spatial distribution is generated by a Gaussian distribution with 10 mm spatial variance. From the positions of the bubbles, the distance matrix $d_{i,j}$ can be computed. The bubble sizes $R_{E,i}$ are sampled from a uniform distribution between sizes of 3 to 8 μm.

For numerical purposes, the coupled systems are rewritten into a dimensionless form defined as

$$\dot{y}_{1,i} = y_{2,i} = f_1(y_{2,i}), \tag{6.12}$$

$$\dot{y}_{2,i} = \underbrace{\frac{N_{\mathrm{KM},i}}{D_{\mathrm{KM},i}}}_{f_2(y_{1,i},y_{2,i},\tau)} \underbrace{-\frac{1}{D_{\mathrm{KM},i}} \sum_{j=1}^{N} \frac{R_{E,j}^3}{R_{E,i}^2\, d_{i,j}} \left(y_{1,j}^2\, \dot{y}_{2,j} + 2 y_{1,j}\, y_{2,j}^2\right)}_{K_i}, \tag{6.13}$$

where the numerator, $N_{\mathrm{KM},i}$, and the denominator, $D_{\mathrm{KM},i}$, are

$$\begin{aligned} N_{\mathrm{KM},i} = (C_{0,i} + C_{1,i} y_{2,i}) \left(\frac{1}{y_{1,i}}\right)^{C_{10,i}} &- C_{2,i}\,(1 + C_{9,i} y_{2,i}) \\ &- C_{3,i} \frac{1}{y_{1,i}} - C_{4,i} \frac{y_{2,i}}{y_{1,i}} - \left(1 - C_{9,i} \frac{y_{2,i}}{3}\right) \frac{3}{2} y_{2,i}^2 \\ &- \left(C_{5,i} \sin(2\pi\tau) + C_{6,i} \sin(2\pi C_{11,i}\tau + C_{12,i})\right)(1 + C_{9,i} y_{2,i}) \\ &- y_{1,i} \left(C_{7,i} \cos(2\pi\tau) + C_{8,i} \cos(2\pi C_{11,i}\tau + C_{12,i})\right), \end{aligned} \tag{6.14}$$

and

$$D_{\mathrm{KM},i} = y_{1,i} - C_{9,i} y_{1,i} y_{2,i} + C_{4,i} C_{9,i}. \tag{6.15}$$

The coefficients are summarised as follows:

$$C_{0,i} = \frac{1}{\rho_L} \left(P_\infty - p_V + \frac{2\sigma}{R_{E,i}}\right) \left(\frac{2\pi}{R_{E,i}\omega_1}\right)^2, \tag{6.16}$$

$$C_{1,i} = \frac{1 - 3\gamma}{\rho_L c_L} \left(P_\infty - p_V + \frac{2\sigma}{R_{E,i}}\right) \frac{2\pi}{R_{E,i}\omega_1}, \tag{6.17}$$

$$C_{2,i} = \frac{P_\infty - p_V}{\rho_L} \left(\frac{2\pi}{R_{E,i}\omega_1}\right)^2, \tag{6.18}$$

$$C_{3,i} = \frac{2\sigma}{\rho_L R_{E,i}} \left(\frac{2\pi}{R_{E,i}\omega_1}\right)^2, \tag{6.19}$$

$$C_{4,i} = \frac{4\mu_L}{\rho_L R_{E,i}^2} \frac{2\pi}{\omega_1}, \tag{6.20}$$

$$C_{5,i} = \frac{P_{A1}}{\rho_L} \left(\frac{2\pi}{R_{E,i}\omega_1}\right)^2, \tag{6.21}$$

$$C_{6,i} = \frac{P_{A2}}{\rho_L} \left(\frac{2\pi}{R_{E,i}\omega_1}\right)^2, \tag{6.22}$$

$$C_{7,i} = R_{E,i} \frac{\omega_1 P_{A1}}{\rho_L c_L} \left(\frac{2\pi}{R_{E,i}\omega_1}\right)^2, \tag{6.23}$$

$$C_{8,i} = R_{E,i} \frac{\omega_1 P_{A2}}{\rho_L c_L} \left(\frac{2\pi}{R_{E,i}\omega_1}\right)^2, \tag{6.24}$$

$$C_{9,i} = \frac{R_{E,i}\omega_1}{2\pi c_L}, \tag{6.25}$$

$$C_{10,i} = 3\gamma, \tag{6.26}$$

$$C_{11,i} = \frac{\omega_2}{\omega_1}, \tag{6.27}$$

$$C_{12,i} = \theta. \tag{6.28}$$

Observe that from the implementation point of view, the number of the parameters of each unit (*unit parameters*) of the system is 13 ($C_{0,i}$ to $C_{12,i}$). Therefore, the physical parameters (in this tutorial only P_{A1},

see the discussion below) and the appearing systems coefficients as parameters must be clearly separated in the code. Although the usage of the coefficients $C_{0,i-12,i}$—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 SolverObject. Observe also that some of the coefficients (e.g., $C_{10,i}$) can be shared amongst all the units and all the coupled systems; however, for simplicity, they are also treated as *unit parameters*. The function f_1 and f_2 denotes the uncoupled part(s) of the equations (the notation of the dependence on the parameters are omitted for brevity).

In the coupling part, in the right-hand side of Eq. (6.13), the variable $\dot{y}_{2,j}$ also appears. This makes the coupling implicit that cannot be handled in the present version of the module **Coupled Systems Per-Block**. To overcome this difficulty and make the coupling explicit, this variable is approximated by the uncoupled part of the system:

$$\dot{y}_{2,j} \approx \frac{N_{\mathrm{KM},j}}{D_{\mathrm{KM},j}}, \tag{6.29}$$

which can be computed for all units (individual bubbles) in the ODE function. Due to this simplification, the numerics can be unstable. This happens when the bubbles are close to each other. Therefore, during the generation of the distances and the coupling matrix, the code throws a warning if a distance is below $100\,\mu$m. Now the coupling terms K_i in Eq. (6.13) can be written in a matrix form as

$$K_i = G_i \odot (A_{i,j}\, H_j)\,, \tag{6.30}$$

where $\odot$ is the element-wise (Hadamard, Schur) product of two vectors. The coupling matrix $A_{i,j}$ has the following structure

$$\begin{pmatrix} 0 & \dfrac{R_{E,2}^3}{R_{E,1}^2\, d_{1,2}} & \cdots & \dfrac{R_{E,N}^3}{R_{E,1}^2\, d_{1,N}} \\ \dfrac{R_{E,1}^3}{R_{E,2}^2\, d_{2,1}} & 0 & \cdots & \dfrac{R_{E,N}^3}{R_{E,2}^2\, d_{2,N}} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{R_{E,1}^3}{R_{E,N}^2\, d_{N,1}} & \dfrac{R_{E,2}^3}{R_{E,N}^2\, d_{N,2}} & \cdots & 0 \end{pmatrix}. \tag{6.31}$$

Observe the zero elements in the diagonal (a bubble is NOT coupled to itself). This matrix is asymmetric as the bubble sizes $R_{E,i}$ are generated randomly. The coupling terms H_i are computed as

$$H_i = y_{1,i}^2\, \frac{N_{\mathrm{KM},i}}{D_{\mathrm{KM},i}} + 2y_{1,i}\, y_{2,i}^2 = h(y_{1,i}, y_{2,i}), \tag{6.32}$$

which depends only on the variables of a single unit; therefore, their values can be computed together with the uncoupled part (functions f_1 and f_2) of the system before performing the computation of K_i. The function h has no serial number here as there is only a single coupling. The coupling factors G_i are simply the reciprocal of the denominators:

$$G_i = \frac{1}{D_{\mathrm{KM},i}} = g(y_{1,i}, y_{2,i}). \tag{6.33}$$

It also depends only on the variables of a single Unit, and its values can be computed together with the functions f_1, f_2 and h. The function g has, again, no serial number as there is only a single coupling. **Note that how the structure of the coupling presented here agrees with the general description in Sec 1.4.2**.

The objectives

The main aim of this tutorial example is to create a bifurcation diagram; that is, present the first component of the Poincaré section $\Pi(y_{1,i})$ as a function of the pressure amplitude P_{A1} as a control parameter. Since

the driving (and thus the state space) is periodic with period $T = 2\pi/\omega_1$, the global Poincaré section can be obtained by sampling the trajectories (of all units) by T. In the dimensionless time coordinate, the period is $\tau_p = 1$. Again, the number of bubbles in a cluster is $N = 128$. The amplitude of the driving P_{A1} is varied between 0 bar and 5 bar with a resolution specified by the template parameter NS. In this specific setup $NS = 600$. The frequency of the driving is kept constant at $f = \omega_1/(2\pi) = 1000$ kHz. However, the number of frequencies used can easily be adjusted to perform 2D parameter scans. The first 512 integration phases are regarded as transients and discarded, and the points of the Poincaré section of the next 32 integration phases are recorded. One integration phase means the integration of the system between $\tau = 0$ and $\tau = 1$. The control flow of the main code and the system definition file is simple as no event detection, and the storage of special points are necessary (only the storage of the endpoints at the last 32 integration phases). Only a single initial condition is applied at each control parameter values; however, $y_{1,i}$ is randomised between 0.75 and 2.25. The initial value of $y_{2,i}$ is always zero.

Remarks on the configuration of the SolverObject

The values of the template parameters are as follows. The total number of the integrated systems NS is the multiplication of the resolution of the pressure amplitudes and frequencies (unity here). The unit-dimension is $UD = 2$ according to the second-order Keller–Miksis equation. The number of the units-per-system and the number of the threads in a block are $UPS = TPB = 128$. In addition, as the system-per-block template parameter is $SPB = 1$, the solver will use the single-system-single-block-launch parallelisation approach. The number of couplings is $NC = 1$. All the 13 equation coefficients in Eq. (6.16) are stored in *unit parameters*. Moreover, the reference time ($2\pi/f$) and length ($R_{E,i}$), and all the six parameters of the dual-frequency driving is also stored in a *unit parameter* for convenience, although most of then are not used during the tutorial. Altogether, $NUP = 21$ *unit parameters* are allocated.

The rest of the code should be self-explanatory. The function definitions in the main code in the source file `BubbleEnsemble.cu` are NOT part of the program package MPGOS; they are special auxiliary functions of this tutorial to make the code more readable.

Remarks on the pre-declared user-defined device functions

The computation of the uncoupled part of the units are exactly the same as in the case of the single bubble dynamical examle presented in Sec. 5.4. Only the first 13 *unit parameters* (`uPAR`) is used, and the frequently used $1/y_{1,i}$ is precomputed into the variable `rx1`. Note that the uncoupled functions f_1 and f_2 are calculated into `F[0]` and `F[1]`, respectively. The coupling term function h and the coupling factor function g are calculated into `CPT[0]` and `CPF[0]`, respectively. Keep in mind that the indexing in C++ starts from zero, and that the number of the coupling is only one ($NC = 1$). Note also that the functions f_1, f_2, h and g are computed in the same pre-declared user-defined device function: `CoupledSystems_PerBlock_OdeFunction()`. This means that these computations are done in parallel independently from each other. After that, the program package MPGOS synchronises the threads and augment the right-hand side of the system with the coupling part. Fundamentylly, it is a matrix-vector multiplication followed by an elementwise vector-vector multiplication. These operations can be easily paralellisable.

Results

The resulted bifurcation diagram of the computations is presented in Fig. 6.1, in which the points of the first component of the Poincaré section $\Pi(y_{1,i})$ is plotted as a function of the pressure amplitude P_{A1}. Out of the 128 bubbles in a cluster, only the results of the first 3 bubbles are shown ($i = 1, 2, 3$), see the three different colour codings. The resolution of the control parameters (that is also the number of the instances of the coupled bubble system) is 600. The total simulation time was approximately 20 min on an Nvidia GeForce GTX Titan Black GPU (1707 GFLOPS peak double-precision performance). The scattered like bifurcation

branches in the periodic regimes are due to the randomised initial conditions and the presence of extremely large number of co-existing attractors (attractor crowding).

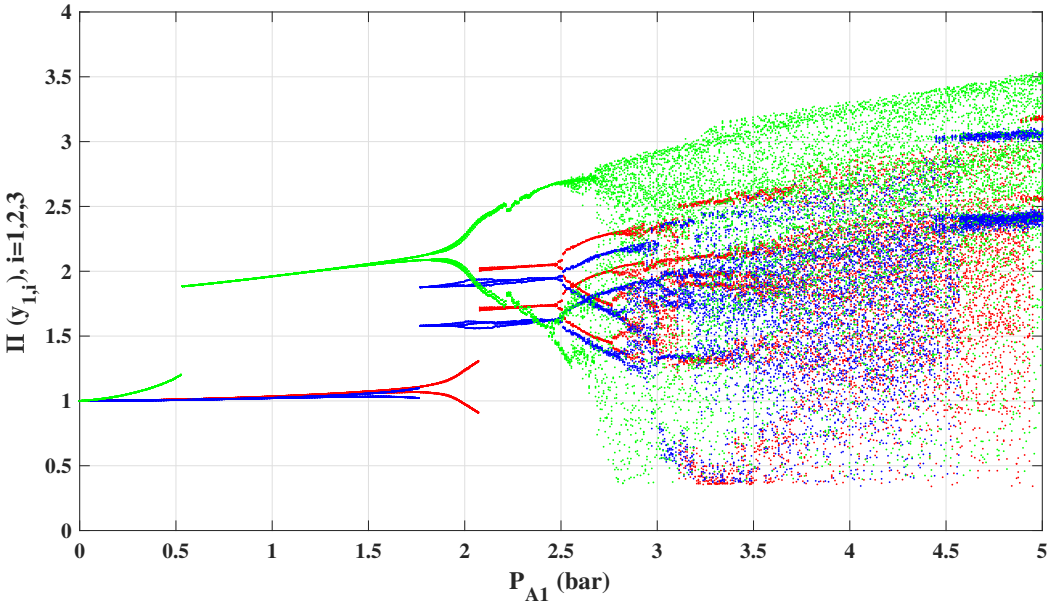

Figure 6.1: Points of the Poincaré section as a function of the pressure amplitude P_{A1} of the first three bubbles in a bubble cluster composed by 128 bubbles. The different colour codes correspond to different bubbles. The resolution of the control parameter is $NS = 600$. The total simulation time was approximately 20 min.