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

Massively Parallel GPU-ODE Solver (MPGOS) PerThread v1.1

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

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

April 9, 2019

Contents

1 Installation and file hierarchy

Program code MPGOS is written in C++ and CUDA C software environments. In order to use MPGOS, one needs only to include an appropriate source file 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 inside the **PerThread/** folder. The term *PerThread* indicates the parallelisation strategy discussed in more details throughout the manual. The list of the source files:

- **Reference.cu**
- **MassivelyParallel_GPU-ODE_Solver.cu**
- **MassivelyParallel_GPU-ODE_Solver.cuh**
- **PerThread_RungeKutta.cuh**
- **PerThread_SystemDefinition.cuh**
- **makefile**
- **Profile.sh**

The file **Reference.cu** contains the main function of the first tutorial example that is used in the subsequent sections to introduce the basics and the main features of the program package. It solves a large amount of independent Duffing oscillators in parallel. This file is not strictly a part of the package rather than an example of how to set up a new problem.

The file **MassivelyParallel_GPU-ODE_Solver.cu** defines/implements all objects, structures and other built-in types of the package. Moreover, it organises the main control flow of the integration phases that is completely hidden from the user. The user only has to fill the built-in object with proper data and then simply call the solver function. The corresponding header file for the declarations is the file **MassivelyParallel_GPU-ODE_Solver.cuh**. This is the only file the user has to include into his/her own main module (his/her own version of the **Reference.cu** file).

The available solver algorithms are implemented in the file **PerThread_RungeKutta.cuh**. Observe that it is a header file, and it is already included into the file **MassivelyParallel_GPU-ODE_Solver.cu**. Therefore, the user has nothing to do with this file, only ensure that the above two files are in the same directory, or ensure that the paths to the files are properly set.

Since function pointers cannot be passed as arguments to a GPU kernel function, the ODE functions (system) and all the other related functions have to be given through pre-declared device (GPU) functions. These functions are collected in the **PerThread_SystemDefinition.cuh** file and **this is the only file the user has to edit** manually during the code development. Naturally, the **.cu** containing the main function must also be edited, but such a file is not the part of the program package. Here, the file **Reference.cu** is included only for demonstration purposes and for better understanding. Observe that this file is again a header file which is included already into the file **MassivelyParallel_GPU-ODE_Solver.cu**. That is, the user has to ensure again that this file is in the same folder as all the others, or again ensure the proper setup of the paths. My personal suggestion to those who are new to C++ programming is to *use all the necessary source files in a single directory dedicated to a specific problem.*

Observe that many parts of the code are actually implemented in header files. This programming style goes against the practice; however, MPGOS is designed to solve ODE systems fast, and to keep the code efficient, it is extremely important to compile the solver algorithm file and the system file together in a single module. In this case, the Nvidia compiler **nvcc** has much more flexibility during the optimization (e.g. function inlining). This is the only way we can keep code efficiency and separate the ODE function(s) from the other source files for better readability.

MPGOS is developed under Linux environment. The file **makefile** is responsible to compile the code corresponding to the reference case by simply type *make* into the command line (as long as the required compilers

are installed). The command *make clean* clears the directory from the unnecessary intermediate files generated during the compilation process. The above **makefile** is a very simple example using only the basics of the makefile environment, the interested user is referred to https://www.gnu.org/software/make/manual/html_node/Introduction.html for more details. Observe that only the files **Reference.cu** and **MassivelyParallel_GPU-ODE_Solver.cu** need to be compiled and then linked together. The rest of the source files are header files already included into **MassivelyParallel_GPU-ODE_Solver.cu**. The generated executable file is called **Duffing.exe**.

The last file **Profile.sh** is a Linux shell script performing exhaustive profiling on the compiled code for the reference case **Duffing.exe**. For other projects, the user has to replace the terms *Duffing.exe* throughout the entire script with its own executable. The output of this file is understandable for advanced users who are a bit familiar with GPU programming and GPU architectures.

1.1 Prerequisite for usage

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

Although some general advice will be given in Sec. 13 to maximise performance, the user is referred to the official Nvidia documentation Programming Guide and Best Practice Guide in https://docs.nvidia.com/cuda/.

1.2 Installation

The program code MPGOS needs no explicit installation. As stated above, in order to use MPGOS, one needs only to include the file **MassivelyParallel_GPU-ODE_Solver.cuh** and properly setup the system file **PerThread_SystemDefinition.cuh**. Nevertheless, the proper installation of a C++ compiler (**e.g g++**) and an Nvidia compiler (**nvcc**) are mandatory. Moreover, the proper setup of the **PATH** environment variables are also advisable (e.g. in a *.bashrc* or *.profile* file in Linux). The following listing shows an example for such a set up in a *.bashrc* file

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

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

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

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

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

Here it is assumed that the user has an access to a Linux machine (either using directly or logged in remotely e.g. from Windows). It is also assumed that **g++** and **nvcc** is already installed, and the **PATH** environment

variables have been properly extended, see Sec. 1.2. In this section, a BASH command shell is used. The reference case served to guide the user through the basics of the program package can be downloaded and run with only a handful of commands. First, the package has to be downloaded from the GitHub repository directly from the site (as a zipped file)

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

or use the following command if git is already installed:

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

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

```
cd Massively-Parallel-GPU-ODE-Solver/PerThread/
```

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

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

```
fhegedus@domino ~$ git clone https://github.com/FerencHegedus/Massively-Parallel-GPU-ODE-Solver
Cloning into 'Massively-Parallel-GPU-ODE-Solver'...
remote: Enumerating objects: 80, done.
remote: Counting objects: 100% (80/80), done.
remote: Compressing objects: 100% (60/60), done.
remote: Total 80 (delta 33), reused 60 (delta 16), pack-reused 0
Unpacking objects: 100% (80/80), done.
fhegedus@domino ~$ cd Massively-Parallel-GPU-ODE-Solver/PerThread/
fhegedus@domino PerThread$ make
nvcc -c Reference.cu                                -O3 -std=c++11
nvcc -c MassivelyParallel_GPU-ODE_Solver.cu     -O3 --ptxas-options=-v --gpu-architecture=sm_35 -lineinfo -maxrregcount=64
ptxas info    : 542 bytes gmem, 360 bytes cmem[3]
ptxas info    : Compiling entry function '_Z17PerThread_RK4_EHO27IntegratorInternalVariables' for 'sm_35'
ptxas info    : Function properties for _Z17PerThread_RK4_EHO27IntegratorInternalVariables
    56 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 64 registers, 464 bytes cmem[0], 152 bytes cmem[2]
ptxas info    : Function properties for __internal_trig_reduction_slowpathd
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Compiling entry function '_Z13PerThread_RK427IntegratorInternalVariables' for 'sm_35'
ptxas info    : Function properties for _Z13PerThread_RK427IntegratorInternalVariables
    56 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 54 registers, 16 bytes smem, 464 bytes cmem[0], 176 bytes cmem[2]
ptxas info    : Function properties for __internal_trig_reduction_slowpathd
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Compiling entry function '_Z20PerThread_RKCK45_EHO27IntegratorInternalVariables' for 'sm_35'
ptxas info    : Function properties for _Z20PerThread_RKCK45_EHO27IntegratorInternalVariables
    88 bytes stack frame, 14 bytes spill stores, 12 bytes spill loads
ptxas info    : Used 64 registers, 32 bytes smem, 464 bytes cmem[0], 404 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
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Compiling entry function '_Z16PerThread_RKCK4527IntegratorInternalVariables' for 'sm_35'
ptxas info    : Function properties for _Z16PerThread_RKCK4527IntegratorInternalVariables
    96 bytes stack frame, 20 bytes spill stores, 20 bytes spill loads
ptxas info    : Used 64 registers, 48 bytes smem, 464 bytes cmem[0], 404 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
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
nvcc -o Reference.exe   Reference.o MassivelyParallel_GPU-ODE_Solver.o
fhegedus@domino PerThread$ make clean
rm -f Reference.o
rm -f MassivelyParallel_GPU-ODE_Solver.o
fhegedus@domino PerThread$ ./Reference.exe
```

Figure 1: Commands in a Linux bash shell for a quick start.

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. 2. After specifying the Host (either via its name or its IP address) and the used port, an SSH command window shall be available where the user can login via his/her username and password.

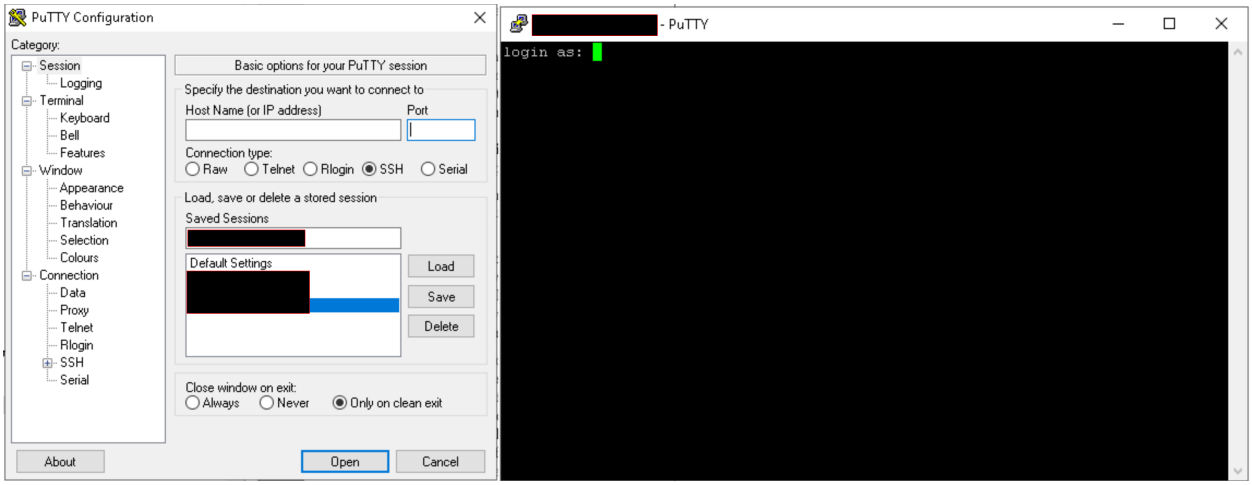

Figure 2: Login with the SSH client PuTTY.

The second tool called winSCP offers an easy way for file transfer and it automatically uploads back the edited source files to the remote machine. The login is very similar as in case of PuTTY, see Fig. 3.

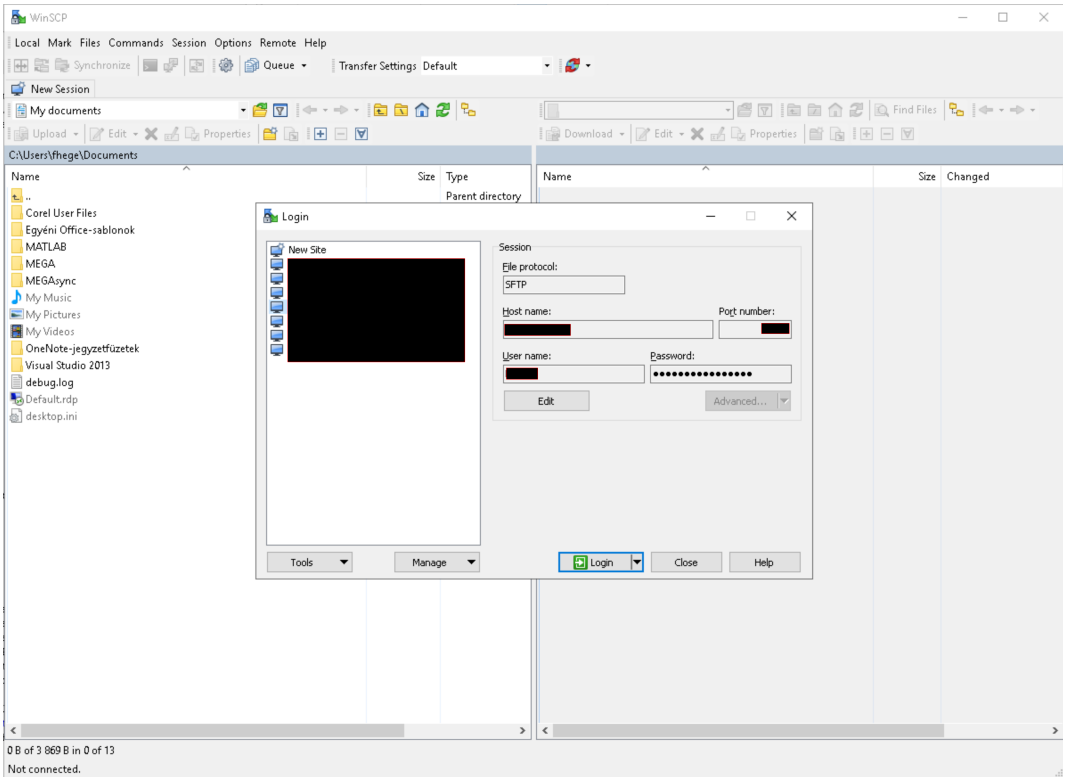

Figure 3: Login with the tool WinSCP for file exchange and editing files.

In the left panel, there is the directory and file structure of the machine from which the user has been logged on (guest machine); in the right panel, there is the directory and file structure of the Linux machine to where the user has been logged on (host machine). Under the menu *Option/Preferences...*, in the *Editors* panel, the user can specify and add editors (I prefer **Notepad++**), see Fig. 4. 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 the in the PuTTY shell the new code can immediately be compiled and run.

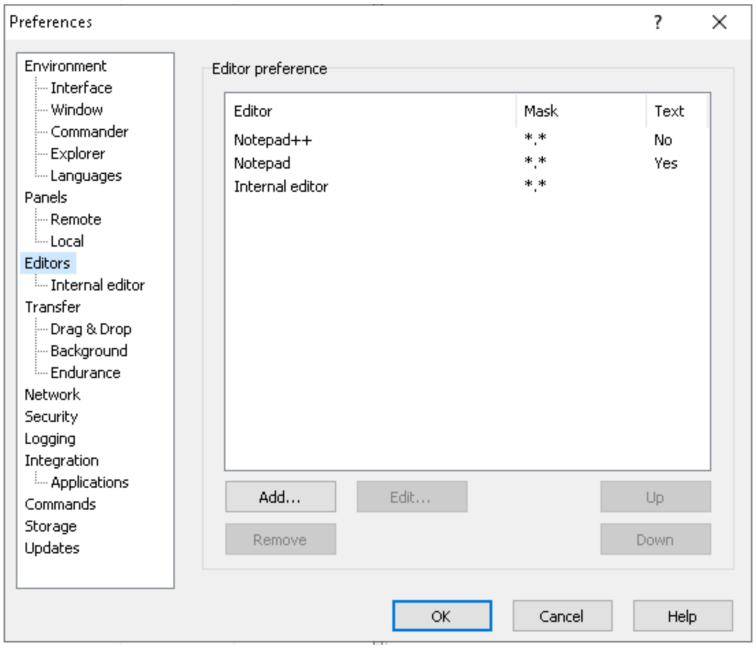

Figure 4: Changing the preferred editor in WinSCP under the menu *Option/Preferences*.

3 Quick start on Windows

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

3.1 Running MPGOS in Visual Studio

Here it is assumed that the above software are already installed. In this case you are able to create a New CUDA project by using Microsoft Visual Studio (MVS). First, run Microsoft Visual Studio and create a new project (File/New/Project or using the hotkeys ctrl+shift+N). In the New Project window (see. Fig. 5) choose the CUDA project. You can name your project and specify its location on your hard drive, then click OK.

Now, Visual Studio sets up the initiated CUDA project and generates a sample source file named kernel.cu (Fig. 6). It is worth mentioning that one may experience errors in this code, e.g.: the command of the Kernel launch " <<< >>> " is not recognized properly by the code editor. If CUDA toolkit has been installed properly beforehand, as you click on the "Play" button ("Local Windows Debugger"), the source

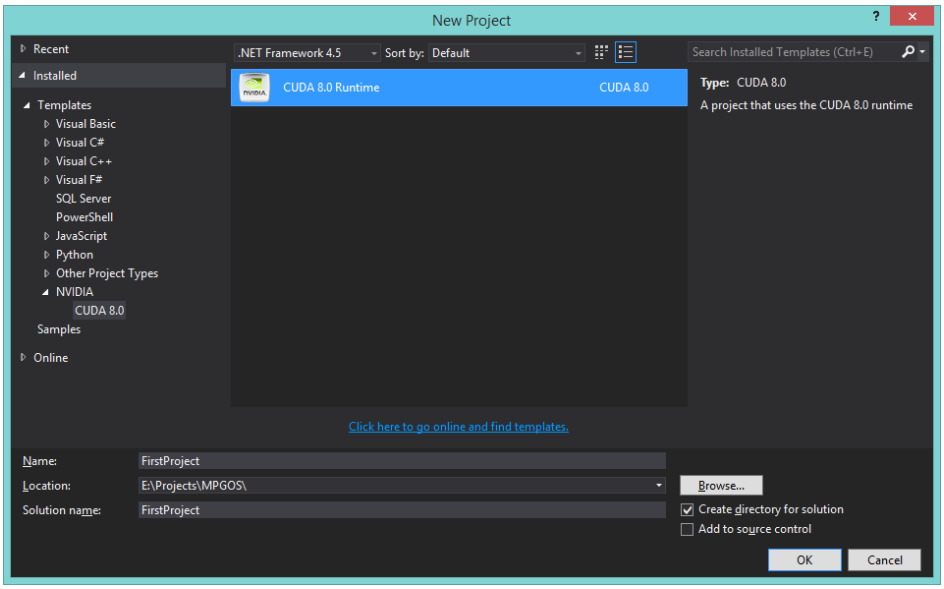

Figure 5: Creating a new CUDA project

code has to be compiled and run by Visual Studio without any complication. Since this kernel.cu file is a simple default example that comes by creating a new CUDA project, it has to be deleted from the project.

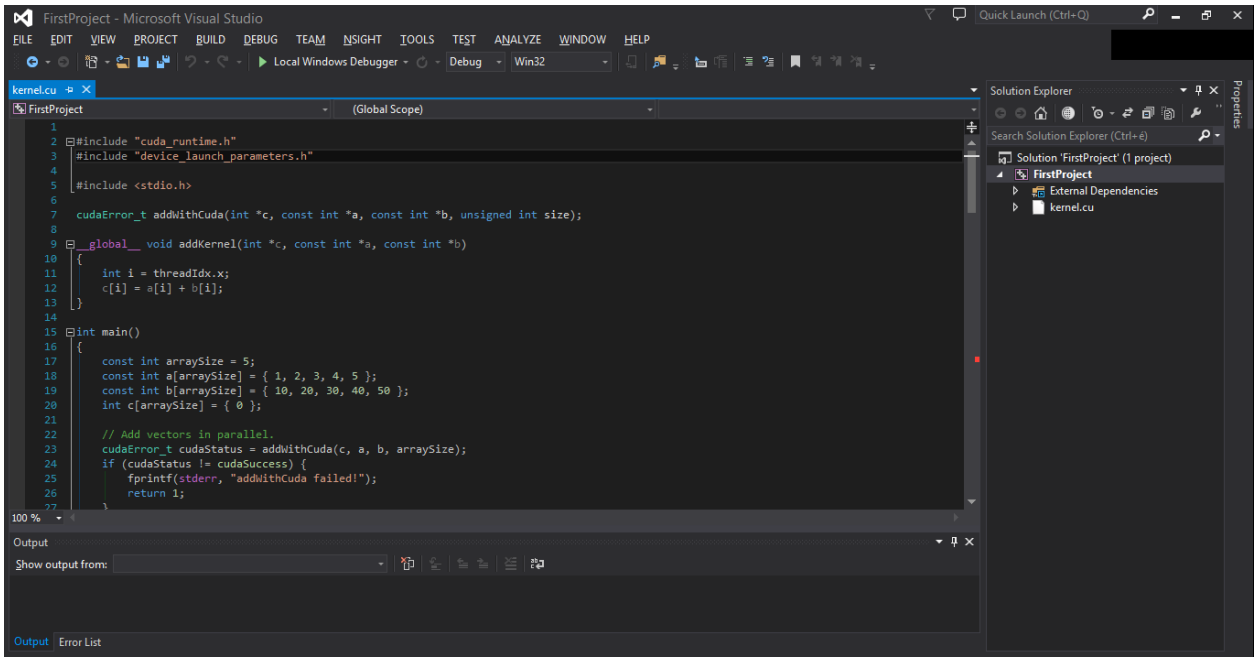

Figure 6: The default CUDA project in the Microsoft Visual Studio

At this point, the MPGOS package has to be downloaded from the GitHub repository: https://github.com/FerencHegedus/M
Parallel-GPU-ODE-Solver. By clicking on the "Clone or download" button and choosing "Download Zip", the package can be downloaded into your PC. Unzip, then copy the following files from the **PerThread** folder into the CUDA project folder:

- **MassivelyParallel_GPU-ODE_Solver.cu**

- **MassivelyParallel_GPU-ODE_Solver.cuh**
- **PerThread_RungeKutta.cuh**
- **PerThread_SystemDefinition.cuh**
- **Reference.cu**

One has to add these source files to the project as well. In Visual Studio, under the "Project" menu, click on "Add Existing Item" or use the hotkeys (Shift+Alt+A), browse your project folder and add the above listed files. Now, an MPGOS project is ready. You will see the MPGOS source files in the Solution Explorer tab, see Fig. 7. You can build and run this project, and it will compute the Duffing equation provided by this manual as a reference example.

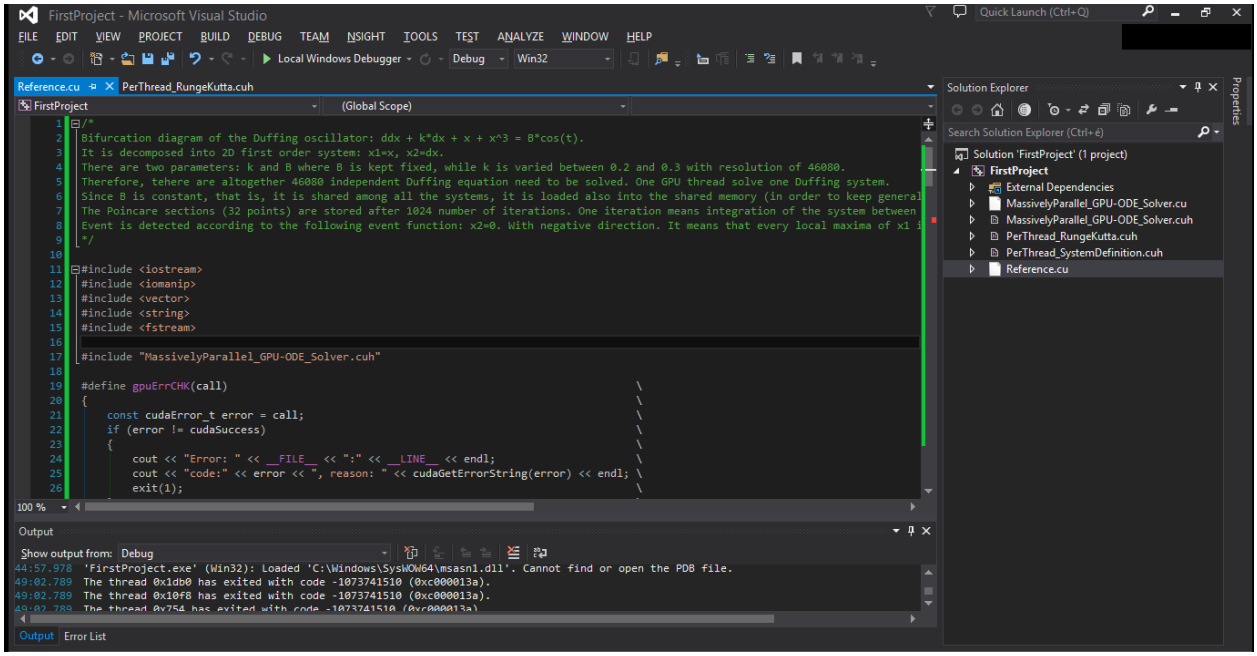

Figure 7: MVS after adding the MPGOS source files

4 Overview of the capabilities

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

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

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

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

The **shared parameters** are common for all instances of the investigated systems. They are parameters of Eq. (1); however, their values do not change from system-to-system. This is the reason they are called shared parameters (shared among all instances of systems). Defining common parameters as shared, the required number of slow memory transactions can be significantly reduced, see also Sec. 13. This can be extremely important in memory bandwidth limited applications.

The last kind of parameters are called **accessories** which are multi-purpose (user programmable) parameters. They are not strictly a parameter of Eq. (1) rather than storages that can be updated after every successful time step or after every successful event detection. In this regard, the number of accessories are independent of Eq. (1), and it is absolutely under the control of the user. Accessory variables are very efficient tools to continuously calculate, monitor and store special properties of the solutions. Throughout the following sections (the first tutorial example) and in the additional tutorial examples in Sec. 14, the efficient use and the capabilities of the accessories are emphasized.

One of the main strength of the package MPGOS is the built-in event detection algorithm. Its usage is very easy and similar to that of available in MATLAB. However, some additional features are incorporated: cooperation with the user programmable accessories, and the possibility to define "action" after a successful event detection to efficiently handle non-smooth dynamics (e.g. impact dynamics, see Sec. 14.2).

The current version of the program code is not capable to provide a dense output of the solutions. It calculates only the endpoint of every integration phases. The reason is to avoid excessive memory store/load operations, and thus maximise the arithmetic performance (this is the main purpose of using GPUs). As a compensate, the event handling and the user programmable accessories are designed in a way that **a large variety of special properties of a solution can be stored and extracted**. In my experience on many dynamical systems, the event handling-accessories combo was more than sufficient, see again the subsequent sections and the tutorial examples in Sec 14.

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

5 Important terms and definitions

There is no such term as GPU programming. During code development, the CPU has a major role to organize workload to a GPU or to many GPUs. Therefore, there is always a CPU and GPU programming. In this regard, the GPU can be viewed as a co-processor which handles the most resource intensive parts of a simulation. It has an extremely high computational throughput, but it cannot manage the main control flow of a program. Always the CPU is who tells the GPU what to do and when. Although the present program package tries to hide most of 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.

6 Detection and selection of CUDA capable devices

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

```
|| ListCUDADevices();
```

lists all the CUDA capable devices in a machine and print their most important properties to the screen. In our test PC, the print results can be seen in the listing below. There are two devices: A GeForce GTX TITAN Black (device number 0) and a GeForce GTX 550 Ti (device number 1). This is a very easy way to obtain the serial numbers (device numbers) of the existing devices. It is important since different GPUs can have very different processing powers and capabilities. Moreover, a CUDA code has to be compiled by specifying the

Table 1: Summary of the frequently user terms

Term	Description
Host	Synonyms of CPU
Device	Synonyms of GPU
Host Function	An "ordinary" function called from the Host and running on the Host as well.
Kernel Function	A function called from the Host and running on the Device. Declared with the __global__ qualifier.
Device Function	A function called from the Device and running on the Device. Declared with the __device__ qualifier. It can be called from a kernel function or from another device function.
Host Code	Pars of the program running on the CPU
Device Code	Part of the program running on the GPU. Code blocks inside a kernel function or code blocks used in a device function.
System Memory	A memory type visible by the codes running on the CPU. It is the memory plugged into the motherboard and managed by the the operating system.
Global Memory	A memory type visible by the codes running on the GPU. It is the memory that can be found on the graphics card and managed by the Nvidia graphics driver.
Shared Memory	A programmable and fast memory on the device. 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.
Problem Pool	A built-in class designed to store the required data of the whole conmputational project.
Solver Object	A built-in class designed to perform integration of smaller number of systems (smaller than or equal to the Problem Pool).

architecture of the used GPU. This is characterised by two number: a major revision and a minor revision. In the example below, for instance, the TITAN Black card has a major revision 3 and a minor revision 5, which is indicated by the number 3.5 in the *Compute capability* row (CC 3.5 for short). For computations throughout this manual, the TITAN Black card is used due to its much higher double precision floating point processing power (1707 GFLOPS). Therefore, the file `MassivelyParallel_GPU-ODE_Solver.cu` is compiled by the option `--gpu-architecture=sm_35` to indicate CC 3.5, see the `makefile` in the same folder. 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.

```
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
```

After the inspection of the list of devices, the required GPU can be associated to an instance of a built-in class called Solver Object. It is introduced in Sec. 10. In this way, the proper selection of a device (GPU) is automatically handled transparently by the Solver Object 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 Solver 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);
```

7 The Duffing equation as a first tutorial example

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

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

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

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

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

- Storage of the Poincaré sections. This can be done simply by integrating the system over the time domain $t \in [0, T]$ several times, and register the subsequent end points of each integration phase. For a thorough discussion on how to define the right-hand side of the system and how to setup the adaptive time marching, see Secs. 12.1 and 12.2.
- Two events are detected. The first event function is $F_{E1} = x_2$, which detects the states where $x_2 = 0$. As a specialisation, only events with a negative tangent of F_{E1} is detected. Since $\dot{x}_1 = x_2$, this event will detect only the local maxima of x_1 (excluding local minima due to the negative tangent restriction). To show how multiple event functions can be defined, a second event function is prescribed as $F_{E2} = x_1$ without restriction on the direction (tangent of F_{E2}). For more details, see Secs. 12.3 and 12.4.
- 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** for each system as additional user programmable storage elements. For the details on how to take advantage of the feature "action" after every successful event detection, see Sec. 12.5.
- The global maxima of each integration phases are also stored. This needs an additional, **third accessory**. The process is discussed in Sec. 12.6.

- The final task is to set the initial time step of an integration phase by the time step of the last step of the previous integration procedure. This needs a further, **fourth accessory** to store and update the initial time step. The detailed discussion of the corresponding pre-declared user functions are found in Sec. 12.7 and 12.8. Parenthetically, the initial time step of the very first integration phase must still be initiated by "hand".

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

Table 2: Summary of the size related notations

Notation	Description
N_{SD}	System dimension; that is, the number of the **state variables** x. The length of vector $\underline{x}$ in Eq. (1). In the reference tutorial example (Duffing equation) it is $N_{SD} = 2$.
N_{CP}	Number of the **control parameters**. In the reference tutorial example it is $N_{CP} = 1$.
N_{SP}	Number of the **shared parameters**. In the reference tutorial example it is $N_{SP} = 1$.
N_{ACC}	Number of the user programmable **accessories**. In the reference tutorial example it is $N_{ACC} = 4$. It is under the control of the user independently from the actual system.
N_{EF}	Number of the event functions. In the reference tutorial example it is $N_{EF} = 2$.
N_P	The total number of problems one intends to solve. Its value is $N_P = 46080$ for the reference case. For more details, see Sec. 9.
N_T	The number of GPU threads during an integration. Its value is $N_T = 23040$ for the reference case. For more details, see Sec. 10.

8 Workflow in a nutshell

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

The first one is responsible to collect all the data for all the systems to be integrated. These are the integration time domain, the initial conditions, the control parameters, the shared parameters and the initial values of the user programmable accessories. The program offers a built-in class called `ProblemPool` to manage this task. An instance of this object is called Problem Pool throughout this manual, see also Tab. 1. Its only responsibility is the proper initialisation (and the corresponding memory allocations in the System Memory)

of the whole computational project. There is usually one instance (object) of the class **ProblemPool** on a single machine or on a single node in a cluster.

The second built-in class called **ProblemSolver** manages the computational process on a smaller chunk of systems. An instance of this object is called Solver Object throughout this manual, see also Tab. 1. Its responsibility is to make the necessary memory allocations on both the System Memory (Host side) and Global Memory (Device side), to get some portion of data from the Problem Pool, to perform the numerical integration on the Device and to copy/synchronise the results back to the System Memory (Host side) from the Global Memory (Device side). On a single machine/node there can be more than one instances of the class **ProblemSolver**. Each can be responsible for computations performed on a specific GPU card. In our reference tutorial example, there is only one instance of both the **ProblemPool** and the **ProblemSolver** classes.

It is important to note that the Solver Object allocates memory on both the Host and Device side, while the ProblemPool allocates memory only on the Host side (usually a much larger portion compared to the SolverObject). Moreover, to properly hide data, the two objects cannot "see" each other's data on the Host side. That is, the data inside the Solver Object is really a copy of the data in the ProblemPool.

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

1. Create an instance of the class **ProblemPool**.
2. Fill the Problem Pool with valid data: integration time domains, initial conditions, control parameters, shared parameters and accessories.
3. Create an instance of the class **ProblemSolver**.
4. Copy some data from the Problem Pool to the Solver Object.
5. Perform numerical integration. It can be done even repeatedly one after another (e.g. simulating initial transients).
6. Meanwhile, collect the required results.
7. Save the collected results into the disk.
8. Repeat the copy—integration/collection—save cycle (phases 4, 5/6 and 7) until all the systems defined in the Problem Pool have been processed.

To be able to perform the integration phase, proper implementation of a system is required. For instance, the right-hand side of the ODEs or the event functions for event detection. These functions have to be implemented inside a few pre-declared device functions, which can be found in the file **PerThread_SystemDefinition.cuh**. The name of these functions should not be changed; otherwise, the solver would not find them. This is a necessary inconvenience as function pointers cannot be passed to a kernel function. These pre-declared user functions are very important parts of a project; therefore, a whole section is devoted to their proper description in Sec. 12.

It must be emphasised that the workflow introduced here is used during the main part of this manual through the reference example. That is, as stated above, using a single problem pool, a single solver object and a single GPU. Due to such simplicity, this simple example uses functionalities of the solver object that cannot overlap CPU and GPU computations, and cannot use multiple GPUs even if they are in a single node/machine. The reason for this is to introduce the main features/ideas of the program package as simply as possible. However, MPGOS offers the aforementioned computation possibilities. Since they need more complicated control logic, they are omitted from the main part of the manual, and these features are introduced via additional tutorial examples, see e.g. Sec. 14.3 for an example to overlap CPU and GPU computations, or Sec. 14.4 for an example of multi-GPU usage.

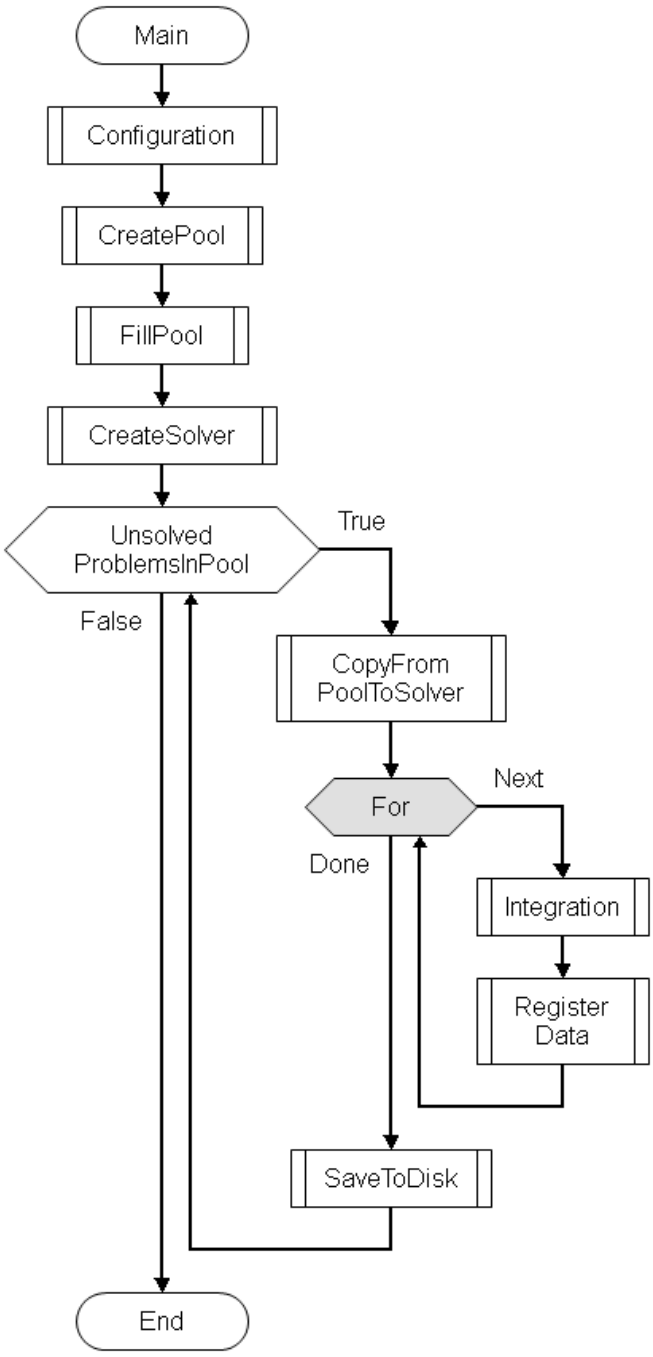

Figure 8: Typical workflow using a single problem pool, a single solver object and a single GPU.

9 The problem pool object

During the initialisation of a Problem Pool, the object needs to know information on how much System Memory has to be allocated. This depends on the various size parameters summarized in Tab. 2. These basic information on the system definitions are collected in a built-in structure **ConstructorConfiguration**. The declaration of an instance and its initialisation for our reference tutorial case is summarised below

```
int PoolSize        = 46080;
int NumberOfThreads = 23040;
```

```
...
ConstructorConfiguration ConfigurationDuffing;

ConfigurationDuffing.PoolSize                  = PoolSize;
ConfigurationDuffing.NumberOfThreads           = NumberOfThreads;
ConfigurationDuffing.SystemDimension           = 2;
ConfigurationDuffing.NumberOfControlParameters = 1;
ConfigurationDuffing.NumberOfSharedParameters  = 1;
ConfigurationDuffing.NumberOfEvents            = 2;
ConfigurationDuffing.NumberOfAccessories       = 4;
```

Observe how the values of the data members agree with the values also given in Tab. 2. The data member **PoolSize** defines the total number of the systems N_P whose data is stored in the Problem Pool. In this tutorial example it is only 46080; however, usually it is in the order of hundreds of thousands or even millions. The data members **NumberOfThreads** and **NumberOfEvents** are not used during the initialisation of the Problem Pool. They will be important during the initialisation of the Solver Object, see Sec. 10. The total allocated memroy size of the Problem Pool is

$$S_P(bytes) = N_P \cdot SoD \cdot (N_{SD} + N_{CP} + N_{ACC} + 2) + SoD \cdot N_{SP}, \tag{4}$$

where SoD is the size of a **double** in bytes (8 bytes). In the reference tutorial example, the total size is approximately 3240 Kb. Observe that in the contribution of the shared parameters (last term), the multiplication with N_P is missing as these parameters are shared among all the problems (we need only one for each). Moreover, the constant value 2 in the last term of the bracket represents the two values of the time domain; the inital time and the final time of the integration phase for each system. Now the **ProblemPool** can be easily declared as

```
ProblemPool ProblemPoolDuffing(ConfigurationDuffing);
```

This declaration makes an instant of the class **ProblemPool** and its constructor performs the necessary memory allocations on the System Memory (Host side) according to the structure **Constructor Configuration**. Keep in mind that the Problem Pool (named **ProblemPool** here) has no connection to any devices (GPUs).

9.1 Filling up the problem pool

The next task is to properly fill up the Problem Pool with valid data. First, a **time domain** $t \in (t_0, t_1)$ has to be assigned to each system; the initial time t_0 and the final time t_1 of the integration phases. It must be emphasized that every system has its own internal track of the actual time, and correspondingly has its own separate time domain. In other words, every system is indeed totally independent of each other including the time stepping and the error control processes in the adaptive algorithms. Second, the **initial conditions** have to be defined. As usual, to each system, as many initial conditions have to be specified as the system dimension N_{SD}. In our reference case, it is 2: $x_1(t_0) = x_{10}$ and $x_2(t_0) = x_{20}$. Third, the **control parameters** have to be given. The only control parameter in our reference case is the damping factor k. It is distributed evenly between 0.2 and 0.3 with a resolution of 46080 (this is exactly the size of the problem pool N_P). That is, a single value of k is assigned to each system. Finally, the values of the **accessories** have to be initialised. It is not a mandatory task here as they can be initialised also via the pre-declared user functions discussed in Sec. 12.7. Nevertheless, we are initialising them with 0. The only exception is the fourth accessory in which the initial time step is stored for each integration phases. Therefore, it must be properly initialised immediately during the filling of the problem pool, since there is no built-in estimation for the initial time step and the very first integration phase has to be initialised here. Later on, during the integration phases, its content is managed and updated by the pre-declared user functions, see Sec. 12.8.

The aforementioned assigments can be easily done by the member function

```
ProblemPoolDuffing.Set(ProblemNumber, ActualVariable, SerialNumber, Value);
```

of the Problem Pool. The **ProblemNumber** is the serial number of the system in the pool ranged between 0 and 46079 according to the total number of problems N_P which is also the size of the pool. Keep in mind that the indexing starts from 0 instead of 1 in C++. The input parameter **ActualVariable** determines the variable whose value is about to set. It is a built-in enumerated type (**VariableSelection**) and its valid options here are **TimeDomain**, **ActualState**, **ControlParameters** and **Accessories**. With the option **ActualState**, the initial conditions can be set (the actual state of a system before any integration is the initial state). The **SerialNumber** is the serial number of the **ActualVariable**. For instance, the serial number of t_1 is 1, the serial number of the first control parameter is 0 or the serial number of the third accessory is 2 (keep in mind again the start of the indexing). The variable **Value** is self-explanatory, it is the value of the corresponding variable to be set. The following listing shows an easy way to fill the created problem pool (wrapped into a function called `void FillProblemPool(...)` in the file **Reference.cu**):

```
#define PI 3.14159265358979323846
...
double X10 = -0.5;
double X20 = -0.1;

int ProblemNumber = 0;
for (auto const& k: k_Values)
{
   ProblemPoolDuffing.Set(ProblemNumber, TimeDomain,  0, 0 );
   ProblemPoolDuffing.Set(ProblemNumber, TimeDomain,  1, 2*PI );

   ProblemPoolDuffing.Set(ProblemNumber, ActualState, 0, X10 );
   ProblemPoolDuffing.Set(ProblemNumber, ActualState, 1, X20 );

   ProblemPoolDuffing.Set(ProblemNumber, ControlParameters, 0, k );

   ProblemPoolDuffing.Set(ProblemNumber, Accessories, 0, 0 );
   ProblemPoolDuffing.Set(ProblemNumber, Accessories, 1, 0 );
   ProblemPoolDuffing.Set(ProblemNumber, Accessories, 2, 0 );
   ProblemPoolDuffing.Set(ProblemNumber, Accessories, 3, 1e-2 );

   ProblemNumber++;
}
```

In the first two rows, the initial conditions (fixed) are set. From system-to-system, only the value of the damping constant k varies. Therefore, we need only one **for** cycle to iterate through its dataset. The values of k is stored in a standard C++ container **k_Values** of type **vector<double>**. Its proper filling is not shown here, for details, please check the file **Reference.cu** or see Sec. 11. The first two rows inside the cycle set the time domain (the values of t_0 and t_1). The final time is 2π (PI is defined in the top of the code) which is the period of the harmonic forcing. Therefore, the end state of an integration phase is a point in the Poincaré section of the system. The next two rows set the initial conditions, the fifth row specifies the sole control parameter and the last four rows initialise the accessories. The first three accessories are initialised with zero, but the last (fourth) accessory is initialised by 10^{-2} which is the initial time step common to all systems for the first integration phase. Observe how the number of the rows here correlates with the values given in the structure **ConstructorConfiguration** introduced at the beginning of this section and with the values presented in Tab. 2.

The shared parameters have to be specified a little bit differently. Since every system share these parameters, they need to be given only once with a specialised member function

```
ProblemPoolDuffing.SetShared(SerialNumber, Value);
```

That is, we need to specify only the serial number of a shared parameter and its corresponding value. In the reference simulation, there is only one shared parameter; thus, a single statement can perform this action:

```
double B = 0.3;

ProblemPoolDuffing.SetShared(0, B );
```

After filling the whole problem pool, it might be necessary to query some of its already written values. The next two member functions return the requested values:

```
ProblemPoolDuffing.Get(ProblemNumber, ActualVariable, SerialNumber);
ProblemPoolDuffing.GetShared(SerialNumber);
```

The details of these functions are omitted here as they can be easily understood according to the above discussion. The return values of these functions are of type **double**, and the type of the variable **ActualVariable** is still the built-in enumerated type **VariableSelection**.

As a final tool, the content of the problem pool can be checked by writing it into separate text files with the following call of the member function

```
ProblemPoolDuffing.Print(ActualVariable);
```

According to the vale of **ActualVariable**, the values for the time domain, actual state (initial condition), control parameters, shared parameters and accessories are written into the files: **TimeDomainInPool.txt**, **ActualStateInPool.txt**, **ControlParametersInPool.txt**, **SharedParametersInPool.txt** and **AccessoriesInPool.txt**, respectively. The **ProblemNumber** is organised in rows while the **SerialNumber** is organised in columns.

In summary, Tabs. 3 and 4 list all the member functions of the class **ProblemPool** and the possible values of the enumerated type **VariableSelection**.

Table 3: Summary of the member functions of the class **ProblemPool**

Function	Arguments	Description
void Set(*args*)	**int ProblemNumber** **VariableSelection ActualVariable** **int SerialNumber** **double Value**	Set the value of a variable in the problem pool according to the system number and the serial number.
void SetShared(*args*)	**int SerialNumber** **double Value**	Set the value of a shared parameter according to its serial number.
double Get(*args*)	**int ProblemNumber** **VariableSelection ActualVariable** **int SerialNumber**	Returns the value of a variable according to its serial number and the system number.
double GetShared(*arg*)	**int SerialNumber**	Returns the value of a shared parameter according to its serial number.
void Print(*arg*)	**VariableSelection ActualVariable**	Print the content of a variable in the problem pool into a text file according to the value of the variable selection.

10 The solver object

After the initialisation of a Problem Pool, the next task is to make an instance of the class **ProblemSolver**. This object is initialized with the same built-in structure **ConstructorConfiguration** used also for the class **ProblemPool**, see the first code listing in Sec. 9. In the initialisation of the Solver Object, the data member **PoolSize** is not used. However, the data member **NumberOfThread** (N_T) is now an important

Table 4: Summary of the possible values of the enumerated type **VariableSelection**

Value
TimeDomain
ActualState
ControlParameters
Accessories
SharedParameters (only for `Print()`)
All (only for data copy, see the next section)

input parameter. During the integration, the Solver Object integrates N_T number of independent systems in parallel. In our case, this means $N_T = 23040$ number of Duffing equations, see again the first listing in Sec. 9 or Tab. 2. Observe, that it is exactly the half of the **PoolSize** (N_P). Therefore, the total number of systems $N_P = 46080$ cannot be solved at the same time. One needs to copy the frist half of the problems into the Solver Object from the Problem Pool, perform the necessary integrations, data collections and save operations (first launch). Next, one has to do the same tasks on the second half of the systems (second launch). In general, the required launches for a simulation is the ratio of the problem size and the applied number of threads N_P/N_T.

It must be emphasized again that the Solver Object has its own allocated storages which are independent of the storages allocated during the initialisation of the Problem Pool. The reason is to clearly separate the actually launched systems. The necessary data (**TimeDomain**, **ActualState**, **ControlParameters**, **SharedParameters** and **Accessories**) have to be stored both in the System Memory (Host side) and the Global Memory (Device side). Their initial values are copied form the Problem Pool. Later, the solver automatically synchronises their content after an integration phase. In addition, the Solver Object has to allocate many other arrays in the Global Memory needed for the numerical algorithms; such arrays are allocated only on the Device side.

The creation of an instance of the solver object **ProblemSolver** can be done with the following instruction

```
ProblemSolver ScanDuffing(ConfigurationDuffing, DeviceNumber);
```

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

10.1 Filling up the solver object

The simplest case to initialise data of the Solver Object (now **ScanDuffing**) from the Problem Pool is to use the following member function of the **ScanDuffing** solver object

```
ScanDuffing.LinearCopyFromPoolHostAndDevice(ProblemPoolDuffing, CopyStartIndexInPool,
    CopyStartIndexInSolverObject, NumberOfElementsCopied, CopyMode);
```

It copies the data of **NumberOfElementsCopied** number of systems from the problem pool to the solver object. Since this function is a member of the solver object, the problem pool **ProblemPoolDuffing** has to be specified in the first argument. It is passed by reference; that is, its type is **const ProblemPool&** to avoid unnecessary memory transactions. The starting index of the first system in the pool is the second argument, the third one is the corresponding starting index in the solver object. The fourth argument is the number of systems whose data is copied from the problem pool to the solver object. The last argument **CopyMode**

determines which data are to be copied; its type is the familiar enumerated type **VariableSelection**. Its possible values here are **TimeDomain**, **ActualState**, **ControlParameters**, **Accessories** and **All**. The option **All** applies the copy command for all the variables. The **SharedParameters** option is not valid here, shared parameters are copied with another member function. The following listing performs the copy of data of the two launches in the reference calculation organised into a **for** cycle

```
int NumberOfSimulationLaunches = PoolSize / NumberOfThreads;

int CopyStartIndexInPool;
int CopyStartIndexInSolverObject = 0;
int NumberOfElementsCopied       = NumberOfThreads;

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

   ScanDuffing.LinearCopyFromPoolHostAndDevice(ProblemPoolDuffing, CopyStartIndexInPool,
       CopyStartIndexInSolverObject, NumberOfElementsCopied, All);

   // Simulations, collaection of data and save them to disc
   ...
}
```

Observe how the **LaunchCounter** variable properly adjusts the starting index of the copy process in the problem pool within the **for** cycle. The **NumberOfSimulationLaunches** is exactly 2 as it is discussed before, since the pool size is $N_P = 46080$ and the number of the used threads is $N_T = 23040$, and their ratio is $N_P/N_T = 2$. Be cautious if this ratio is not an integer number, in that case, special care is needed during the last copy process to adjust the variable **NumberOfElementsCopied** for the remaining systems. This is handled properly in the reference example in the file **Reference.cu**, see also the full listing of the code in Sec. 11. Keep in mind that in case of such a partial copy, some old data will still remain inside the Solver Object, and the integration should be omitted on those "remnants". This can be easily set up by specifying the active number of threads during an integration, see again Sec. 11 for more details.

The **SharedParameters** have to be copied only once as they are the same for all systems and for all launches. The following member function call

```
ScanDuffing.SharedCopyFromPoolHostAndDevice(ProblemPoolDuffing);
```

will do the job, where the only argument needed is the problem pool **ProblemPoolDuffing**. Again, it is passed by reference to avoid extensive memeory trasanctions.

Similarly as for the problem pool, the content (data of every system) of the solver object can also be printed into files via the member function

```
ScanDuffing.Print(ActualVariable);
```

The type of the variable **ActualVariable** is the enumerated type **VariableSelection**. Its values **TimeDomain**, **ActualState**, **ControlParameters**, **SharedParameters** and **Accessories** writes the correspoding data into the following files: **TimeDomainInSolverObject.txt**, **ActualStateIn SolverObject.txt**, **ControlParametersInSolverObject.txt**, **SharedParametersInSolver Object.txt** and **AccessoriesInSolverObject.txt**, respectively. Similarly, the **ProblemNumber** is organised in rows while the **SerialNumber** is organised in columns.

10.2 Extract data from the solver object

After every integration phase, the data corresponding to the **Timedomain**, the **ActualState** and **Accessories** are overwritten with the new results both on the Global Memory (Device side) and on the System Memory (Host side). On how to perform the integration phases, see Sec. 11.

The simulated results can be accessed via the member functions

```
ScanDuffing.SingleGetHost(ProblemNumber, ActualVariable, SerialNumber);

ScanDuffing.SharedGetHost(SerialNumber);
```

The input arguments are self-explanatory and they were discussed thoroughly in this and the previous sections; thus, it is omitted here. The following code snippet shows how to store the computed quantities

```
ofstream File;
File.open ( "Duffing.txt" );
int Width = 18;
File.precision(10);
File.flags(ios::scientific);

for (int LaunchCounter=0; LaunchCounter<NumberOfSimulationLaunches; LaunchCounter++)
{
   // Copy data from problem pool to solver object

   // Transient iterations
   for (int i=0; i<1024; i++)
   {
      // Perform integration
      ...
   }

   // The data of the next 32 integration phases are stored
   for (int i=0; i<32; i++)
   {
      // Perform integration
      ...

      for (int idx=0; idx<NumberOfThreads; idx++)
      {
      File.width(Width); File<<ScanDuffing.SingleGetHost(idx, ControlParameters, 0)<<',';
      File.width(Width); File<<ScanDuffing.SharedGetHost(0)<<',';
      File.width(Width); File<<ScanDuffing.SingleGetHost(idx, ActualState, 0)<<',';
      File.width(Width); File<<ScanDuffing.SingleGetHost(idx, ActualState, 1)<<',';
      File.width(Width); File<<ScanDuffing.SingleGetHost(idx, Accessories, 0)<<',';
      File.width(Width); File<<ScanDuffing.SingleGetHost(idx, Accessories, 1)<<',';
      File.width(Width); File<<ScanDuffing.SingleGetHost(idx, Accessories, 2)<<',';
      File.width(Width); File<<ScanDuffing.SingleGetHost(idx, Accessories, 3)<<',';
      File << '\n';
      }
   }
}

File.close();
```

The first part of the code opens a file to store data (**Duffing.txt**) and sets the output format and precision. The outer **for** cycle is responsible to iterate through the problem pool as it was already discussed in this section. Inside this loop, the first inner **for** cycle is responsible to integrate through the initial transient. In our experience, 1024 integration phases are enough for the trajectories to settle down to a stable solution (attractor). As a reminder, a single integration phase means the integration of the system in the time domain $(0, 2\pi)$. Parenthetically, starting the integration always from $t_0 = 0$ is not a problem, since the system is time-periodic exactly by 2π. The second inner **for** cycle is aimed to perform some further iterations of the converged trajectory and store some of its properties. Altogether 32 number of subsequent iterations are performed. Observe how the `SingleGetHost()` and `SharedGetHost()` functions are called altogether eight times for all the threads (**NumberOfThreads**) in each iteration phase to store the required values to the text file for all the integrated systems residing in the solver object.

The first two function calls store the sole control and shared parameters into the first two columns of the text file. As the control parameter varies from system-to-system, the corresponding first column in the text file should vary accordingly. The second column, on the other hand, should be filled with the constant shared parameters with the value of $B = 0.3$. The next two lines (function calls) store the points of the Poincaré section of the system. Poincaré section means sampling the continuous trajectory by the period

of the excitation of the system. Therefore, storing the endpoints of each integration phase is exactly the points of the Poincaré section. The next four lines store the values of the accessories. These values are continuously updated during the integration phases via the pre-declared user functions, their detailed setup is discussed in Sec. 12. Observe that the number of lines in the text file shall be pool size N_P times 32; that is, $46080 \times 32 = 1474560$. The number of columns is eight. Observe also that the data is written in two periods into the file, as the number of the simulation launches is 2, see the above discussion. Finally, the text file is closed.

In summary, Tab. 5 lists all the member functions of the Solver Object to copy data from the Problem Pool, to obtain a data from its internal storages, and write its data into a text file for a quick check.

Table 5: Summary of the member functions of the class **SolverObject**

Function	Arguments	Description
void LinearCopyFromPool HostAndDevice(*args*)	**const ProblemPool&** **ProblemPoolDuffing** **int CopyStartIndexInPool** **int CopyStartIndex** **InSolverObject** **int NumberOfElementsCopied** **VariableSelection CopyMode**	Copy a linear portion of the data from the given problem pool into the internal storages of the solver object. The starting indices in both the pool and solver have to be specified together with the number of elements copied. For the copy mode options, see Tab. 4.
void SharedCopyFromPool HostAndDevice(*args*)	**const ProblemPool&** **ProblemPoolDuffing**	Copy the shared parameters from the problem pool into the solver object.
double SingleGetHost(*args*)	**int ProblemNumber** **VariableSelection** **ActualVariable** **int SerialNumber**	Returns the value of a variable according to its serial number and its problem number. The problem number cannot be larger than the number of threads.
double SharedGetHost(*arg*)	**int SerialNumber**	Returns the value of a shared variable according to its serial number.
void Print(*arg*)	**VariableSelection** **ActualVariable**	Prints the content of a variable of the solver object into a text file.

10.3 Filling up the solver object without using a problem pool

It is possible to write data immediately into the System Memory (Host side) of the Solver Object, and then synchronise them with their Global Memory counterpart (Device side). The style of the corresponding member functions is similar to the ones already discussed in details in Sec 10. Moreover, the name of the member functions and the name of their input arguments are self-explanatory; therefore, the possibilities are only summarised in tabulated format, see Tabs. 6 and 7. Nevertheless, short descriptions of the functions are given as usual.

11 Perform integration

After the proper data management operations, the next step is to perform one or more integration phases. The integration can be performed simply by calling the member function

```
|| ScanDuffing.Solve(SolverConfigurationSystem);
```

Table 6: Summary of the member functions of the class **SolverObject** to fill up the Solver Object with data directly (without using a Problem Pool).

Function	Arguments	Description
void SingleSetHost(*args*)	**int ProblemNumber** **VariableSelection** **ActualVariable** **int SerialNumber** **double Value**	Sets the value of a variable in the solver object according to the system number and the serial number of the variable. The variable is set only in the System Memory (Host side); that is, synchronisation with the Global Memory (Device side) will be necessary.
void SingleSet HostAndDevice(*args*)	**int ProblemNumber** **VariableSelection** **ActualVariable** **int SerialNumber** **double Value**	Sets the value of a variable in the solver object according to the system number and the serial number of the variable. The variable is set both in the System Memory (Host side) and the Global Memory (Device side). Therefore, synchronisation is not necessary. Copy a single value into a device memory is, however, NOT efficient.
void SetSharedHost(*args*)	**int SerialNumber** **double Value**	Sets the value of a shared parameter according to its serial number. The variable is set only in the System Memory (Host side); that is, synchronisation with the Global Memory (Device side) will be necessary.
void SetShared HostAndDevice(*arg*)	**int SerialNumber** **double Value**	Sets the value of a shared parameter according to its serial number. The variable is set both in the System Memory (Host side) and the Global Memory (Device side). Therefore, synchronisation is not necessary. Copy a single value into a device memory is, however, NOT efficient.
void SynchroniseFrom HostToDevice(*arg*)	**VariableSelection** **ActualVariable**	Synchronises the selected variable from the System Memory (Host side) into the Global Memory (Device side). After filling the variables with the member function SingleSetHost(), this command is necessary to put data into the Global Memory. Otherwise, the integration will be performed on undefined data. Here, the **All** option is valid as an input argument.
void SynchroniseFrom DeviceToHost(*arg*)	**VariableSelection** **ActualVariable**	Synchronises the selected variable from the Global Memory (Device side) into the System Memory (Host side). This member function is seldom used in the present version of the code since, after every integration phase, the new results are automatically synchronised. The **All** option is valid.

Table 7: Continuation of Tab. 6.

Function	Arguments	Description
void SynchroniseShared FromHostToDevice()		Synchronises all the shared parameters from the system memory (Host side) into the global memory (Device side). After filling the variables with the member function SetSharedHost(), this command is necessary to put data into the global memory. Otherwise, the integration will be performed on undefined data.
void SynchroniseShared FromDeviceToHost()		Synchronises all the shared parameters from the global memory (Device side) into the system memory (Host side). This member function is seldom used in the present version of the code, as the shared parameters do not change during an integration phase. That is, no synchronisation back is necessary.

where the variable **SolverConfigurationSystem** is a built-in structure of type **SolverConfiguration** that determines the behaviour of the solver. An example of a proper function call to integrate all systems residing in the Solver Object is as follows

```
int BlockSize = 64;

SolverConfiguration SolverConfigurationSystem;

SolverConfigurationSystem.BlockSize       = BlockSize;
SolverConfigurationSystem.InitialTimeStep = 1e-2;
SolverConfigurationSystem.Solver          = RKCK45;
SolverConfigurationSystem.ActiveThreads   = NumberOfThreads;

ScanDuffing.Solve(SolverConfigurationSystem);
```

The variable **BlockSize** is an important parameter of the solver configuration which defines how the threads are organised during the run on the Device (GPU). It can have a major impact on the overall performance of the code. For details, the reader should be referred to section Sec. 13. Here, it is enough to know that it must be an integer multiple of 32. If one chooses its value to 32 or 64, the performance of the code will very likely to be high. The variable **InitialTimeStep** is self explanatory. In case of algorithms with a fixed time step, this value will be applied throughout the whole integration process. For adaptive algorithms, this is only the initial time step which is immediately adapted after the first step. An initial time step is necessary even for adaptive numerical schemes since there is no incorporated initial time step estimation in the program code. The possible options of the data member **Solver**, whose type is a built-in enumerated type of **SolverAlgorithms**, are listed in Tab. 8. It specifies the numerical algorithm. The last data member sets the active number of threads that must be lower than or equal to the **NumberOfThreads**. Even if the total amount of memory is allocated for **NumberOfThreads** number of instances of the ODE system and even if the same amount of data is copied from the Problem Pool, the integration will be performed only on the first **ActiveThreads** number of threads. In the above example the active number of threads is equal to the number of threads; that is, integration performed for every instance. This option can help to overcome the difficulty if the number of insatnces for the last simulation launch is lower than the number of threads. Try to check the reference example with different problem pool size and see how the number of the launches

changes. Even if the last launch contain only a single active threads, the code runs perfectly.

Table 8: Summary of the integration algorithms

Value	Description
RKCK45	The adaptive Runge–Kutta–Cash–Karp method with embedded error estimation of orders 4 and 5.
RK4	The classic $4th$ order Runge–Kutta scheme with fixed time step.
RKCK45_EH0	The adaptive Runge–Kutta–Cash–Karp method without event handling.
RK4_EH0	The classic $4th$ order Runge–Kutta scheme without event handling.

The last line of the above code listing performs the integration over the time domain, between t_0 and t_1. Remember that each system can has its own time domain; that is, its own t_0 and t_1. In the reference case, however, they are identical for all systems: $t_0 = 0$ and $t_1 = 2\pi$. It is important to note that the integration can be stopped due to event handling even if the integration procedure did not reached the end of the time domain t_1, for details see Secs. 12.3, 12.4 and 12.5.

After the integration phase, the new or modified values of the variables **TimeDomain**, **ActualState** and **Accessories** are updated both in the Global Memory and the System Memory; that is, they are immediately synchronised and the `SingleGetHost()` function can be immediately applied, see again Tab. 5. The variables **ActualState** are simply the endpoint of the state variable of the integration phase; the variables **Accessories** are the user programmable storages modified and/or stored via the pre-declared user functions discussed in details during section Sec. 12. Finally, the variable **TimeDomain** can also be modified during an integration; therefore, its synchronisation is also performed. The quasiperiodic forcing is an excellent example where the modification of time domain is useful, it is described in Sec. 14.1.

Due to this explicit synchronisation, the integration phases can be repeated immediately one after another just like during the transient simulation of the reference case:

```
for (int i=0; i<1024; i++)
{
    ScanDuffing.Solve(SolverConfigurationSystem);
}
```

The integration is automatically continued from the previous states of the systems. In autonomous systems, such a continuation cannot cause any problem as the system is time-independent. In periodically forced systems, however, the time domain should be chosen according to the period of the excitation (like in the reference case). Otherwise, there will be a jump in the evaluation of the driving term causing a jump in the solution as well. In every other case (e.g. the quasiperiodic forcing), the proper track of the beginning of the time domain is necessary. Fortunately, with a special pre-declared user function this task can be done easily, see again Sec. 14.1.

As a final part of this section, let us put together the `main()` function of our reference case. Now every detail should be clear; thus, we do not add any further explanations to the code snippet. The name of the variables and the name of the used functions are very clear and self-explanatory. Parenthetically, the code in the **Reference.cu** contains time measurement instructions as well. These are omitted here to shorten the code listing; therefore, the two code have some really minor differences.

```
int main()
{
    int PoolSize        = 46080;
    int NumberOfThreads = 23040;
    int BlockSize       = 64;
```

```
    ListCUDADevices();

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

    PrintPropertiesOfSpecificDevice(SelectedDevice);

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

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

    ConstructorConfiguration ConfigurationDuffing;
       ConfigurationDuffing.PoolSize                  = PoolSize;
       ConfigurationDuffing.NumberOfThreads           = NumberOfThreads;
       ConfigurationDuffing.SystemDimension           = 2;
       ConfigurationDuffing.NumberOfControlParameters = 1;
       ConfigurationDuffing.NumberOfSharedParameters  = 1;
       ConfigurationDuffing.NumberOfEvents            = 2;
       ConfigurationDuffing.NumberOfAccessories       = 4;

    CheckStorageRequirements(ConfigurationDuffing, SelectedDevice);

    ProblemSolver ScanDuffing(ConfigurationDuffing, SelectedDevice);

    ProblemPool ProblemPoolDuffing(ConfigurationDuffing);
       FillProblemPool(ProblemPoolDuffing, Parameters_k_Values, Parameters_B,
           InitialConditions_X1, InitialConditions_X2);

    //ProblemPoolDuffing.Print(TimeDomain);
    //ProblemPoolDuffing.Print(ActualState);
    //ProblemPoolDuffing.Print(ControlParameters);
    //ProblemPoolDuffing.Print(SharedParameters);
    //ProblemPoolDuffing.Print(Accessories);

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

    int NumberOfSimulationLaunches = PoolSize / NumberOfThreads + (PoolSize %
        NumberOfThreads == 0 ? 0:1);

    SolverConfiguration SolverConfigurationSystem;
       SolverConfigurationSystem.BlockSize       = BlockSize;
       SolverConfigurationSystem.InitialTimeStep = 1e-2;
       SolverConfigurationSystem.Solver          = RKCK45;
       SolverConfigurationSystem.ActiveThreads   = NumberOfThreads;

    int CopyStartIndexInPool;
    int CopyStartIndexInSolverObject = 0;

    ofstream DataFile;
    File.open ( "Duffing.txt" );
       int Width = 18;
       File.precision(10);
       File.flags(ios::scientific);

    ScanDuffing.SharedCopyFromPoolHostAndDevice(ProblemPoolDuffing);
    for (int LaunchCounter=0; LaunchCounter<NumberOfSimulationLaunches; LaunchCounter++)
    {
       CopyStartIndexInPool = LaunchCounter * NumberOfThreads;

       if ( LaunchCounter == (NumberOfSimulationLaunches-1) )
```

```
            SolverConfigurationSystem.ActiveThreads = (PoolSize % NumberOfThreads == 0 ?
                NumberOfThreads : PoolSize % NumberOfThreads);

        ScanDuffing.LinearCopyFromPoolHostAndDevice(ProblemPoolDuffing,
            CopyStartIndexInPool, CopyStartIndexInSolverObject, SolverConfigurationSystem.
            ActiveThreads, All);

        for (int i=0; i<1024; i++)
            ScanDuffing.Solve(SolverConfigurationSystem);

        for (int i=0; i<32; i++)
        {
            ScanDuffing.Solve(SolverConfigurationSystem);

            for (int tid=0; tid<SolverConfigurationSystem.ActiveThreads; tid++)
            {
            File.width(Width); File<<ScanDuffing.SingleGetHost(tid,ControlParameters,0)<<','
                ;
            File.width(Width); File<<ScanDuffing.SharedGetHost(0)<<',';
            File.width(Width); File<<ScanDuffing.SingleGetHost(tid, ActualState, 0)<<',';
            File.width(Width); File<<ScanDuffing.SingleGetHost(tid, ActualState, 1)<<',';
            File.width(Width); File<<ScanDuffing.SingleGetHost(tid, Accessories, 0)<<',';
            File.width(Width); File<<ScanDuffing.SingleGetHost(tid, Accessories, 1)<<',';
            File.width(Width); File<<ScanDuffing.SingleGetHost(tid, Accessories, 2)<<',';
            File.width(Width); File<<ScanDuffing.SingleGetHost(tid, Accessories, 3)<<',';
            File << '\n';
            }
        }
    }

    File.close();
}

// AUXILIARY FUNCTION FOR LINEAR DISTRIBUTION ------------------------------------------
void Linspace(vector<double>& x, double B, double E, int N)
{
     double Increment;

    x[0]   = B;

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

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

// AUXILIARY FUNCTION TO FILL THE PROBLEM POOL ------------------------------------------
void FillProblemPool(ProblemPool& Pool, const vector<double>& k_Values, double B, double
    X10, double X20)
{
    int ProblemNumber = 0;
    for (auto const& k: k_Values)
    {
        Pool.Set(ProblemNumber, TimeDomain,  0, 0 );
        Pool.Set(ProblemNumber, TimeDomain,  1, 2*PI );

        Pool.Set(ProblemNumber, ActualState, 0, X10 );
        Pool.Set(ProblemNumber, ActualState, 1, X20 );

        Pool.Set(ProblemNumber, ControlParameters, 0, k );

        Pool.Set(ProblemNumber, Accessories, 0, 0 );
```

```
        Pool.Set(ProblemNumber, Accessories, 1, 0 );
        Pool.Set(ProblemNumber, Accessories, 2, 0 );
        Pool.Set(ProblemNumber, Accessories, 3, 1e-2 );

        ProblemNumber++;
    }

    Pool.SetShared(0, B );
}
```

It can be seen that the main strength of the program package is that one needs only a two pages long code block to perform a detailed parameter study of a dynamical system. The resulted bifurcation diagram is shown in Fig. 9, where the first component of the Poincaré section $P(x_1)$ is plotted as a function of the control parameter k. The total simulation time of the above-described `main()` function is merely 33.1 s on an Nvidia Titan Black card (1707 GFLOPS theoretical peak processing power). This runtime includes the writing of the accumulated data to disc as well.

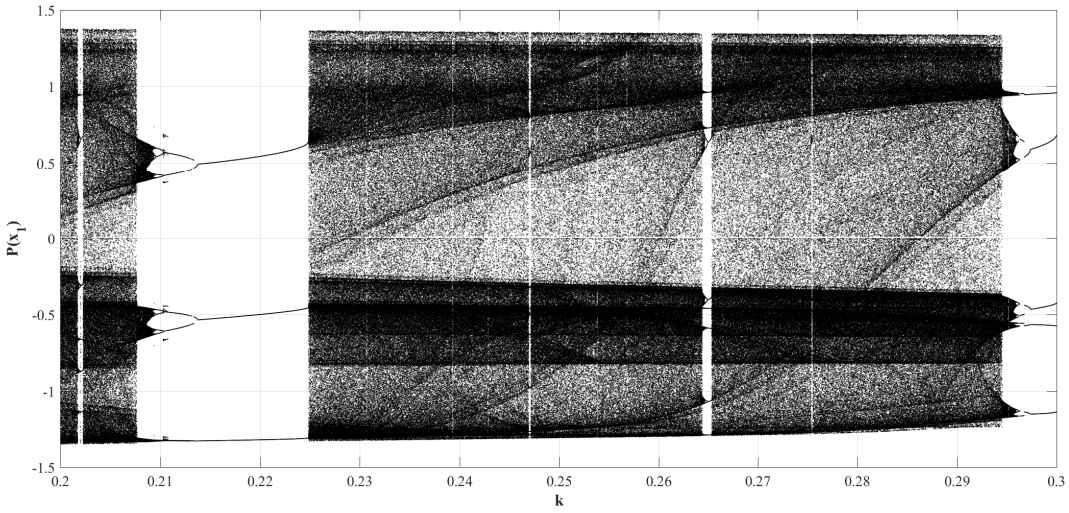

Figure 9: The bifurcation diagram of the Duffing oscillator; that is, the first component of the Poincaré section $P(x_1)$ as a function of the control parameter k. The control parameter is linearly distributed between 0.2 and 0.3 with a resolution of 46080.

12 Details of the pre-declared device functions

The last important step to put together a problem is the definition of the system via the pre-declared user functions. All these functions are introduced in the next subsections on the reference case (the Duffing oscillator), see also in Sec. 7 for a refresh of memory on the main tasks. These functions are found in the file **PerThread_SystemDefinition.cuh**. It is very important that these functions cannot be renamed or replaced into another file; otherwise, the solver algorithms will not be able to find them. If one intends to examine another system, it is advisable to create a completely new folder, copy the necessary files there and rewrite the system file **PerThread_SystemDefinition.cuh**. This is a necessary inconvenience as function pointers cannot be passed to a kernel function (a function running on the GPU and called from the CPU, see Tab. 1). Observe also that the system file is actually a CUDA header file with an extension **.cuh** which is included simply into the file **MassivelyParallel_GPU-ODE_Solver.cu**. This "trick" is important due to performance consideration. In this way, the system file can be separated from the solver file but they actually compiled together in the same module; thus, the compiler has many options

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

Before proceeding further into the details of the pre-declared function, the abbreviated names of the used arguments and their descriptions are summarised in Tab. 9. All the other argument names are self-explanatory; therefore, they are not listed in the table.

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

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

12.1 The right-hand side of the system

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

```
__device__ void PerThread_OdeFunction(int tid, int NT, double* F, double* X, double T,
    double* cPAR, double* sPAR, double* ACC)
{
   int i0 = tid + 0*NT;
   int i1 = tid + 1*NT;

   double x1 = X[i0];
   double x2 = X[i1];

   double p1 = cPAR[i0];
   double p2 = sPAR[0];

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

Observe 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. Although the implementation of the right-hand side is simple (almost as simple as in case of MATLAB), some special care of the indexing is necessary. During the integration phase, each GPU thread calls its own instance of this device function. Therefore, a unique thread identifier (tid) is required to access the proper data of a specific ODE system. With the help of the total number of threads (NT), the proper indices of the state variables X, the ode function F, the control parameters $cPAR$ and the accessories ACC can be generated simply as $tid + i \times NT$, where i is the serial number of a variable (e.g. the state variables or the control parameters). The reason for this kind of indexing is the efficient access to the global memory called coalesced global memory access. It is extremely important to achieve high performance; therefore, it is a necessary inconvenience. Nevertheless, the proper indices of a system can be calculated in advance following the aforementioned simple rule (see the first two code lines inside the function), and then they can be used easily throughout the rest of the function body. Keep in mind again that the indexing starts from 0 in C++.

Since the shared parameters are shared among all threads, they need no special indexing. They can be loaded simply by their serial number. Observe that the total number of control parameters in a system is $N_{CP} \times N_T$ that can cause significant pressure on global memory throughput/bandwidth. On the other hand, the number of the shared parameters are immediately loaded into the so-called shared memory that is a really fast memory type of GPUs (similarly to CPUs, these memories are caches of GPUs). In addition, a single request from a shared memory can be "broadcasted" to every thread further reducing the pressure on memory operations. Therefore, it is advisable to define every parameter common to all systems as shared.

Observe that via the user programmable accessories ACC, the user has great flexibility during the evaluation of the right-hand side. For instance, if a term should be approximated to the first order, it can be stored in an accessory. Then, the related accessory can be updated after every successful time step (see Sec, 12.5) refreshing the first order approximation. Next, it can be used during the evaluation of the right-hand side during the next step. Although such a hybrid "algorithm" is seldom used, there can be cases where it can save a tremendous amount of computations if the proper evaluation of such terms is extremely complicated.

12.2 The properties of the adaptive time-marching

In case of using adaptive numerical schemes, some details of the time step predictions (e.g. error tolerances) have to be specified. In the pre-declared user function

```
__device__ void PerThread_OdeProperties(double* RelativeTolerance, double*
    AbsoluteTolerance, double& MaximumTimeStep, double& MinimumTimeStep, double&
    TimeStepGrowLimit, double& TimeStepShrinkLimit)
{
   RelativeTolerance[0] = 1e-9;
   RelativeTolerance[1] = 1e-9;

   AbsoluteTolerance[0] = 1e-9;
   AbsoluteTolerance[1] = 1e-9;

   MaximumTimeStep     = 1.0e6;
   MinimumTimeStep     = 2.0e-12;
   TimeStepGrowLimit   = 5.0;
   TimeStepShrinkLimit = 0.1;
}
```

these details can be specified. The length of the **RelativeTolerance** and **AbsoluteTolerance** must be the same as the system dimension (N_{SD}). In case of the Duffing reference systems, $N_{SD} = 2$. In this function, the minimum and the maximum possible time step can also be specified. Moreover, the time step growth factor for accepted steps and the shrink factor for rejected step can be controlled as well. In this specific example, the next time step cannot be larger than 5 and cannot be smaller than 0.1 times the previous one.

12.3 The event functions

In the reference calculations, two events are specified as follows

```
__device__ void PerThread_EventFunction(int tid, int NT, double* EF, double* X, double T,
     double* cPAR, double* sPAR, double* ACC)
{
   int i0 = tid + 0*NT;
   int i1 = tid + 1*NT;

   double x1 = X[i0];
   double x2 = X[i1];

   EF[i0] = x2;
   EF[i1] = x1;
}
```

The rule of the indexing is the same as in case of the righ-hand side function. The first event function is $F_{E1} = x_2$ (evaluated into the variable EF[i0]) meaning that a special point is detected if $x_2 = 0$. Since $\dot{x}_2 = x_1$, this means the local maxima or minima of the varible x_1, see Sec. 7. The second event function is $F_{E2} = x1$ (evaluated into the variable EF[i1]); that is, a special point is detected if the value of x_1 is zero (zero displacement). Observe that most of the possible arguments listed in Tab. 9 are passed to this function providing a great fexibility to define an event function. For instance, the number of the succesful time steps can be registered in a user pregrammable accessory variable ACC, then it can be used to specify an event corresponding to a maximum number of time steps. After that, even a stop condition can be specified (see Sec. 12.4).

12.4 The properties of the event functions

Similarly to the properties of the adaptive time-marching, some properties of the event detection are also necessary to specify. The related pre-declared user function (for the reference case) is written as

```
__device__ void PerThread_EventProperties(int* EventDirection, double* EventTolerance,
    int* EventStopCounter, int& MaxStepInsideEvent)
{
   EventDirection[0]    = -1;
   EventDirection[1]    =  0;

   EventTolerance[0]    = 1e-6;
   EventTolerance[1]    = 1e-6;

   EventStopCounter[0] = 0;
   EventStopCounter[1] = 0;

   MaxStepInsideEvent  = 50;
}
```

There must be as many **EventDirection**, **EventTolerance** and **EventStopCounter** as the number of the event function N_{EF}. The **EventDirection** has three possible values: $(0, +1, -1)$. The $+1$ and -1 means detection only if the gradient of the event function at the detected location (zero values of the event function) is positive or negative, respectively. Zero value means detection in both directions. The variable **EventTolerance** gives an absolute tolerance to control the error of the event detection. The absolute value of the event function must be within this tolerance for a successful detection. The **EventStopCounter** controls that after how many successful event detections the integration must stop. It can be any positive integer number or zero. Zero means that the integration should never stop. Specifying a proper **EventStopCounter** is the only way to prevent the program to integrate until the end of the time domain.

In some cases, mainly in autonomous systems, it is possible that a trajectory settles down to a stable fixpoint. If this fixpoint lies in the absolute error band of an event function, that particular trajectory will never leave this event detection zone. Since integrating further is meaningless in this situation, the integration can be stopped after a certain number of successful time steps given by **MaxStepInsideEvent**.

12.5 Action after every succesful event detection

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

```
__device__ void PerThread_ActionAfterEventDetection(int tid, int NT, int IDX, int CNT,
    double &T, double &dT, double* TD, double* X, double* cPAR, double* sPAR, double* ACC
    )
{
   int i0 = tid + 0*NT;
   int i1 = tid + 1*NT;

   double x1 = X[i0];
   double x2 = X[i1];

   if ( x1>ACC[i0] )
      ACC[i0] = x1;

   if ( (IDX==1) && (CNT==2) )
      ACC[i1] = x2;
}
```

Inside this function, many special operations can be performed. For instance, the trajectory can be manipulated or some properties related to an event can be stored into an accessory. In case of the example above, the local maxima (local, as it is detected via an event) of the Duffing system is stored into the first accessory (its serial number is 0 and its index is $i0$) via testing the value of x_1 with the previous value of the related accessory.

In the last part of the code snippet, the value of x_2 is stored into the second accessory (index $i1$) at the second successful detection ($CNT == 2$) of the second event function ($IDX == 1$). Keep in mind again that the indexing starts from 0; therefore, the value of the event index IDX should be 1. This example has no particular physical importance, it is just an example to demonstrate the flexibility and strengths of the implemented event detection mechanism.

A very powerful feature of this pre-declared user function relates to impact dynamics. After the detection of an impact, the corresponding trajectory can be "thrown away" to another location determined by a suitable impact law. This can be done simply by overwriting the variable X, see also Sec. 14.2. 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.

12.6 Action after every successful time step

Similarly to the Sec. 12.5, an action implemented by a control flow inside the following function

```
__device__ void PerThread_ActionAfterSuccessfulTimeStep(int tid, int NT, double T, double
     dT, double* TD, double* X, double* cPAR, double* sPAR, double* ACC)
{
   int i0 = tid + 0*NT;
   int i2 = tid + 2*NT;

   double x1 = X[i0];

   if ( x1>ACC[i2] )
      ACC[i2] = x1;
}
```

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. 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 $i2$). It is global since the accessory shall be overwritten at every time step even if x_1 increases monotonically.

12.7 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

```
__device__ void PerThread_Initialization(int tid, int NT, double T, double &dT, double*
    TD, double* X, double* cPAR, double* sPAR, double* ACC)
{
   int i0 = tid + 0*NT;
   int i1 = tid + 1*NT;
   int i2 = tid + 2*NT;
   int i3 = tid + 3*NT;

   double x1 = X[i0];
   double x2 = X[i1];

   ACC[i0] = x1;
   ACC[i1] = x2;
   ACC[i2] = x1;

   dT = ACC[i3];
}
```

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

line overwrites the time step *dT* specified during the solver configuration, see Sec. 11, with the value of the fourth accessory. Therefore, this accessory has to be properly initialised during the filling of the Problem Pool; otherwise, the initial time step of the very first integration phase would be undetermined. This forth accessory can be properly adjusted during the end of every integration phases as shown in Sec. 12.8.

12.8 Finalisation after every integration phase

The function shown below is called after the end of every integration phase. Again, any kind of control logic can be implemented inside. In this specific example, the time step of the last step is written into the fourth accessory.

```
__device__ void PerThread_Finalization(int tid, int NT, double T, double dT, double* TD,
    double* X, double* cPAR, double* sPAR, double* ACC)
{
   int i3 = tid + 3*NT;

   ACC[i3] = dT;
}
```

13 Performance considerations

For small systems (approximately $N_{SD} \approx 10 - 20$), with the "default" setup introduced via the reference tutorial example, the performance will likely suffer **no** harm. The only issue the user has to take care about is to set the **BlockSize** to an integer multiple of 32, and set a sufficiently large number of threads in order to fully utilise the GPU arithmetic processing units (approximately $N_T > 10000$).

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

13.1 Threading in GPUs

The basic logical unit performing calculations is a thread. The number of threads simultaneously reside in a GPU can be in the order of hundreds of millions. This is the reason for the widely used term: massively parallel programming. In general, threads in a GPU are organised in a 3D structure called *grid*. However, for our purpose, a 1D organisation is sufficient. That is, a unique identifier of a thread can be characterised by a 1D integer coordinate. The total number of threads N_T are divided into thread blocks. The variable **BlockSize** introduced in Sec. 11 defines the number of threads that can reside in a single block. In the reference case, the number of threads is $N_T = 46080$ and the block size is $N_{BS} = 64$; that is, the number of the blocks is $N_B = N_T/N_{BS} = 720$.

13.2 The parallelisation strategy

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

13.3 The main building block of a GPU architecture

Each GPU consists of one or more Streaming Multiprocessors (SMs) which are at the highest level of hierarchy in the hardware compute architecture. Each SM contains a number of processing units capable of performing floating point operations, load/store operations from the Global Memory or from the Shared Memory, control flow operations or other specialised instructions. The workload in a GPU is distributed to SMs with block granularity. That is, the block scheduler of the GPU (Giga Thread Engine) fills every SM with blocks until reaching hardware or resource limitation, for the details see Secs. 13.5 adn 13.7. The maximum number of the residing threads in an SM is dependent on the compute capability of the hardware. Unfortunately, there is no CUDA API to query this information, the user have to retrieve this number from the CUDA Programming Guide [1] according the Compute Capability of the hardware. For the Kepler architecture used during the reference calculations (Compute Capability 3.5, see also Sec. 6), the maximum number of blocks residing in an SM simultaneously is 16. For the same architecture, the number of the SMs are 15, see again Sec. 6 on how to query this information. This means that the total number of blocks $N_B = 720$ cannot be distributed to the SMs all at the same time. The strategy of the Giga Thread Engine is that if computation of a block is finished in an SM, its computational resources are freed and a new block is immediately assigned to the SM. This cycle continues until all the block have been processed. This distribution technique has a consequence that blocks can be processed in any order; thus, the user should not write a code assuming a specific order of block assignment. Processing of blocks is independent of each other. The program package MPGOS fulfils this requirement as every thread is inherently independent of each other.

The above discussion has an impact on the performance: the total number of blocks should be an integer multiple of the number of the SMs. In this way, the total computational workload can be distributed evenly between the SMs, assuming that the time required to process the blocks are nearly the same. Otherwise, during the last phase of the computation, some SMs shall be idle. This phenomenon is called tailing effect. In the reference case, the total number of the blocks during one simulation launch is $N_B = N_T/N_{t/n} = 23040/64 = 360$, where $N_{t/b} = 64$ is the number of threads in a block (the block size), see the code snippet of the full code in Sec. 11. Thus, the number of blocks assigned to an SM during a simulation launch is $360/15 = 24$. Naturally, if the total number of the blocks during a run is very high, the tailing effect is minimal. In summary, the number of threads N_T, the block size $N_{t/b}$ and the number of the SMs together determine the tailing effect.

13.4 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. The warp schedulers of an SM take warps and assign them to execution units one after another until there are eligible warps for executions or there are free execution units on the SM. A warp is eligible if there is no data dependency from another computational phase or all the required data has already arrived from a memory load operation.

Every thread in a warp perform the same instruction but on different data. This technique is known as single instruction multiple data paradigm (SIMD). Therefore, every thread in a warp executes their series of instructions in order. This is the only way the hardware can efficiently handle a massive number of threads, as for 32 number of threads only one control unit is necessary to track their program state. This results in more place for compute units and more arithmetic throughput.

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

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

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

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

13.5 Hardware limitations for threads, blocks and warps

There are several hardware limitations on how many threads, blocks and warps can reside simultaneously in an SM. There are also limitations on how many threads can reside in a single block, and on the maximum dimensions of the grid of blocks and block of threads. Keep in mind that the block and grid dimensions are 3D structures; however, for our purpose, a linear 1D organisation is sufficient. Thus, only the first component of their dimensions are relevant for the program package MPGOS. Most of these hardware restrictions can be queried by the built-in function `ListCUDADevices();`, see again Sec. 6. The only exception is the number of blocks that can simultaneously reside in an SM discussed already in Sec. 13.3. The information has to be obtained from the CUDA Programming Guide [2].

13.6 The memory hierarchy

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

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

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

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

13.6.1 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 memory transactions [3]. This is the smallest amount of data that can be delivered in a single request. Therefore, in order to fully utilise the memory bus, threads in a warp during a global memory load/store operation should access consecutive memory locations. For instance, if each thread in a warp require 4 bytes (e.g. a float), it can be delivered via a single 128 byte memory transactions, and memory bus utilisation is 100 %. In any other cases, the number of memory transactions are increased, and the global memory load/store efficiency decreases. Therefore, such coalesced memory accesses are mandatory for highly efficient code.

Fortunately, the users do not have to worry about coalesced accesses. The program package is already written to access global memory in an optimal, coalesced way. The only consequence of the data access requirements is the special indexing behaviour inside the pre-declared user functions discussed in Sec. 12.1. Remember that every variable have to be accessed similarly as

```
int i0 = tid + 0*NT;
int i1 = tid + 1*NT;

double x1 = X[i0];
double x2 = X[i1];

double p1 = cPAR[i0];
double p2 = sPAR[0];

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

where the variable **tid** is a unique identifier of the threads during a run. Keep in mind that a single thread is assigned to an instance of a system Eq. (1). In a warp, threads have a consecutive identifier. For instance: $0 \cdots 31$, $32 \cdots 63$ or $128 \cdots 159$. Therefore, indices **i0** and **i1** are pointing to consecutive memory locations as well. As a consequence, variables **x1**, **x2** and **p1** are also loaded from a consecutive memory locations; that is, the memory load operation is coalesced. Memory store operation corresponding to the variable **F** follows the same rule. The above described global memory access requirement explains the reason of this "special" indexing technique.

13.6.2 Shared memory

Shared memories of the GPUs are on-chip, low latency and high-bandwidth memory types. It is on-chip as every SM has a certain amount of its own shared memory. The total shared memory/SM is an architectural property, and it can be listed with the function call `ListCUDADevices();`, see Sec. 6. Shared memory can be allocated by block granularity; that is, each block has its own amount of allocated shared memory. All threads within a block can "see" the content of the corresponding shared memory. In this way, threads in a block can cooperate with each other. It is important to note that threads within different blocks cannot cooperate as they cannot access the shared memory of another block. Remember that the execution order of the blocks and the block assignment to SMs are not deterministic, see Sec. 13.3. Thus, one cannot rely on the cooperation between threads residing in different blocks.

The program package MPGOS offers the possibility to exploit the advantage of shared memory by putting parameters common to all threads into the shared memory, for the technical details see Secs. 9 and 10. The main advantage is that the expensive (in time) global memory load operations can be minimised with the proper use of the shared memory. For the reference tutorial example, this technique does not have a significant impact as the number of the shared parameters is only one, which could have been hard-coded into the right-hand side evaluation.

However, there can be situations where shared memories can have a significant impact on performance. A perfect example if the evaluation of the right-hand side involves matrix-vector multiplication where the

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

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

Some other examples for the possible use of shared memories: coupled systems where the **coupling marix** is the same for all systems; the right-hand side of the system is very complicated and contains **many pre-computed coefficients**, and some of them can be shared among all systems, for an example see Sec. 14.1.

13.6.3 Registers

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

```
int i0 = tid + 0*NT;
int i1 = tid + 1*NT;

double x1 = X[i0];
double x2 = X[i1];

double p1 = cPAR[i0];
double p2 = sPAR[0];

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

the variables **`i0`**, **`i1`**, **`x1`**, **`x2`** and **`p1`** are put into the registers. Keep in mind, however, that registers are scarce resources. Allocating large arrays or structures inside a pre-declared user function (e.g. in the right-hand side evaluation), the compiler will automatically place them into the slow global memory if there is not enough available registers. This phenomenon is called register spilling done "behind the scenes". An optimising compiler also allocates some amount of registers for intermediate storages to accelerate computations. The total required registers depends on the systems complexity and usage of transcendental functions like sin, log, division or power. Compiling the code with the option **`--ptxas-options=-v`**, the compiler will write information about the required registers for a kernel function:

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

Here, the kernel function **PerThread_RKCK45_EH0** (the Runge–Kutta–Cash–Karp solver without event handling) requires 175 number of 32-bit registers for each thread. Letting the compiler use as many registers as it requires may not be feasible. If a thread needs a large number of registers than the total number of threads residing in an SM can be significantly reduced, for details see Sec. 13.7. This can have a significant impact on the performance as a small number of residing threads have less opportunity to hide the high Global Memory latency, see again the introduction of Sec. 13.6.

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

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

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.

13.7 Resource limitations and occupancy

The number of simultaneously residing threads and blocks in an SM is not limited only by the hardware restrictions (Sec, 13.5) but also by their resource usage (registers and shared memory). If the maximum number of registers per SM is denoted by R_{SM} and the number of allocated registers per thread is R_T, the maximum number of threads can simultaneously reside in the SM is $N_{TSM} = R_{SM}/R_T$. For instance, in our Titan Black card $R_{SM} = 65536$ and $R_T = 64$ (compiler option); the maximum number of residing threads in this case are $N_{TSM} = 1024$. The hardware limitation is 2024 (see Sec. 6); that is, only with $R_T = 32$ can the hardware limitation be saturated. Therefore, setting the compiler option for register usage smaller than 32 is meaningless.

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

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

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

[4] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications
[5] https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

of the compute capability of the hardware, used registers and allocated shared memory. The latest version can also be found in the GitHub repository of the program package MPGOS.

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

The built-in function

```
|| CheckStorageRequirements(ConfigurationKellerMiksis, SelectedDevice);
```

can help to check the resource limitations as a function of the configuration parameters and the device number.

13.8 Maximising instruction throughput

As a general rule, floating point and integer addition/multiplication can be performed within one clock cycle; that is, these operations are fast. However, there are really expensive arithmetic computations such as the usage of any kind of **trascendental functions** (e.g. sine, cosine, logarithm etc.), integer or floating point **division** and the commonly used **power** function `pow()`. These operations have much larger latencies and need many clock cycles to perform the instruction. Moreover, they require a large number of registers during their intermediate computations. Therefore, the user should minimise their usage as much as possible:

- If the reciprocal of a variable is necessary more than one times, it is best to calculate only once in an intermediate variable and then apply multiplication successively.
- Put such expensive operations inside a loop only if it is really necessary. Otherwise, the performance of the code will perish.
- Do not use the function `pow()` unless it is the only possible option to perform the calculation. For integer power, use successive multiplications: instead of `pow(x,2)` use `x*x`. For fractional power, try to combine division, square root function `sqrt()`, reciprocal square root function `rsqrt()` and their cubic counterpart [6]. For instance, instead of `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 have 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.

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

13.9 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
```

The user has to replace his/her own executable and the output file name inside the script. It has three phases. First, an aggregated statistics is written about the used kernel functions. Second, an output file is created for visual profiler ***.nvprof**. Third, the most important event and metrics are listed about the related kernel functions. The simplified output of the third phase of the profiling is listed below for the reference tutorial example

The 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 for the reference case is 0.5. Therefore, the 0.42 achieved occupancy is considered good.
- Global Store Throughput and Global Load Throughput: if the sum of these throughput values (203 Gb/s) are close the theoretical bandwidth (for our Titan Black card 336 Gb/s), the code is very like to be limited by memory bandwidth. In this case, the Global Memory load/store transactions should be reduced; for instance, via more extensive usage of the shared memory (shared parameters). The reference tutorial example is not bound by memory bandwidth.
- Eligible Warps Per Active Cycle: the average number of eligible warps in an active cycle. A warp is eligible if it is ready for computations; that is, all the data required for the next calculation have already arrived from the Global Memory. This value should be above the number warp schedulers of an SM. In the Titan Black card, it is 4.
- Arithmetic Function Unit Utilization: it measures the arithmetic function utilisation of an SM in a scale between 0 and 10. This is the most important metrics, as one would like to utilise the full processing power of a GPU. As we can see, it is already at its maximum; therefore, the kernel of the reference example 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.

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%

14 MPGOS tutorial examples

This section serves as an additional material to highlight the flexibility and features of the program package MPGOS. The list of the tutorial examples will be continuously extended.

14.1 Tutorial 1: quasiperiodic forcing

The model of the first 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

$$\left(1-\frac{\dot{R}}{c_L}\right)R\ddot{R}+\left(1-\frac{\dot{R}}{3c_L}\right)\frac{3}{2}\dot{R}^2=\left(1+\frac{\dot{R}}{c_L}+\frac{R}{c_L}\frac{d}{dt}\right)\frac{(p_L-p_\infty(t))}{\rho_L}, \tag{5}$$

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

$$p_\infty(t) = P_\infty + P_{A1}\sin(\omega_1 t) + P_{A2}\sin(\omega_2 t+\theta), \tag{6}$$

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

$$p_G + p_V = p_L + \frac{2\sigma}{R} + 4\mu_L\frac{\dot{R}}{R}, \tag{7}$$

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

$$p_G = \left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\left(\frac{R_E}{R}\right)^{3\gamma}, \tag{8}$$

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

System (5)-(8) is rewritten into a dimensionless form defined as

$$\dot{y}_1 = y_2, \tag{9}$$

$$\dot{y}_2 = \frac{N_{\mathrm{KM}}}{D_{\mathrm{KM}}}, \tag{10}$$

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

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

and

$$D_{\mathrm{KM}} = y_1 - C_9 y_1 y_2 + C_4 C_9. \quad (12)$$

The coefficients are summarised as follows:

$$C_0 = \frac{1}{\rho_L}\left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\left(\frac{2\pi}{R_E\omega_1}\right)^2, \quad (13)$$

$$C_1 = \frac{1 - 3\gamma}{\rho_L c_L}\left(P_\infty - p_V + \frac{2\sigma}{R_E}\right)\frac{2\pi}{R_E\omega_1}, \quad (14)$$

$$C_2 = \frac{P_\infty - p_V}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \quad (15)$$

$$C_3 = \frac{2\sigma}{\rho_L R_E}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \quad (16)$$

$$C_4 = \frac{4\mu_L}{\rho_L R_E^2}\frac{2\pi}{\omega_1}, \quad (17)$$

$$C_5 = \frac{P_{A1}}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \quad (18)$$

$$C_6 = \frac{P_{A2}}{\rho_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \quad (19)$$

$$C_7 = R_E\frac{\omega_1 P_{A1}}{\rho_L c_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \quad (20)$$

$$C_8 = R_E\frac{\omega_1 P_{A2}}{\rho_L c_L}\left(\frac{2\pi}{R_E\omega_1}\right)^2, \quad (21)$$

$$C_9 = \frac{R_E\omega_1}{2\pi c_L}, \quad (22)$$

$$C_{10} = 3\gamma, \quad (23)$$

$$C_{11} = \frac{\omega_2}{\omega_1}, \quad (24)$$

$$C_{12} = \theta. \quad (25)$$

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 Problem Pool.

14.1.1 The objectives

This tutorial example calculates the collapse strength of the air filled single spherical bubble placed in liquid water and subjected to dual-frequency ultrasonic irradiation. An example for the radial oscillation of a bubble is demonstrated in Fig. 10, 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. 10. Observe that the time scales can be very different near the local maximum and the local minimum.

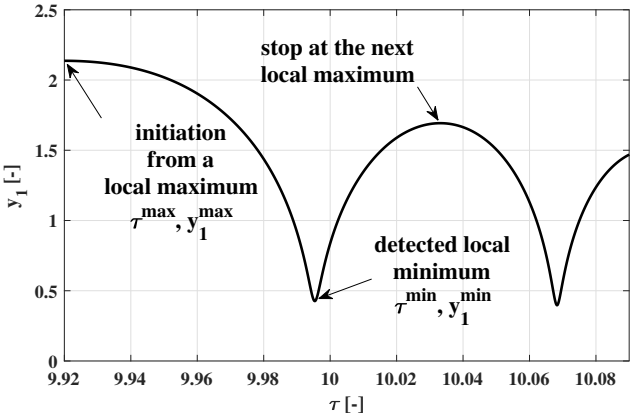

Figure 10: Demonstration of a bubble collapse via a dimensionless bubble radius vs. time curve. An integration start from a local maximum y_1^{max} at time instant τ^{max} and ends at the next local maximum. During the integration, the local minimum y_1^{min} and its time instant τ^{min} is also determined.

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. 10. Observe that the number of the investigated parameters of each pressure amplitude is only two (only the minimum and the maximum). The overall number of scanned parameters is $2 \times 2 \times 128 \times 128 = 65536$. In each simulation, the first 1024 iteration (collapses) are regarded as initial transients and discarded. The next 64 collapses are used to collect data that is written into a file.

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

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

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

14.1.2 Configuration of the Problem Pool and the Solver Object

The configuration of the Problem Pool and the Solver Object is as follows:

```
int NumberOfFrequency1 = 128;
int NumberOfFrequency2 = 128;
int NumberOfAmplitude1 = 2;
int NumberOfAmplitude2 = 2;

int PoolSize = NumberOfFrequency1 * NumberOfFrequency2 * NumberOfAmplitude1 *
    NumberOfAmplitude2;
int NumberOfThreads = NumberOfFrequency1 * NumberOfFrequency2;

ConstructorConfiguration ConfigurationKellerMiksis;

ConfigurationKellerMiksis.PoolSize                  = PoolSize;
ConfigurationKellerMiksis.NumberOfThreads           = NumberOfThreads;
ConfigurationKellerMiksis.SystemDimension           = 2;
ConfigurationKellerMiksis.NumberOfControlParameters = 21;
ConfigurationKellerMiksis.NumberOfSharedParameters  = 0;
ConfigurationKellerMiksis.NumberOfEvents            = 1;
ConfigurationKellerMiksis.NumberOfAccessories       = 4;

ProblemSolver ScanKellerMiksis(ConfigurationKellerMiksis);
ProblemPool ProblemPoolKellerMiksis(ConfigurationKellerMiksis);
```

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

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

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

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

14.1.3 The pre-declared user functions of the system

In this section, only some of the pre-declared user functions are shown, which provides new insight into the implementation possibilities. The complete system definition file can be found in the corresponding GitHub repository [7].

The implemented right-hand side is as follows

```
__device__ void PerThread_OdeFunction(int tid, int NT, double* F, double* X, double T,
    double* cPAR, double* sPAR, double* ACC)
{
   double x1 = X[tid + 0*NT];
   double x2 = X[tid + 1*NT];

   double p0  = cPAR[tid +  0*NT];
   double p1  = cPAR[tid +  1*NT];
   double p2  = cPAR[tid +  2*NT];
   double p3  = cPAR[tid +  3*NT];
   double p4  = cPAR[tid +  4*NT];
   double p5  = cPAR[tid +  5*NT];
   double p6  = cPAR[tid +  6*NT];
   double p7  = cPAR[tid +  7*NT];
   double p8  = cPAR[tid +  8*NT];
   double p9  = cPAR[tid +  9*NT];
   double p10 = cPAR[tid + 10*NT];
   double p11 = cPAR[tid + 11*NT];
   double p12 = cPAR[tid + 12*NT];

   double rx1 = 1.0/x1;
   double p   = pow(rx1, p10);

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

   double s2 = sin(2.0*p11*PI*T+p12);
   double c2 = cos(2.0*p11*PI*T+p12);

   double N;
   double D;
   double rD;

   N = (p0+p1*x2)*p - p2*(1.0+p9*x2) - p3*rx1 - p4*x2*rx1 - 1.5*(1.0-p9*x2/3.0)*x2*x2 - (
        p5*s1 + p6*s2 ) * (1.0+p9*x2) - x1*( p7*c1 + p8*c2 );
   D = x1 - p9*x1*x2 + p4*p9;
   rD = 1.0/D;

   F[tid + 0*NT] = x2;
   F[tid + 1*NT] = N*rD;
}
```

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

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

```
__device__ void PerThread_Initialization(int tid, int NT, double T, double &dT, double*
    TD, double* X, double* cPAR, double* sPAR, double* ACC)
{
   double x1 = X[tid + 0*NT];
   ACC[tid + 0*NT] = x1;
   ACC[tid + 1*NT] = T;
   ACC[tid + 2*NT] = x1;
   ACC[tid + 3*NT] = T;
}
```

[7] https://github.com/FerencHegedus/Massively-Parallel-GPU-ODE-Solver/tree/master/PerThread/Tutorials/T1_Quasi-periodicForcing

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

```
__device__ void PerThread_ActionAfterSuccessfulTimeStep(int tid, int NT, double T, double
    dT, double* TD, double* X, double* cPAR, double* sPAR, double* ACC)
{
   double x1 = X[tid + 0*NT];

   if ( x1<ACC[tid + 2*NT] )
   {
      ACC[tid + 2*NT] = x1;
      ACC[tid + 3*NT] = T;
   }
}
```

In general, the dual-frequency excitation can be quasiperiodic (or at least nearly quasiperiodic due to the finite resolution of the frequencies). Therefore, the proper track of the time domain is crucial. The end of the time domain is set to a very high number; namely, 10^{10}, the simulation will be stopped by the event. At the end of each integration phase, the beginning of the time domain must be properly adjusted as follows

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

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

14.1.4 Results

In Fig. 11, the relative expansion ratio y_{exp} is presented via four bi-parametric contour plots. The colour code indicates the magnitude of the relative expansion saturated at $y_{exp} = 5$. The higher the value of y_{exp} the stronger the bubble collapse (red domains). The control parameters in each subplot are the excitation frequencies ω_1 and ω_2. The pressure amplitude combinations are highlighted in the labels of the axes. At a given parameter set, the largest value out of the 64 stored relative expansions is depicted. The total simulation time of the 65536 number of Keller–Miksis equation is 8.5 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. 11). The employed numerical scheme was the Runge–Kutta–Kash–Carp. Both the absolute and relative tolerances were 10^{-10}.

14.2 Tutorial 2: impact dynamics

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

$$\dot{y}_1 = y_2, \tag{26}$$

$$\dot{y}_2 = -\kappa y_2 - (y_1 + \delta) + y_3, \tag{27}$$

$$\dot{y}_3 = \beta(q - y_1\sqrt{y_3}), \tag{28}$$

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

In Eqs (26)-(28), 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

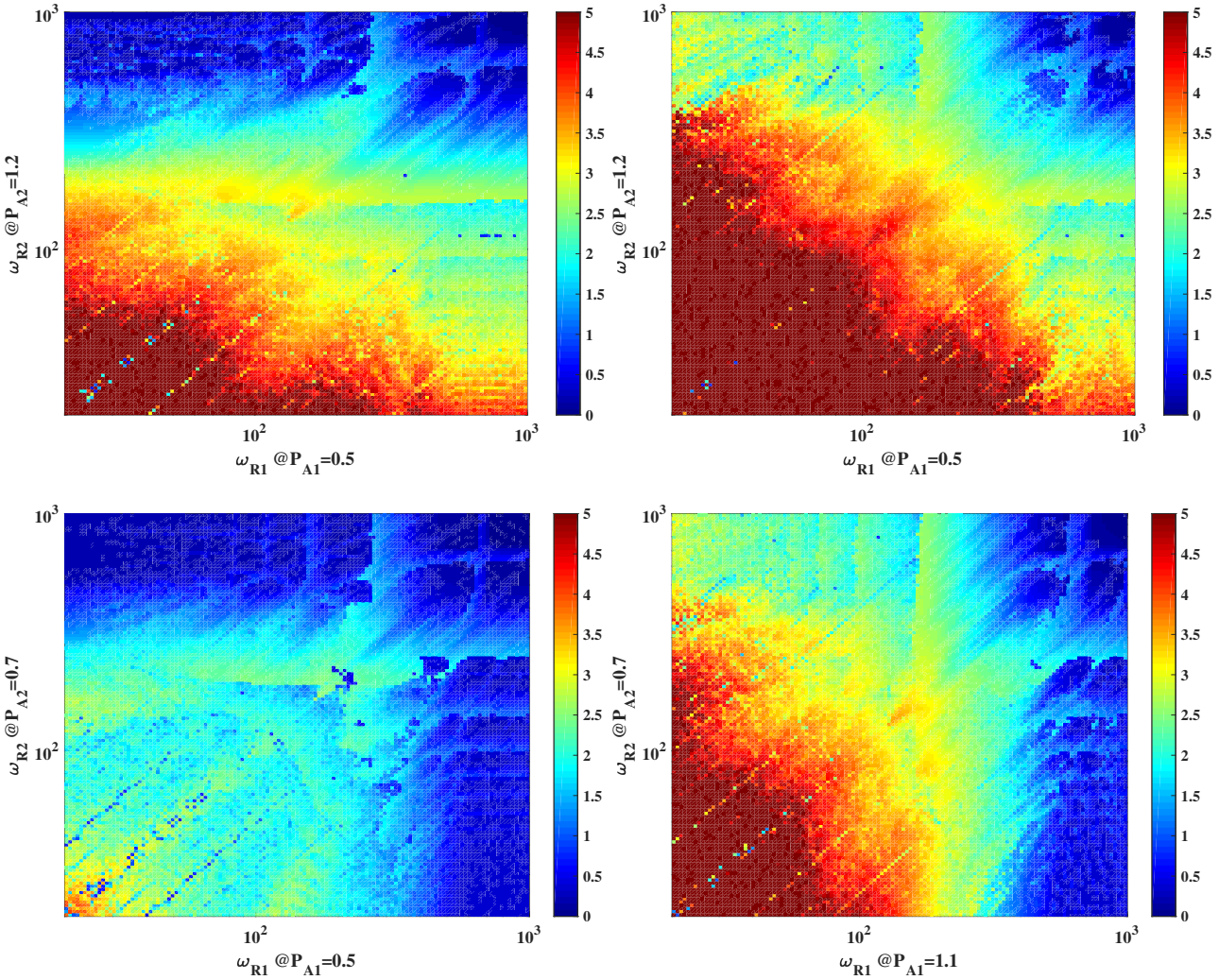

Figure 11: Four dimensional parameter scan of the relative expansion of an oscillating single spherical bubble.

following impact law

$$y_1^+ = y_1^- = 0, \tag{29}$$
$$y_2^+ = -r y_2^-, \tag{30}$$
$$y_3^+ = y_3^- \tag{31}$$

is applied. The Newtonian coefficient of restitution $r = 0.8$ approximates the loss of energy of the impact. It shall be shown that by applying multiple event handling together with a special "action function" call at the impact detection, system (26)-(28) can be handled very efficiently.

14.2.1 The objectives

The objective of this tutorial example is very simple, generate a bifurcation diagram, where the maximum and the minimum displacement of the valve body y_1^{max} and y_1^{min} is plotted as a function of the low rate q. Since the system is autonomous, an event handling is necessary to detect the Poincaré section defined as the local maxima of y_1 at $y_2^+ = 0$. Moreover, another event handling is necessary to handle the impact at $y_1 = 0$. One integration phase means the integration of the systems from Poincaré section to Poincaré section. The

sole control parameter is the flow rate q varied between 0.2 and 10 with a resolution of 30720. In case of impact, the minimum value of the displacement will contain the value of $y_1^{min} = 0$. In each simulation, the first 1024 iterations are regarded as initial transients and discarded. The data of the next 32 iterations are recorded and written into a text file. The first, second and third columns in the file are related to the control parameter q, y_1^{max} and y_1^{min}, respectively.

14.2.2 Configuration of the Problem Pool and the Solver Object

The configuration of the Problem Pool and the Solver Object is as follows:

```
int NumberOfFlowRates = 30720;

int PoolSize        = NumberOfFlowRates;
int NumberOfThreads = NumberOfFlowRates;

ConstructorConfiguration ConfigurationPressureReliefValve;

ConfigurationPressureReliefValve.PoolSize                  = PoolSize;
ConfigurationPressureReliefValve.NumberOfThreads           = NumberOfThreads;
ConfigurationPressureReliefValve.SystemDimension           = 3;
ConfigurationPressureReliefValve.NumberOfControlParameters = 1;
ConfigurationPressureReliefValve.NumberOfSharedParameters  = 4;
ConfigurationPressureReliefValve.NumberOfEvents            = 2;
ConfigurationPressureReliefValve.NumberOfAccessories       = 2;

ProblemSolver ScanPressureReliefValve(ConfigurationPressureReliefValve);
ProblemPool ProblemPoolPressureReliefValve(ConfigurationPressureReliefValve);
```

Here the size of the Problem Pool and the number of threads in the Solver Object are equal (the resolution of the control parameter q). Therefore, there is only one simulation launch. The system dimension is $N_{SD} = 3$ according to the model of the pressure relief valve. The number of the control parameters is only $N_{CP} = 1$ (flow rate q). However, there are 4 shared parameters: κ, δ, β and r. Since the right-hand side evaluation is much less arithmetic operation intensive compared to the first tutorial example in Sec. 14.1, defining these four parameters as shared can ease the pressure on Global Memory load operations significantly. Still, profiling the code one can experience a total of 332 Gb memory throughput that is very close to the theoretical peak bandwidth of 336 Gb. Nevertheless, the arithmetic function utilisation is maximum: 10. Thus the code is still very efficient. For the computations, two event functions are needed. One for defining a Poincaré section and one for detecting the impact. Finally, two user programmable parameters (accessories) are used to store the values of y_1^{max} and y_1^{min}.

14.2.3 The pre-declared user functions of the system

In this section, only some of the pre-declared user functions are shown, which provides new insight into the implementation possibilities. The complete system definition file can be found in the corresponding GitHub repository [8].

The implemented right-hand side is as follows

```
__device__ void PerThread_OdeFunction(int tid, int NT, double* F, double* X, double T,
    double* cPAR, double* sPAR, double* ACC)
{
   int i0 = tid + 0*NT;
   int i1 = tid + 1*NT;
   int i2 = tid + 2*NT;

   double x1 = X[i0];
   double x2 = X[i1];
   double x3 = X[i2];
```

[8] https://github.com/FerencHegedus/Massively-Parallel-GPU-ODE-Solver/tree/master/PerThread/Tutorials/T2_Impact-Dynamics

```
    double p1 = sPAR[0];
    double p2 = sPAR[1];
    double p3 = sPAR[2];

    double p4 = cPAR[i0];

    F[i0] = x2;
    F[i1] = -p1*x2 - (x1+p2) + x3;
    F[i2] = p3*( p4 - x1*sqrt(x3) );
}
```

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

The definition of the two event functions are very straightforward:

```
__device__ void PerThread_EventFunction(int tid, int NT, double* EF, double* X, double T,
     double* cPAR, double* sPAR, double* ACC)
{
    int i0 = tid + 0*NT;
    int i1 = tid + 1*NT;

    double x1 = X[i0];
    double x2 = X[i1];

    EF[i0] = x2; // Poincare section
    EF[i1] = x1; // Impact detection
}
```

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

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

```
__device__ void PerThread_ActionAfterEventDetection(int tid, int NT, int IDX, int CNT,
    double &T, double &dT, double* TD, double* X, double* cPAR, double* sPAR, double* ACC
    )
{
    int i1 = tid + 1*NT;

    double p5 = sPAR[3];

    if ( IDX == 1 )
       X[i1] = -p5 * X[i1];
}
```

Observe how the velocity y_2 is reversed by applying the impact law Eq. (30). The coefficient of restitution is load from the Shared Memory.

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

14.2.4 Results

The maxima y_1^{max} (black dots) and minima y_1^{min} (red dots) of the displacement of the valve body is depicted in Fig. 12 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 42 s (writing the data into the text file is again included). The employed numerical scheme was the Runge–Kutta–Kash–Carp. Both the absolute and relative tolerances were 10^{-10}.

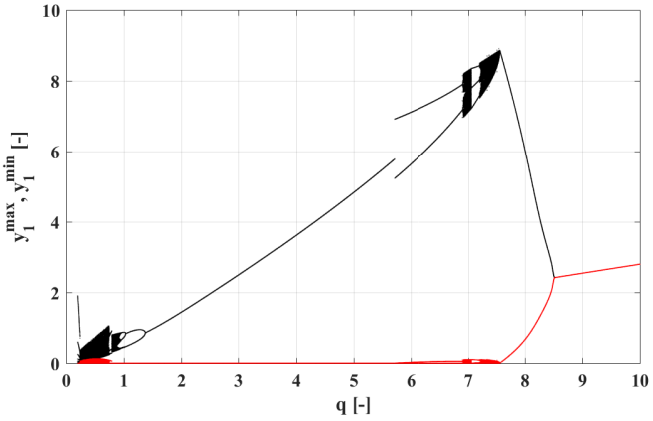

Figure 12: The maximum (black) and minimum (red) values of the valve position y_1 as a function of the dimensionless flow rate q for the pressure relief valve. The damping coefficient is $\kappa = 1.25$, the precompression parameter is $\delta = 10$ and the compressibility parameter is $\beta = 20$. The number of the used threads and the resolution of the control parameter is $N_T = 30720$.

14.3 Tutorial 3: overlap CPU and GPU computations (double buffering)

So far in the main part of the manual (the reference example) and in the first two tutorial examples, all the GPU related operations (a member function call of the Solver Object) were synchronous. This means that performing such an operation, the CPU thread will be idle until the dispatched task is completely finished on the GPU. Due to this behaviour, the simple `Solve()` member function immediately synchronises the new data back to the CPU side. Observe that this behaviour do not allow the usage of multiple GPUs a the CPU thread should dispatch task to the GPUs one after another without delay, without waiting for a GPU to finish its work. Otherwise the tasks performed on different GPUs will be serialised.

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

Nevertheless, in order to clearly understand the behaviour of the asynchronous member functions, 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 like in the case of the simple `Solve()` member function).

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

To each Solver Object, a device number is immediately associated during its declaration, see Sec. 10. Similarly, a CUDA stream is also created and associated to a Solver Object inside its constructor. Any GPU

related actions via its member function will use its own CUDA stream to queue the tasks to the associated GPU. The corresponding member functions of the Solver Object to put tasks asynchronously into its CUDA stream is summarised in Tab. 11.

14.3.1 The objectives and the workflow

The main objective of this tutorial example is to revise the reference example and try to overlap as many computations between the CPU and GPU as possible. Keep in mind that the total number of problems need to be solved is 46080 which was divided into two launches with $N_T = 23040$ using a single GPU and a single Solver Object. Here, the usage of Problem Pool is omitted, and two Solver Objects are employed for overlapping computations (the variable **`PoolSize`** is used only to indicate the total number instances of the ODE intend to be solved). The data storages of the Solver Objects are filled up with aid of the member functions summarised in Tabs. 6 and 7. The fill up procedure is wrapped into tutorial specific functions, for details see the code in **`DoubleBuffering.cu`** in the folder of the third tutorial example. Since the system definition is identical with the reference case, its discussion is omitted here.

First, the two Solver Objects are created as follows:

```
int PoolSize        = 46080;
int NumberOfThreads = 23040;

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

ConstructorConfiguration ConfigurationDuffing;
   ConfigurationDuffing.PoolSize                  = PoolSize;
   ConfigurationDuffing.NumberOfThreads           = NumberOfThreads;
   ConfigurationDuffing.SystemDimension           = 2;
   ConfigurationDuffing.NumberOfControlParameters = 1;
   ConfigurationDuffing.NumberOfSharedParameters  = 1;
   ConfigurationDuffing.NumberOfEvents            = 2;
   ConfigurationDuffing.NumberOfAccessories       = 4;

ProblemSolver ScanDuffing1(ConfigurationDuffing, SelectedDevice);
ProblemSolver ScanDuffing2(ConfigurationDuffing, SelectedDevice);
}
```

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

Next, the shared parameters are filled up in an asynchronous manner:

```
FillSharedParameters(ScanDuffing1,Parameters_B);
ScanDuffing1.SynchroniseSharedFromHostToDeviceAsync();

FillSharedParameters(ScanDuffing2,Parameters_B);
ScanDuffing2.SynchroniseSharedFromHostToDeviceAsync();
}
```

The function `FillSharedParameters(args)` is a tutorial specific function served to set up the values of the shared parameters. In the first line of this code snippet, the CPU fills the first Solver Object (ScanDuffing1) with the shared parameters of the CPU side (System Memory). It is purely a work done by the CPU. In the second line, the shared parameters are copied to the Device Memory of the selected GPU. It is done asynchronously; thus, the control is immediately passed back to the CPU and can fill up the second Solver Object (third row, ScanDuffing2) in parallel with the copy process related to the first Solver Object. Finally, the last row initiates the copy process for the second Solver Object, again asynchronously. Thus the CPU can proceed further immediately.

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

Table 11: Summary of the asynchronous member functions of the class **SolverObject**. After dispatching these tasks, the control immediately returns to the CPU.

Function	Arguments	Description
void SolveAsync(*args*)	**SolverConfiguration** **SolverConfigurationSystem**	Asynchronous dispatch of an integration phase to a GPU. There is no automatic synchronisation of the computed data between the GPU and CPU sides after the completion of the integration.
void SynchroniseFromHost ToDeviceAsync(*args*)	**VariableSelection** **ActualVariable**	Synchronises the selected variable from the System Memory (Host side) into the Global Memory (Device side). Here, the **All** option is valid as an input argument.
void SynchroniseFrom DeviceToHost Async(*arg*)	**VariableSelection** **ActualVariable**	Synchronises the selected variable from the Global Memory (Device side) into the System Memory (Host side). The **All** option is valid.
void SynchroniseShared FromHostToDevice Async()		Synchronises all the shared parameters from the system memory (Host side) into the global memory (Device side).
void SynchroniseShared FromDeviceToHost Async()		Synchronises all the shared parameters from the global memory (Device side) into the system memory (Host side). This member function is seldom used in the present version of the code, as the shared parameters do not change during an integration phase.
void Insert Synchronisation Point()		Insert a synchronisation point into the CUDA stream of the corresponding Solver Object.
void SynchroniseSolver()		Synchronises the CUDA stream with respect to the calling CPU thread according to the synchronisation point inserted by the previous member function.
void SynchroniseDevice()		Synchronises the associated GPU (even if there are multiple streams from multiple Solver Object) with respect to the calling CPU thread.

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

int FirstProblemNumber;

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

   FillControlParameters(ScanDuffing1, Parameters_k_Values, InitialConditions_X1,
       InitialConditions_X2, FirstProblemNumber, NumberOfThreads);
   ScanDuffing1.SynchroniseFromHostToDeviceAsync(All);
   ScanDuffing1.SolveAsync(SolverConfigurationSystem);
   ScanDuffing1.InsertSynchronisationPoint();

   FirstProblemNumber = FirstProblemNumber + NumberOfThreads;

   FillControlParameters(ScanDuffing2, Parameters_k_Values, InitialConditions_X1,
       InitialConditions_X2, FirstProblemNumber, NumberOfThreads);
   ScanDuffing2.SynchroniseFromHostToDeviceAsync(All);
   ScanDuffing2.SolveAsync(SolverConfigurationSystem);
   ScanDuffing2.InsertSynchronisationPoint();

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

      ScanDuffing2.SynchroniseSolver();
      ScanDuffing2.SolveAsync(SolverConfigurationSystem);
      ScanDuffing2.InsertSynchronisationPoint();
   }

   for (int i=0; i<31; i++)
   {
      ScanDuffing1.SynchroniseSolver();
      ScanDuffing1.SynchroniseFromDeviceToHostAsync(All);
      ScanDuffing1.SolveAsync(SolverConfigurationSystem);
      ScanDuffing1.InsertSynchronisationPoint();

      SaveData(ScanDuffing1, DataFile, NumberOfThreads);


      ScanDuffing2.SynchroniseSolver();
      ScanDuffing2.SynchroniseFromDeviceToHostAsync(All);
      ScanDuffing2.SolveAsync(SolverConfigurationSystem);
      ScanDuffing2.InsertSynchronisationPoint();

      SaveData(ScanDuffing2, DataFile, NumberOfThreads);
   }

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

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

Observe that compared to the reference case, the number of simulation launches are halved due to the usage of two Solver Objects. More precisely, only a single launch is enough, as each Solver Object is responsible for a half of the total number of problems (instances of the ODE). The tutorial specific function called `FillControlParameters(args)` fills up the passed Solver Object with the aforementioned data. After finishing this task on the first Solver Object (**`ScanDuffing1`**), the CPU immediately put the following

tasks to the associated CUDA stream asynchronously: a) copy the data from the System Memory to the Device Memory, b) perform an integration phase and c) insert a synchronisation point to be able to check whether the simulation of the first integration phase is completed or not. Since all these operations are asynchronous, while the GPU is working on the above initiated tasks, the CPU can proceed further and fill up the second Solver Object (**`ScanDuffing2`**) with the rest of the data. Moreover, after that, the CPU can again immediately initiate the copy process, the integration phase and can insert a synchronisation point.

The next loop iterates over the initial transient performing integration phases one after another. The technique to dispatch the tasks are very similar. The difference is that data copy between the host side and the device side is not necessary as they are not used during the transient phase. Moreover, synchronisation takes place before the initiation of another integration phase, and a synchronisation point is inserted right after the initiation of the integration (to be able to synchronise it in the next cycle). This is done to avoid overfilling of the CUDA streams of the Solver Objects. Observe that inside the loop there are only asynchronous GPU related function calls. Therefore, after every function call the CPU immediate calls the next one almost without no delay. Without synchronisation, this would result in CUDA streams each filled immediately with 1024 number of kernel launches (integration phases)[9].

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

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

14.3.2 Results

The results of this tutorial example is exactly the same as the reference example discussed thoroughly in the main part of this manual. Only the total runtime is different. Using the Nvidia GeForce GTX Titan Black graphics card, the total simulation of the reference case is 33.5 s. This simulation time is reduced to 28.3 s by employing the double buffering technique introduced in this tutorial example. The relative small performance increase is due to much higher required time to save data to disc than to perform an integration phase (approximately 22 ms vs. 88 ms). Therefore, only a fraction of the saving process can be overlapped.

14.4 Tutorial 4: using multiple GPUs in a single node

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

```
int PoolSize        = 46080;
int NumberOfThreads = 23040;

ListCUDADevices();
int SelectedDevice1 = 0; // According to the output of ListCUDADevices();
int SelectedDevice2 = 2;

ConstructorConfiguration ConfigurationDuffing;
   ConfigurationDuffing.PoolSize                  = PoolSize;
```

[9] 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.

```
    ConfigurationDuffing.NumberOfThreads           = NumberOfThreads;
    ConfigurationDuffing.SystemDimension           = 2;
    ConfigurationDuffing.NumberOfControlParameters = 1;
    ConfigurationDuffing.NumberOfSharedParameters  = 1;
    ConfigurationDuffing.NumberOfEvents            = 2;
    ConfigurationDuffing.NumberOfAccessories       = 4;

ProblemSolver ScanDuffing1(ConfigurationDuffing, SelectedDevice1);
ProblemSolver ScanDuffing2(ConfigurationDuffing, SelectedDevice2);
}
```

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

During the computation similar techniques have to be used already introduced in the third tutorial example. Therefore, if the reader is not familiar with asynchronous member functions of the Solver Objects He/She is referred back to the third tutorial example discussed in Sec. 14.3. The control flow is also very similar to the one shown there, see the following listing

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

FillSharedParameters(ScanDuffing1,Parameters_B);
ScanDuffing1.SynchroniseSharedFromHostToDeviceAsync();

FillSharedParameters(ScanDuffing2,Parameters_B);
ScanDuffing2.SynchroniseSharedFromHostToDeviceAsync();

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

   FillControlParameters(ScanDuffing1, Parameters_k_Values, InitialConditions_X1,
       InitialConditions_X2, FirstProblemNumber, NumberOfThreads);
   ScanDuffing1.SynchroniseFromHostToDeviceAsync(All);
   ScanDuffing1.SolveAsync(SolverConfigurationSystem);
   ScanDuffing1.InsertSynchronisationPoint();

   FirstProblemNumber = FirstProblemNumber + NumberOfThreads;

   FillControlParameters(ScanDuffing2, Parameters_k_Values, InitialConditions_X1,
       InitialConditions_X2, FirstProblemNumber, NumberOfThreads);
   ScanDuffing2.SynchroniseFromHostToDeviceAsync(All);
   ScanDuffing2.SolveAsync(SolverConfigurationSystem);
   ScanDuffing2.InsertSynchronisationPoint();

   for (int i=0; i<1023; i++)
   {
      ScanDuffing1.SynchroniseSolver();
      ScanDuffing1.SolveAsync(SolverConfigurationSystem);
      ScanDuffing1.InsertSynchronisationPoint();

      ScanDuffing2.SynchroniseSolver();
      ScanDuffing2.SolveAsync(SolverConfigurationSystem);
      ScanDuffing2.InsertSynchronisationPoint();
   }

   ScanDuffing1.SynchroniseSolver();
   ScanDuffing2.SynchroniseSolver();
   for (int i=0; i<32; i++)
   {
      ScanDuffing1.SolveAsync(SolverConfigurationSystem);
      ScanDuffing1.SynchroniseFromDeviceToHostAsync(All);
      ScanDuffing1.InsertSynchronisationPoint();

      ScanDuffing2.SolveAsync(SolverConfigurationSystem);
      ScanDuffing2.SynchroniseFromDeviceToHostAsync(All);
```

```
        ScanDuffing2.InsertSynchronisationPoint();


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

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

The main difference is that here there is no overlapping computations with the CPU. After dispatching integration phases to both GPUs, they are synchronised (both) before proceeding further with any other tasks. Therefore, the main gain here is the parallel computation of each half of the total number of problems distributed to different GPUs. Observe that asynchronous function calls are still necessary, otherwise the computation cannot be issued to the other GPU (the CPU will wait for the first computation of the previous GPU to be finished). In order to overlap CPU-GPU computations, at least two Solver Objects have to be associated to each GPU similarly as in case of the previous tutorial example. This means at least 4 Solver Objects in this specific example. It is left to the reader to assemble a control flow for such a problem. However, keep in mind that the total number of problems has to be divided up to 4 different parts.

14.4.1 Results

Again the results of this tutorial example is exactly the same as the reference example. Therefore, only the runtimes are discussed here. Using a single Nvidia Tesla K20m graphics card for the reference case, the total simulation time is 30.1 s while the time needs for the transients is 22.9 s. This simulation times are reduced to 27.3 s and 20.5 s by employing the double buffering technique introduced in the third (previous) tutorial example (the graphics card is still the Nvidia Tesla K20m). Finally, the runtimes are further reduced to 18.6 s and 11.5 s using two Nvidia Tesla K20m graphics cards.

Observe that the transient part scales almost linearly with the number of the GPUs. Such a comparison, however, is NOT perfectly exact. In the double buffering case, there is always at least one active kernel function (running simulation) on the sole GPU due to the overlapping technique. In contrast, in the multi GPU example, synchronisation points applied to a GPUs is an implicit synchronisation point to the other one as well (since the CPU will not proceed further even if the other GPU has already finished its work); thus, there can be cases where one GPU is idle waiting for the other to finish its work.