POLITECNICO DI MILANO

Dipartimento di Scienze e Tecnologie Aerospaziali

Master of Science in Aeronautical Engineering

Master Thesis

Accelerating reactive flow simulations via GPGPU ODE solvers in OpenFOAM

POLITECNICO
MILANO 1863

Advisor:
prof. Federico Piscaglia

Co-advisor::
Federico Ghioldi

Candidate:
Filippo Trevisiol
921326

Academic year 2019/20

Abstract

The simulation of a combustion phenomenon is a challenging task in modern Computational Fluid Dynamics (CFD) due to the high complexity of the experiment. The stiffness of the chemical mechanism, in addition to a large computational mesh and the CPU serial architecture, can increase the computational effort of several order of magnitude. In the last years, General Purpose Graphic Processor Unit (GPGPU) computing has become very important in computer industry thanks to its capability to exploit massive parallelization and achieve great performance improvements. Thanks to this technique, it should be possible to obtain the same results of a Central Processing Unit (CPU) combustion code while saving plenty of time. This master thesis offers a possible implementation of GPGPU computing in the chemistry combustion model that is present in the open-source CFD software OpenFOAM. The code adopted is introduced and main planning choices are described. An explicit adaptive Runge-Kutta Cash-Karp of 4th and 5th order has been used. Different chemical kinetic mechanisms were tested, showing that accuracy is preserved when using an hybrid CPU/GPU implementation. Furthermore, an increase of the performance has been obtained for all the multi-cell test case. Low, medium and high-stiffness combustion mechanisms performed more than 1.4, 3.5 and 4.3 times faster than the original counterpart. Finally, possible future implementation were discussed in order to achieve a better speed-up.

Sommario

La simulazione di fenomeni di combustione nella moderna CFD è un'operazione ambiziosa a causa della sua enorme complessità. La rigidità (stiffness) del meccanismo chimico, la dimensione della mesh utilizzata e l'architettura seriale della CPU possono andare ad aumentare il costo computazionale di diversi ordini di grandezza. Negli ultimi anni la *General Purpose Graphic Processor Unit (GPGPU) computing* è diventata sempre più importante nell'industria informatica a causa della sua capacità di sfruttare un'ingente parallelizzazione e ottenere grandi miglioramenti in termini di performance. Grazie a questa implementazione, è possibile ottenere gli stessi risultati di un codice di simulazione di fenomeni di combustione funzionante sulla CPU ma risparmiando molto più tempo. Questo lavoro di tesi offre una possibile implementatazione della GPGPU per i solutori della combustione presenti nel software open-source OpenFOAM. Si descriverà il codice prodotto e le maggiori scelte progettuali adottate. Per poter portare il codice in GPU, è stato scelto di utilizzare un metodo di integrazione esplicita delle equazioni differenziali ordinarie derivanti dalla chimica, in particolare il metodo di Runge-Kutta Cash-Karp del quarto e quinto ordine. Sono stati testati meccanismi chimici differenti, andando a mostrare come l'accuratezza delle soluzioni si preservi passando ad un'implementazione CPU/GPU ibrida come quella descritta. Inoltre, per i casi multi cella si ottiene anche un sensibile incremento nelle performance. I casi testati, a vari livelli di rigidezza numerica, hanno ottenuto un miglioramento rispettivamente di 1.4, 3.5 e 4.3 volte in confronto alla simulazione svolta totalmente in CPU. In conclusione, sono state presentate alcune possibili implementazioni future per poter ottenere un ulteriore miglioramento.

Ringraziamenti

Desidero ringraziare innanzitutto il Prof. Federico Piscaglia per avermi dato l'opportunità di poter lavorare con lui e poter portare avanti questo lavoro di tesi, oltre che per il suo aiuto durante questi mesi.

Ringrazio inoltre il mio correlatore Federico Ghioldi per la pazienza avuta nei miei confronti, per i numerosi consigli che sicuramente hanno ampliato il mio bagaglio di conoscenze e per tutto il tempo dedicatomi durante la realizzazione di questo lavoro.

Desidero poi ringraziare la mia famiglia che mi è stata accanto in questi anni di studi, non facendomi mai mancare nulla e supportandomi sempre in questo percorso universitario. Grazie per i sacrifici che avete fatto per permettermi di completare questo percorso. Questa tesi è dedicata a voi.

Infine, desidero ringraziare i miei amici più cari ed Alberto per il supporto che mi è stato dato e per tutte le esperienze che ho condiviso con loro in questi anni, dandomi la giusta carica e serenità per vivere al meglio il periodo universitario.

Contents

List of Figures

List of Tables

Nomenclature

$\dot{\omega}_i$	rate-of-progress variable of i-reaction
$\dot{\omega_k}$	reaction rate of k-species
$\dot{Q}$	chemical heat source
$\dot{T}$	temperature variation in time
$[\mathrm{X}_k]$	molar concentration of k-species
$\Delta h^0_{f,k}$	formation enthalpy of k-species at T_0
μ	cinematic viscosity
ν_k	stochiometric coefficient of k-species
Ω	control volume
ρ	density
ρ_k	density of k-species
τ	viscous stress tensor
$\mathbf{g}$	gravity acceleration
$\mathbf{I}$	identity tensor
$\mathbf{n}$	normal versor
$\mathbf{q}$	heat flux
$\mathbf{T}$	shear stress tensor

$\mathbf{U}$	velocity field
$\mathbf{U}_b$	mesh velocity
$\mathbf{V}_k$	diffusion velocity
AT_a	collision frequency
c_p	heat capacity at constant pressure
c_v	heat capacity at constant volume
e	specific internal energy
E_a	activation frequency
h	specific enthalpy
K	specific kinetic energy
K_c	equilibrium constant
k_f	forward reaction rate constant
k_r	reverse reaction rate constant
M_k	k-chemical specie
p	pressure
p_k	partial pressure of k-species
R	universal gas constant
r	specific heat source
S	momentum source/sink
T	temperature
t	time
W	molecular weight of mixture

W_k	molecular weight of k-species
X_k	mole fraction of k-species
Y_k	mass fraction of k-species

Chapter 1

Introduction

Computational fluid dynamics (CFD) simulations attempt to describe appropriately the fluid mechanics in different situations. Like in experimental tests where a more powerful and precise equipment can bring a better insight on the phenomena under consideration, usually in numerical experiment a more accurate result is achieved increasing the complexity of the simulation in terms of geometry and time discretization. This can cause a much higher cost of the overall activity. Therefore, one of the most important issue when performing a CFD test is the computational cost that could overcome the available resources.

In particular, an important research field in fluid dynamics is represented by combustion problems. This world is extremely various, including very different applications; many of them are present in every day life. Usually, different species are introduced in a combustion chamber: the exothermic interaction between an oxidant and a reductant (the fuel) releases heat that can be used as energy. Main goals in combustion research are focused in increasing the heat generated by the reaction, analysing the conditions for which combustion takes place, reducing the level of pollutant generated, and much more. Nowadays, a computational approach guarantees the same accuracy in the results as an experimental procedure. While a numerical simulation can be cheaper from an economic point of view with respect to the experimental counterpart (that needs expensive tools, a suitable place, safety procedures, ...), it can become very expensive regarding the computational cost.

Since combustion problems require the solution of several chemical kinetics ordinary differential equations (ODEs) for each computational cell, high computational power is needed in order to achieve good results in reasonable time. As a matter of fact, the chemical differential equations were solved serially by CFD software like OpenFOAM; thus, the cost increases with mesh refinement. Furthermore, this particular equations can be hard to solve because of the chemical stiffness and the large number of species and reactions involved in the mechanism. Solving reacting flows motion in huge domains for an important time range can be exorbitant for companies and small institutions that usually do not own great computational resources.

In the last decades, supercomputers and multi-core Central Processing Units (CPUs) have progressively replaced single-core CPU computers. The massive parallelization that can be reached with this type of systems has literally changed the paradigm of scientific computing, allowing to sensibly reduce the computational cost. Regarding CFD, this is achieve by splitting the mesh through the CPU cores. Anyway, this solution requires great CPU clusters to get enormous advantages due to the fact that micro-transistors technology has almost reached its maximum development. Thus, a cluster is very expensive and it can be possible for established companies and big institutions only. A more interesting solution is the massive parallelization offered by the NVIDIA graphics card. Many researches have shown that this approach can speed up the operations about dozen of times, depending on the implementation and the hardware specifications; moreover, this tool can be useful for a large variety of problems. This technique is called General Purpose Graphics Processor Unit (GPGPU) computing and it can be implemented thanks to a specific programming language (based on C++), named CUDA (Compute Unified Device Architecture) that was developed by NVIDIA.

Several recent studies have investigated the usage of GPGPU for solving chemical kinetics ODEs in parallel [1–5]. Spafford et al. [1] analyzed S3D, a Fortran-based DNS solver for the fully compressible Navier-Stokes equations coupled with chemistry. They reported a kernel speed-up of 31.4x for the single precision version and of 17.0x for the double precision version with respect to the CPU version. Niemeyer and Sung [2] implements Runge-Kutta explicit integration methods to solve chemical mechanism with different stiffness level, demonstrating that also high stiffness mechanism (i.e. ethylene oxidation) can be solved with an explicit method on GPU. They obtained a 17x speed-up with respect to six-core RKC on CPU implementation for about 30’000 ODEs. Stone and Davis [3] import explicit and semi-implicit integration Runge-Kutta methods on CUDA for solving independent chemical kinetics ODEs. They also compared different algorithm showing the dependency of the results from the number of ODEs solved in parallel.

While GPGPU has been used in a slight number of CFD solvers nowadays, latest OpenFOAM version has not the possibility to use the advantages of a graphic card utilization for combustion problems. Nevertheless, researches are moving in this direction. In particular, a library written in CUDA-C was developed for chemical kinetics OpenFOAM solver (`chemFoam`) in Politecnico di Milano by Ghioldi [6]. The work briefly analyzes a Runge-Kutta Cash-Karp implementation in CUDA, showing same accuracy (figure 1.1) but better performance in terms of speed referring to the original CPU version. In particular, the research shows an operation speed-up of 2.4x with respect to the CPU for a simple tutorial case (H_2/O_2 combustion); nonetheless, memory allocation and data copying, as well as the activation of CUDA libraries, do not allow a fastest overall simulation time, suggesting that the best results can be achieved with a greater number of cells (figure 1.2).

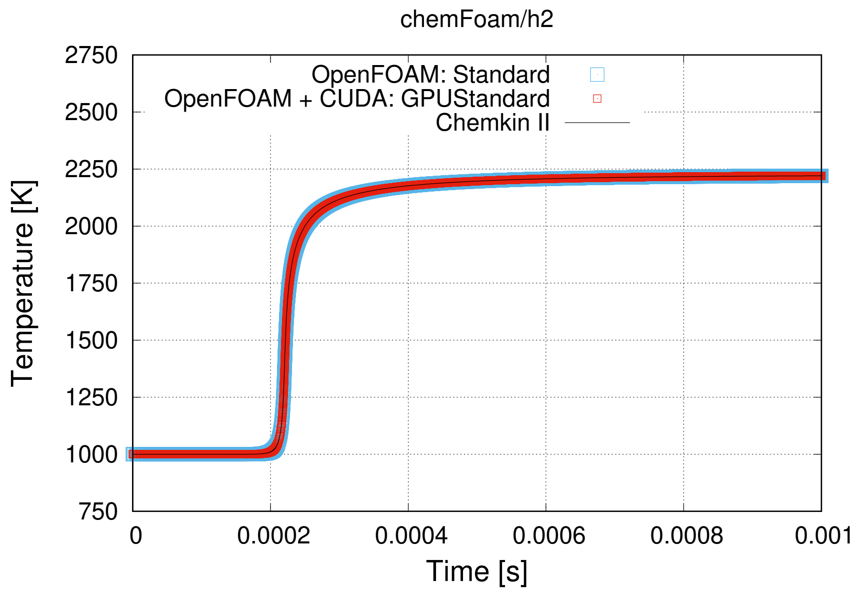


Figure 1.1: Temperature profile of chemFoam/h2 tutorial (Ghioldi [6]).

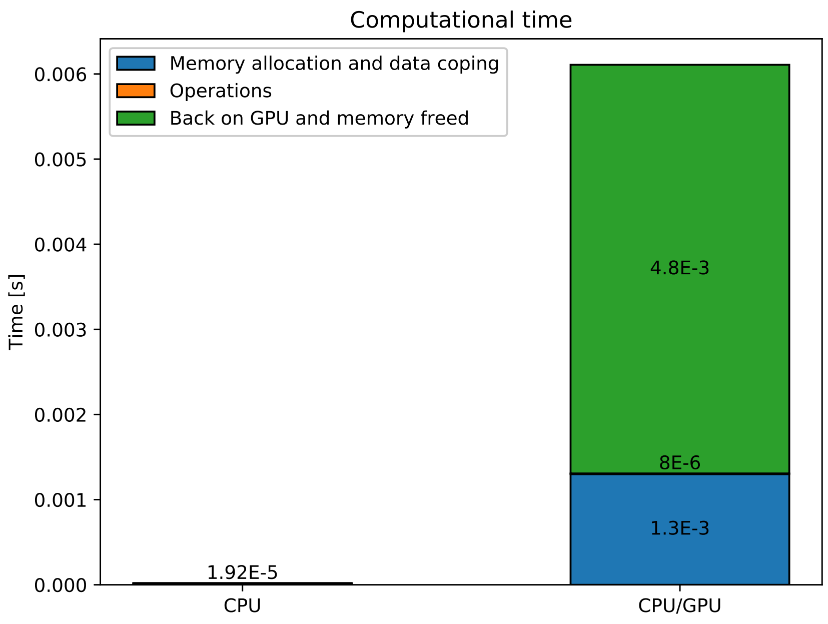


Figure 1.2: Computational time for runtime steps of chemFoam/h2 tutorial (Ghioldi [6]).

Starting from Ghioldi's work, the main goal of this master thesis is to validate the GPU-based chemical solver in terms of results. To do this, different chemical kinetics model will be tested with the single-cell OpenFOAM solver `chemFoam`. Moreover, the code has been modified and improved in order to support multi-cells computing of reacting flows (`reactingFoam`). This solver is used for spray-less reacting flows simulation and gives a better picture of the potentiality of GPGPU chemistry application, while the single-cell implementation can be very useful to test a new chemical mechanism but it does not reproduce a concrete physical phenomenon. Also the multi-cell has been validated showing the accuracy of results with respect of the CPU counterpart. Since the goal of a GPU implementation is to reduce computational cost, speed test have been performed in order to show improvements in terms of simulation times. Actually, it is expected that best speed-up will be achieved for a great number of cells and a complex chemical mechanism, while lower mesh size and simple reactions will not greatly benefit from a graphic card implementation.

1.1 Structure of the thesis

The thesis is structured as follows:

- **chapter 2** briefly presents the history of GPGPU and why it has been introduced in computer market. Then CUDA programming language is described, focusing on the main features. Finally, the hardware adopted for the tests is illustrated.
- **chapter 3** shows the physical and mathematical problem of the chemical kinetics, highlighting the model equations of the problem and the solution methods.
- **chapter 4** describes how the physical problem has been implemented in OpenFOAM thanks to the special features of GPGPU computing.
- **chapter 5** presents the results of several test cases, focusing the attention of the differences with the original OpenFOAM version.
- **chapter 6** concludes the work, summarizing the topic analyzed and briefly reviewing possible future implementations.

Chapter 2

An overview on GPGPU computing

In this chapter, General Purpose GPU computing is presented. A short review of the milestones in the history of CPU and parallel computing is first introduce, to better motivate the purpose of this work giving it a contextualization. Afterwards, principle of parallel computing will be defined. CUDA language will be presented in order to highlight the most important general concepts. These topics are principally taken from the documentation available with the CUDA toolkit, in particular “CUDA C programming guide” and “CUDA C Best practices guide” [7, 8]. Other concepts are taken from *Professional CUDA c programming* [9] and *CUDA by example: an introduction to general-purpose GPU programming* [10]. Finally, hardware specifications used for the goal of this work will be presented.

2.1 History of parallel computation

In the last few years, parallel computation has become very important in the computing industry because of its enormous advantages in computational speed.

For decades, CPU’s manufactures have tried to increase performance of devices rising the processor’s clock speed. In 1965 the future co-founder of the Intel electronic company published “Cramming more components onto integrated circuits”, making an empirical observation that it would become famous as Moore’s law [11]:

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.*

Assuming a proportional relation between number of transistor and chip complexity, this statement has kept its validity for years. By the way, in the last decades CPU producers have realized that Moore’s processor rate of growth has slowed down. Year by year, transistors have become even smaller, reaching a size limit. Furthermore, smaller transistors dissipate much more heat, that requires bigger cooling systems (with cost increasing). Moore’s forecasts can not be observed yet due to the rising of technology complexity. So CPU producers have decided to add processors on personal computers instead of keeping on growing computational clock speed [10]. Initially adopted in companies and researcher supercomputer, this solution have reached the consumer PC market starting the so called *multicore revolution* (Herlihy, [12]). *Parallel computing* was born consequently: computation is divided in several blocks and is carried on by the multiple cores in parallel.

Graphical Processor Units (GPUs) became important in the computer history in the 90s when desktop computers began to spread among average consumers and OS companies (i.e. Microsoft, Apple) implemented first software that required high graphical operations. Before 1991, all the graphics operations were carried out by CPU. In 1991 S3 Graphics started selling the first graphics accelerator. Users initially bought 2-dimensional graphic accelerators, while in the same time Silicon Graphics developed and released its OpenGL library, introducing 3-dimensional graphics in the computer world. By the end of the 20th century, the demand of GPU increases rapidily, thanks also to the even more elaborated video-game industry. Companies such NVIDIA and ATI technologies

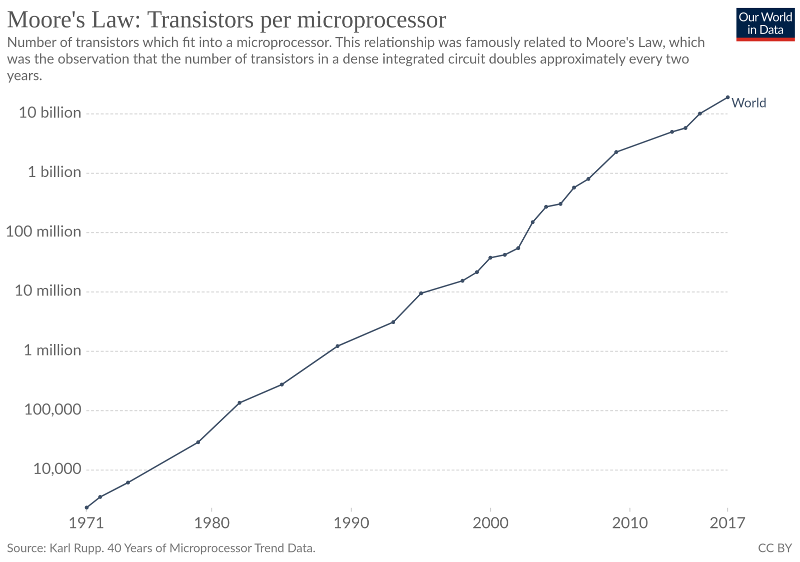


Figure 2.1: Transistor per microprocessor from 1971 to 2018 (*Published online at OurWorldInData.org*, [13]).

starts investing in these new technologies.

In 2001 NVIDIA released the GeForce 3 series, breaking into the graphic processors market. These GPUs used a programmable arithmetic unit, called *pixel shader*, to produce colors on each pixel on the screen. A pixel shader uses the (x, y) position of each point on the screen, together with other additional information (e.g. color, brightness, ..): through the combination of all the information, the picture can be displayed. Thus, the video output depended on a combination of external inputs given by the programmer, so they start implementing various computing language to deal with this possibility. In addition to that, programmers discovered that inputs could be any kind of data. In other words, giving a color input with a particular meaning (e.g. different levels of brightness mean different numerical values) it was possible to obtain a result that was derived by a numerical computation of the data, using the graphical power of GPUs. What seems to be a great intuition, able to disrupt the way of thinking scientific computing, was in the first years very limited. Color data input and output were too much difficult to manage, in addition to the fact that no appropriate debugging method existed. Again, it was only thanks to

NVIDIA that in 2006 a new GPU architecture able to be programmed was released. General Purpose GPU (GPGPU) became popular and applied to a huge field of applications.

In the last years, GPGPU has been developed for a great number of field that have some common characteristics [14]. First of all, these applications demand large computational requirement: GPUs in fact can executes millions of operations at the same time, as real time rendering requires the contemporary computation of millions of pixels. Second, they require parallelism. Third, the throughput is more important than latency: modern processor takes some nanosecond to perform operation, while a human is able to catch visual information on a millisecond scale; this huge gap implies that the latency is less important than the quantities of information processed on a time interval. Some of GPGPU applications are:

- scientific computing, like weather forecasting, climate researching, ray tracing Montecarlo simulations;
- medical imaging;
- cryptocurrency mining;
- computational fluid dynamics (CFD) simulations.

In particular, computational fluid dynamics has exploited the great potential of GPU computing. The perspectives of achieving large parallelizations and great speed-ups well fit with the large amount of data structure and the great computational effort in CFD simulations. As it will be explained later, the ability of executing the same task concurrently can give huge advantage in fluid dynamics, where all the finite volume in the computational domain do essentially the same activities. The GPU implementation of chemical combustion, topic of this work, will be described accurately later in chapter 4.

2.2 Basic concept on parallel computing

Before describing how parallel computation can be transposed in programming language thanks to the NVIDIA tools, some definitions and concepts concerning parallel computation are necessary.

It is compulsory to define two basic concept, **latency** and **throughput**. Latency indicates how long it takes from an instruction to be completed since it has been issued. Throughput gives information about how many instruction are processed in a time unit.

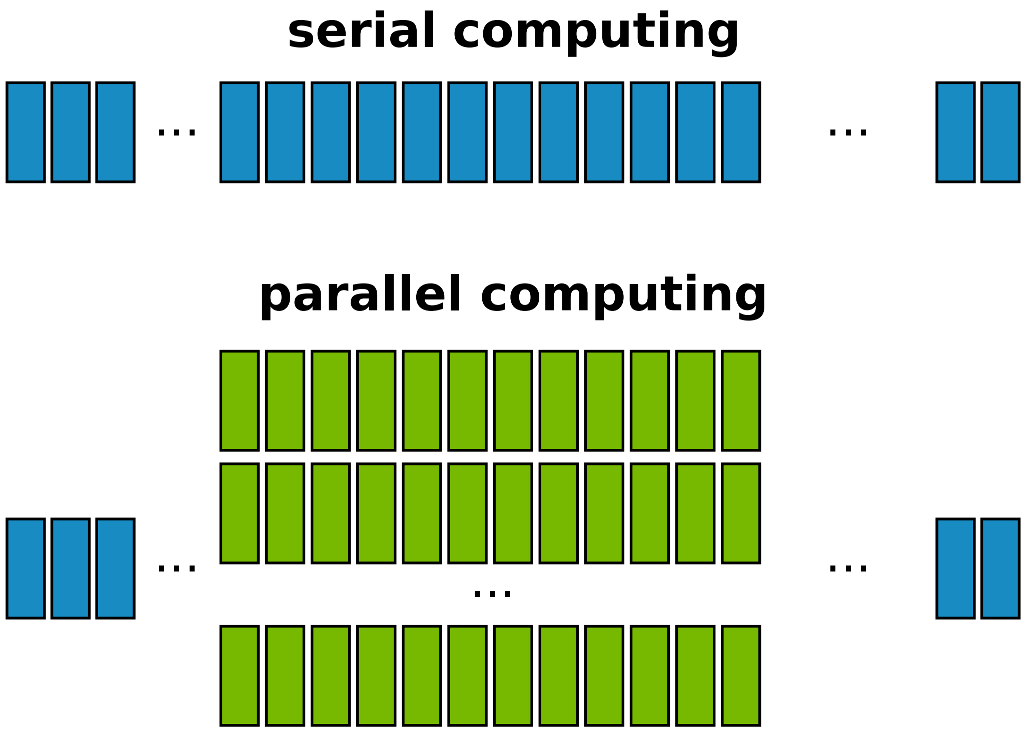


Figure 2.2: Serial and parallel computation.

Talking about parallel computing, we refer to two different points: an hardware aspect, that is due to the computer architecture, and a software aspect, that consist in solving a problem concurrently with the power of the hardware chosen.

First of all, in computer language a **task** is the elementary part in what a computational problem can be broken down: a task receives an input, does a function and gives back an output. A classification of programming can be made from the point of view of how the task are executed during the program. In **serial** (or **sequential**) **programming** each instruction is executed in a certain order and each activity starts when the previous has ended. On the other side, in **parallel programming** tasks are executed simultaneously. Figure 2.2 shows schematically the two different implementations. Clearly, parallel computing can have some sequential part.

Two different types of parallelism are common these days, **task** and **data parallelism** that differ in what is the object of parallelization. The first type concerns parallelization of tasks among multiple threads, the second is related to these kind of implementation where each core executes the same tasks but with different data. CUDA language is better compatible with data parallelism.

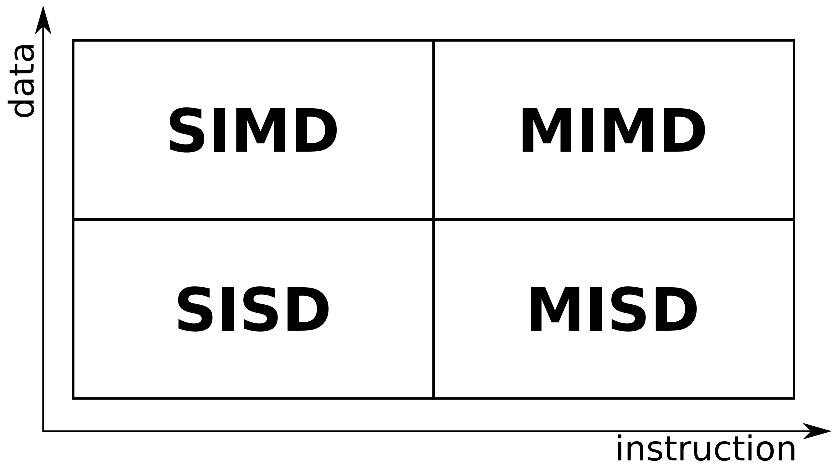


Figure 2.3: Flynn's taxonomy.

Another classification is related to computer architectures (figure 2.3). It was proposed by Flynn in 1966 and then in 1972 [15, 16] and it classifies computers based on data and instruction streams concurrently executed. Even though more detailed classifications have been made through the years, Flynn's taxonomy is still valid. Four different categories can be defined:

- **Single Instruction Single Data** (SISD): it is the simplest architecture, with one single core. Instruction are executed serially on a single data.
- **Single Instruction Multiple Data** (SIMD): there are multi cores that executed a single instruction, but on multiple different data. Usually, a single central processor sends instructions to the cores that operates on different data streams.
- **Multiple Instruction Single Data** (MISD): up to now, there are no commercial implementation with this kind of architecture.

- **Multiple Instruction Multiple Data** (MIMD): an example could be computer clusters.

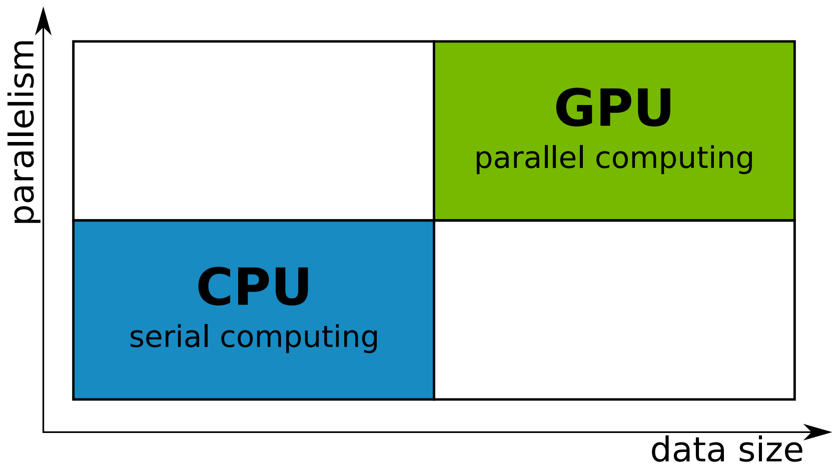


Figure 2.4: GPU VS CPU.

Computer architectures can also be classified on the base of their memory organization. In **multi node systems** (often referred as clusters), there is a series of processors, each of them constituting a single engine, connected each other with a common distributed memory. On the other side, in **multiprocessors systems** there can be thousands of processors that are connected each other to the same memory or have the same low latency link (as a PCI-Express bus).

In conclusion, it seems that GPU can always be the proper choice for programming compared to a traditional CPU. Really, the usage of the two devices depends on the particular programming case. While GPUs give the best results with data-parallel computation task, CPUs reach better goals with control task. For the best overall performance, both the devices should be used in what is called *heterogeneous computing* that will be described accurately in section 2.3. Figure 2.4 highlights the main differences between the two architectures. Small amount of data and low parallelism are better handled by CPUs due to the ability of executed complex tasks and instruction levels parallelism; on the opposite, GPU can work with large amount of data and massive parallelism.

2.3 Parallel computation with CUDA

The computer graphics company NVIDIA had developed its own personal platform to work with GPGPU computing on its GPUs. This architecture was introduced in 2006 with GeForce 8800 GTX. CUDA stands for Compute Unified Device Architecture and it refers to both the hardware architecture of the GPU and the API needed to develop software.

A NVIDIA GPU is made of single components named **Streaming Multiprocessors** (**SM**). This element is the main responsible for the high grade of parallelism that can be achieved in GPUs. SMs are very similar to the cores in multi-core CPU implementation. There are several streaming multiprocessors in a GPU, depending on the particular device.

An example of NVIDIA GPU streaming multiprocessor is shown in figure 2.5: the NVIDIA GeForce 930MX, used for the purpose of this work, uses this exact architecture, called *Maxwell architecture*. Of course other architecture exists and the Maxwell architecture is a natural evolution of the previous architecture, the Kepler architecture. The main components that are part of the streaming multiprocessor are the CUDA cores, 128 in Maxwell. They are also called streaming processors and they are the smallest unit where the computation takes place. Each CUDA core is composed of an aritmetic logic unit (ALU) and a floating point unit (FPU) that executes one integer or floating point operation per clock cycle. In addition to the cores, we can notice 32 load-store units (LD/ST or LSU), that are responsible for the load of data from memory or store on memory from the registers, and 32 special function unit (SFU), that execute transcendental operations (e.g. sine, cosine, square root). It is noticeable that all these elements are grouped and each group is controlled by an instruction buffer, a warp scheduler and 2 dispatch units. Both the elements are in charge of organizing the way in which data and operation are processed by the GPU, as it will be explained later on this section. In previous GPU architectures (as the Kepler one), this partition does not exists and these scheduling elements control all the cores. Finally, three different memory locations complete the Maxwell streaming multiprocessor: a 64 KB shared memory that let the CUDA cores communicate; 4 registers memory; 2 texture/L1 Cache memory. A detailed description of the different types of GPU memories will be presented later.

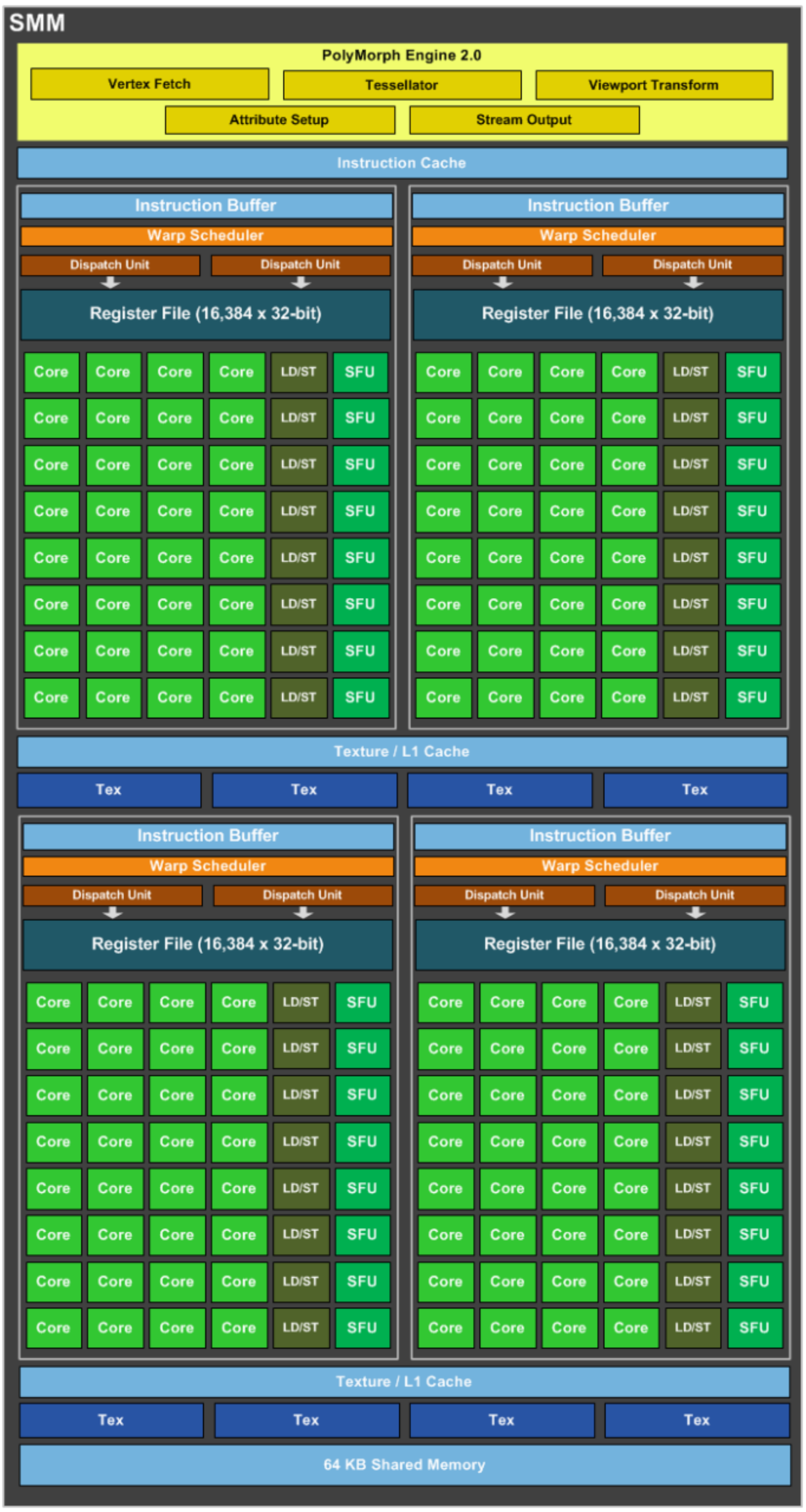


Figure 2.5: NVIDIA Maxwell Architecture

The CUDA execution model reflects the GPU hardware implementation. The smallest element of CUDA parallelism is the **thread**. Threads correspond to the CUDA cores in a streaming multiprocessor and each of them execute the same instruction with different data inputs, thus it is possible to compute the same task dozen of times concurrently on the same GPU. Threads are organized in **thread-blocks** (or simple **blocks**): each of them is executed in a single SM and its execution cannot be divided in multiple SM. The SMs can compute several blocks, depending on hardware limitations. Also the thread-blocks can compute parallel tasks. The main difference between a block and a thread is that the latter can share information with other threads in a block, while communication inter-blocks is not possible. This intra-block communication is possible thanks to a special SM memory, named shared memory, that usually is very small compared to other GPU memories (some KB) and consequently there are often memory limitations on the number of threads that can be used for each block. Finally, several blocks form a **grid** that can have more than one dimension: multi-dimension grids are very important for graphical applications but less for other uses so they will not be mentioned in the next pages.

Not all the blocks and the threads can be executed simultaneously on the GPU. Threads are processed in groups of 32 elements, named **warps**. Of course, blocks with more than 32 threads are organized in multiple warps and the number of warps is calculated as follow:

$$WarpsPerBlock = ceil(\frac{ThreadsPerBlock}{32}) \tag{2.1}$$

For an optimal performance, blocks should have a number of threads multiple of 32. A warp can not be split between different blocks. When all the resources of a block have been allocated on registers and share memory, the block is active (and so the warps). A warp is defined as *selected* when it is ready to execute the instruction and it is dispatched to the execution unit, as *eligible* when it is ready but is not currently executing, and as *stalled* whenever the warp is not read. The warp schedulers give the instruction to the warps that execute the command concurrently (up to 64 simultaneous warps in the latest NVIDIA architectures). When a warp finishes its directive, another one is activated by the scheduler and executes the same instruction.

This procedure has the important role of hide the latency by promoting the throughput. **Latency hiding** is an important concept in parallelism. While CPUs speed up program execution by minimizing the latency, GPUs manage huge amounts of small threads maximizing the throughput. So the latency in GPU is hidden by the computation of other warps. Most significant latency is due to load-store operations (400-800 clock cycles), while arithmetical operation have latencies of about 20-40 clock cycles.

An index of how the GPU manages the warp parallelism is expressed by the **occupancy**, defined as the ratio between the active warps and the maximum warps per SM. When a 100% occupancy is achieved, the maximum number of warps are executed concurrently on a SM. Nevertheless, the number of resident blocks or threads on the same SM is limited and depends on the particular hardware utilized. In particular, it is related to three aspect:

- hardware restriction, as the maximum number of warps per SM (64) and the maximum number of threads per SM (2048);
- shared memory per SM and shared memory per block;
- number of registers and register memory.

A good programmer has to manage all these three limitations in order to find the best equilibrium. Notwithstanding, increasing the number of resident blocks and warps per SM does not imply an increment of performance (Volkov, [17]). Finally, a thread-block remains activated on a SM until all the tasks on its threads are ended.

The NVIDIA GPU architecture, on the basis of the hardware and software implementation previously presented, can certainly be classified as SIMD following the Flynn's taxonomy. Actually NVIDIA coined the term *Single Instruction, Multiple Threads* (SIMT) because all the threads in a warp execute the same instruction on the same time. A problem that can arise when this does not happen is called **warp divergence**. This can be caused when threads in the same warp take different paths in a program due to a conditional statement (`if`, `if-else`, `switch`). If threads in a warps diverge, each path is executed serially; threads that are not part of the path currently executed are disabled until the branch is over. This causes poor performance of course. A simple scheme of the CUDA blocks and threads division is shown in figure 2.6.

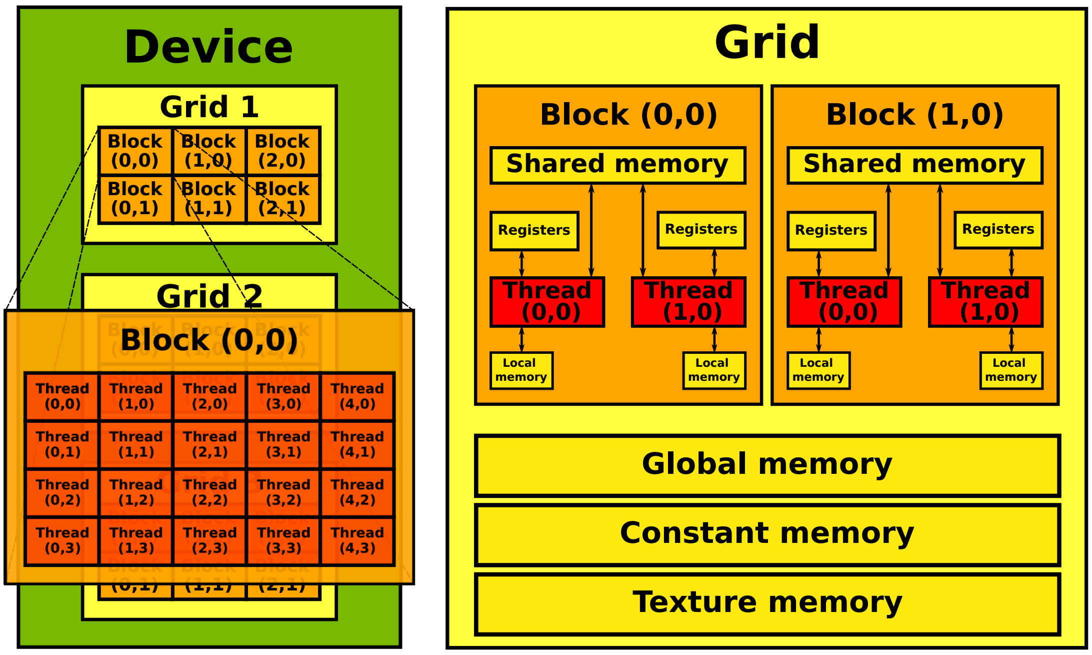


Figure 2.6: Simple representation of CUDA architecture.

Computing elements of CUDA GPU can access different types of memories, each of them having a different role in terms of parallelization, speed and storage. The most important memories are:

- register memory;
- local memory;
- shared memory;
- texture memory;
- global memory;
- constant memory.

Register memory collects data stored by a thread and it is visible only by the thread itself. When the thread becomes inactive after executing an instruction, data on registers are destroyed. Storage capacity is very small and there are limitation on the number of registers per thread due to the SM architecture. Nevertheless, registers are very quick accessible locations. Like CPU registers, the role of this memory partition is useful to store data that are frequently used by processor, increasing the execution speed. **Local memory** has the same role of register memory: it lasts until thread lasts but it performs slower. It has also high latency and it can store bigger size data with respect to the register. **Shared memory** is very important for parallelization because it allows data communication inter-threads in a thread-block. It has an high bandwidth and low latency, so it performs very fast. It lasts until the lifetime of the block. There could be the problem of synchronization between the threads, so data in shared memory must to be synchronize to perform exact calculations. **Texture memory** is a read-only memory that is used in principal for two-dimensional graphical tasks. **Global memory** is the most used memory and it has also the biggest size (in the terms of GB). It is visible by all the blocks on the device and it lasts until the allocation ends. Finally **constant memory** stores data that are declared as constant in the code (`__constant__`), so it is a read-only memory. It also has a small storage size. A schematic summary of the main features of each memory is presented in table 2.1.

memory type	scope	size	speed
registers	thread	very small	very fast
local	thread	small	very slow
shared	block	small	fast
texture	kernel	small	slow
global	kernel	high	very slow
constant	kernel	small	fast

Table 2.1: Summary of most important memories features.

GPU provides also cache memory, non-programmable fast memory that stores data frequently requested for reducing memory latency. Type of cache memories in GPU are: L1, L2, read-only texture, read-only constant.

CUDA parallel programming is based on the idea of **heterogeneous computing**. CPU and GPU concurrently participate to the execution of the program and each of them does different activities. Data transfer between the two components is permitted by a PCI (Peripheral Component Interconnect) bus. The opposite programming structure is called **homogeneous computing**, where tasks are performed by one or more processors (typically CPUs) with the same architecture. Thus a CUDA application consists of two parts:

- the **host** code;
- the **device** code.

The host code refers to the CPU, the device code to the GPU. The code is initialising in the host, where data is allocated in device memory and device functions are called. Then in the GPU high performance operations are executed in parallel. A typical CUDA program can be summarized in these steps:

1. allocating of GPU memory;
2. copying data from CPU to GPU;
3. invoking CUDA functions to perform specific computations;
4. copying data back from GPU to CPU;
5. destroying data on GPU.

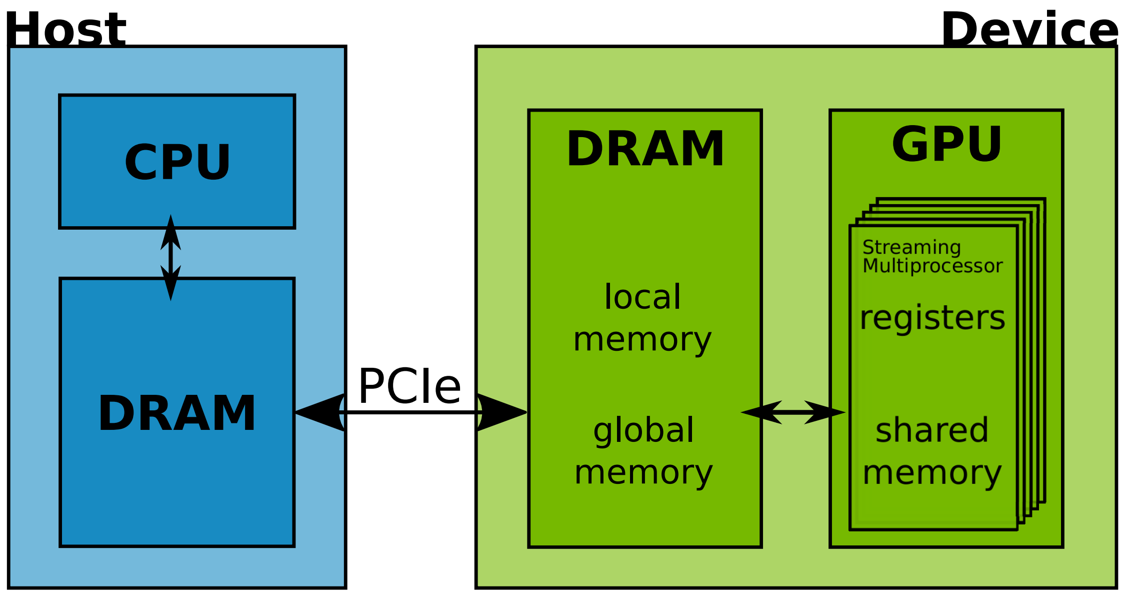


Figure 2.7: Host and device communication.

To execute the program on the device, the user has to write a particular function, named *kernel*. This function uses a special qualifier, `__global__`, and it has to be declared in the host code. The kernel is then invoked in the `main` function as:

```
myKernel<<<nBlocks, nThreads, sharedMem, stream>>>
    (
        arguments
    );
```

In addition to the function argument, the kernel has to be launched using a particular execution configuration syntax in triple "⋘ ... ⋙" brackets. The kernel parameters are: the number of thread-blocks; the number of thread per each block; the size of shared memory per block; the stream where the kernel is executed (for concurrent operation).

CUDA gives specific built-in variable to get the index of each element. The variable `blockIdx.x` gives the index of each block (in one dimension: other dimensions are characterized by suffix `.y` and `.z`), while `threadIdx.x` gives the index of each thread. There are also variables for size of the block (`blockDim.x`) and the grid (`gridDim.x`). Most of the times, it is useful to create a global index, to control all the threads of all blocks, in this standard manner:

```
tid = blockIdx + threadIdx * blockDim
```

An example of indexing with built-in CUDA variables is shown in figure 2.8.

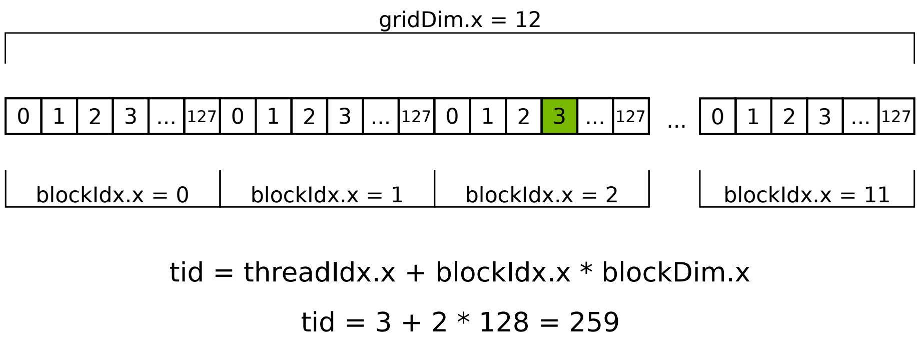


Figure 2.8: CUDA indexing example.

The `__global__` type functions are executed on the device but they are callable from the host. New generation GPU architectures have allowed function calling from the device as well. CUDA had defined other two function qualifiers: `__device__` and `__host__`. The former is executed on the device and is callable from the device only. All the extern functions that are called by the kernel code are of this type. The latter are executed by the host and are callable by the host only, so it can be omitted.

Operations of allocating, copying and destroying memory are possible thanks to specific CUDA functions, similar to the counterparts contained in `standard` C library:

- for memory allocation: `cudaMalloc` (`malloc` in C);
- for copying: `cudaMemcpy` (`memcpy` in C);
- for memory destroying: `cudaFree` (`free` in C).

All those functions are synchronous, which means that they wait previous operation to finish until they start. In fact, CUDA let the possibility to have asynchronous functions that are called without waiting previous operations to be finished (e.g. `cudaMemcpyAsync`). Thus, there is the possibility of running the code in multiple concurrent streams that are asynchronous. Thanks to this, a great time saving can be obtained, as show in figure 2.9.

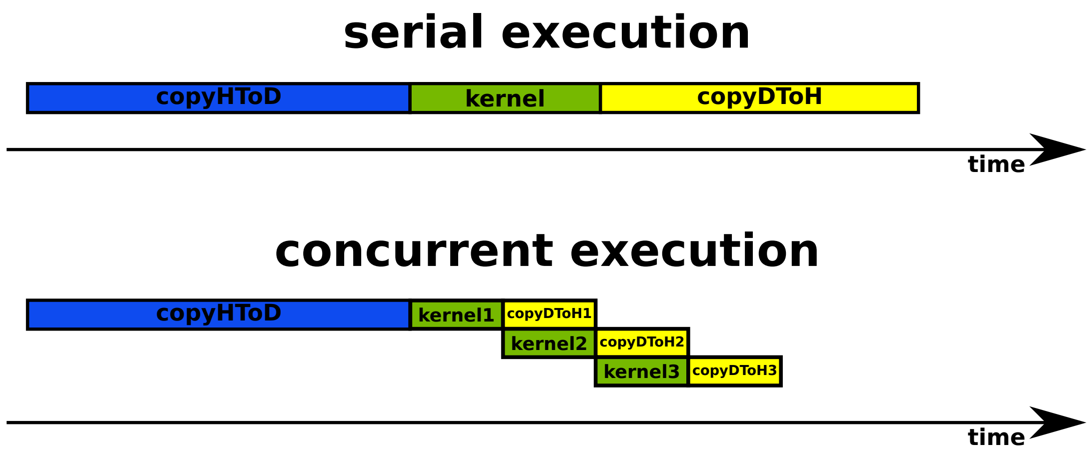


Figure 2.9: Serial vs concurrency execution.

Finally, the apposite declaration specifier `__shared__` has to be used to manage shared memory. Shared memory is usually declared dynamically in this form:

```
extern __shared__ variable_type myVariable;
```

and this structure is used when the amount of memory is unknown at compile time. In this case shared memory size is specified as third option in the kernel, giving the allocation size per block in bytes, and it is usually given by `sharedMem = sizeof(variable_type) * nThreads`. When data is shared between threads, one of the complex issues to manage is the synchronization between threads. As a matter of fact, threads in a block do not really run in parallel but the amount of resident units that can work simultaneously is limited by hardware specifications. To control this behaviour CUDA offer a built-in function, `_syncthreads`, that acts as a barrier and stops execution until all the threads in the same block have finished their activity.

2.4 Hardware used

Results obtained in chapter 5 have been obtained by means of three different computing systems, each one equipped with a different graphic processor unit:

- the NVIDIA GeForce 930MX (*GPU1*);
- the NVIDIA GeForce GTX 1660 SUPER (*GPU2*);
- the NVIDIA TITAN V (*GPU3*).

The NVIDIA GeForce 930MX is a entry level mobile GPU, developed for laptops and released on March 2016. It is based on Maxwell architecture (`sm_50`). It has 3 streaming multiprocessors operating at 0.9 GHz with 384 total CUDA cores, and it is equipped with 2 GB of DDR3 global memory. The NVIDIA GeForce GTX 1660 SUPER is a performance-segment graphics card released on October 2019. It has 1408 CUDA cores and 22 SMs and a 6 GB GDDR6 memory. Finally, the NVIDIA TITAN V is a top-class GPU launched in December 2017. Based on Volta architecture (`sm_70`), it mounts a 12 GB HBM2 memory and it

	GPU1	GPU2	GPU3
Microarchitecture	Maxwell	Touring	Volta
Number of SMs	3	22	80
Number of CUDA Cores	384	1408	5120
Global Memory [GB]	2	6	12
Memory bus [bit]	64	192	3072
Bandwidth [GB/s]	14.40	336	651.3
Memory Type	DDR3	GGDR6	HBM2
GPU max clock rate [MHz]	1020	1785	1455
Memory clock rate [MHz]	900	1750	848
Bus interface	PCIe 3.0 x8	PCIe 3.0 x16	PCIe 3.0 x16
Constant Memory [KB]	64	64	64
L2 Cache size [KB]	1024	1536	4500
Max shared memory per SM [KB]	64	96	96
Max shared memory per block [KB]	48	48	96
Registers available per thread	255	255	255
Registers available per block	64 K	64 K	64 K
Registers available per SM	64 K	64 K	64 K
Max number of threads per block	1024	1024	1024
Max resident blocks per SM	32	32	32
Max resident warps per SM	64	64	64
Max resident threads per SM	2048	2048	2048

Table 2.2: Technical specifications for each device.

is equipped with 80 SMs and 5120 CUDA cores. Main technical specification of the three devices are presented in table 2.2

The choice of using different GPUs with very different technical properties is related to the intention of underlining how the time advantage is strictly connected to the particular hardware used. As it will be cleared in chapter 5, GPGPU can allow a huge time saving if the simulation is executed using a powerful hardware; on the contrary, low-end GPUs does not allow to obtain a sensible decreasing in computational time. For this reason, *GPU1* represents an entry-level device that can give little improvements in terms of time taken for the simulation but has poor performances for complex calculation. *GPU2* is a little more powerful: most of simulations executed on this device take less time than executing it on CPU. Nevertheless, it has not the best hardware that the market can offer, so the speed improvements that can be achieved are not the greatest. Finally, *GPU3* represents the flagship device in terms of technical specification; thus, it gives important results in terms of speed-up. Figure 2.10 shows a performance comparison of the graphical processor units based on *techpowerup.com* review data. Devices were tested at the same condition and they are also compared with NVIDIA GeForce RTX 3090, the most powerful GPU present on the market up today.

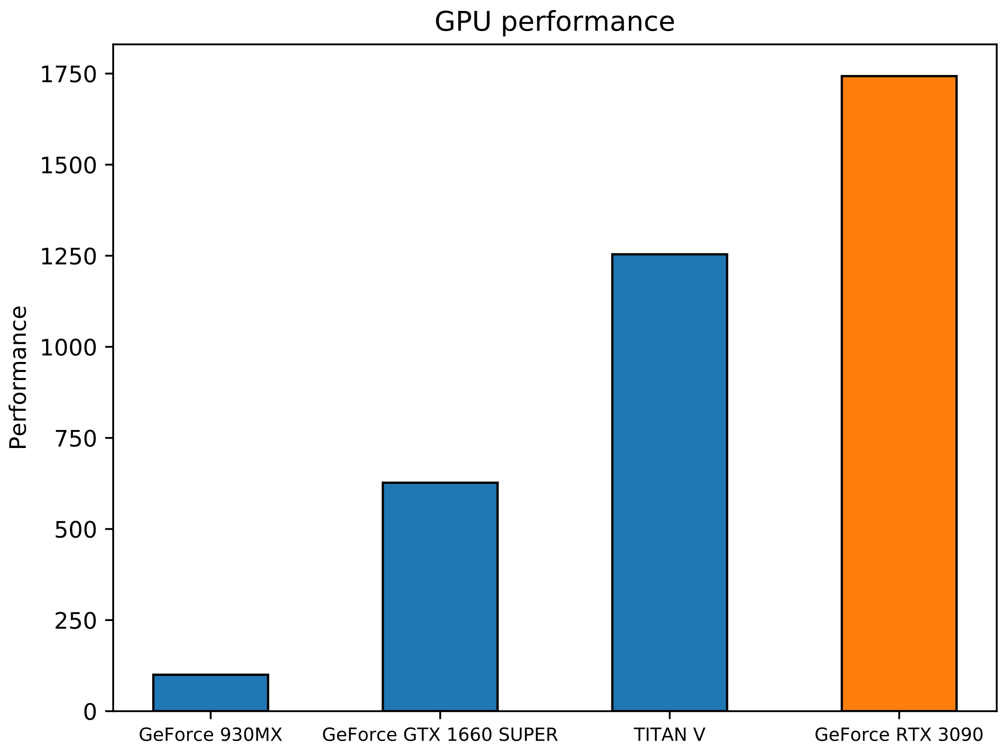


Figure 2.10: Performance of used GPUs based on *techpowerup.com* review data "Performance summary".

Chapter 3

Physics of reacting flows

Physics of reacting flows motion is now presented. First, the classical equations of fluid dynamics are described. In particular, the attention is focused to the role of the chemistry that increases the complexity of the system, adding further relations to solve. Furthermore, chemistry is ruled by a set of ordinary differential equations that has to be solved with numerical integration methods. Numerical solution of initial value problems will be presented, with particular attention to the Runge-Kutta Cash-Karp method that will be used in the implementation of the code.

3.1 Governing equation for fluid motion

Navier-Stokes equations describe the motion of a fluid. From a mathematical point of view, they can be seen as a system of 5 partial differential equations in 5 unknowns that expresses the conservation of three quantities: mass, momentum, total energy. Usually, they are derived by taking a control volume immersed in the fluid and by making a balance of the interested quantities. The form of the conservation equation for the general scalar intensive variable ϕ, used in OpenFOAM, can be:

$$\frac{\partial}{\partial t}\int_\Omega \rho\phi \, d\Omega + \int_\Omega \rho\phi(\mathbf{U} - \mathbf{U}_b) \cdot \mathbf{n} \, d\partial\Omega = \int_\Omega \Lambda\nabla\phi \cdot \mathbf{n} \, d\partial\Omega + \sum f_\phi \tag{3.1}$$

that is expressed in a integral form (it can be expressed also in a differential form). The first two terms represent a material derivative of the scalar quantity $\frac{D\phi}{Dt}$: the first term is the temporal variation of the conserved quantity in the control volume over the time, the second is the convection term that represents the motion of the quantity due to the fluid motion $\mathbf{U}$. In the convective part, there is also the contribution due to a possible mesh motion $\mathbf{U}_b$. The third part is the diffusive term that represents motion of the fluid by diffusion. The fourth part includes source and sinks, surface and volume forces and other mechanisms of transport that differ from the other two.

The Navier-Stokes equations will be now presented in differential form.

Mass conservation

The conservation of mass (called also the continuity equation) is ruled by:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{U}) = 0 \tag{3.2}$$

It can be derived from equation 3.1 by using $\phi = 1$. The mass rate of change in time into the control volume is equal to the flux of mass entering and exiting the control volume since mass is conserved.

When compressibility effects can be neglected, thus density can be considered constant, the equation reduce to the simpler form $\nabla \cdot \mathbf{U} = 0$.

Momentum conservation

The conservation of momentum can be expressed as:

$$\frac{\partial (\rho \mathbf{U})}{\partial t} + \nabla \cdot (\rho \mathbf{U} \otimes \mathbf{U}) = (-\nabla p + \frac{2}{3}\mu \nabla^2 \mathbf{U}) \cdot \mathbf{I} + \nabla \cdot [\mu(\nabla \mathbf{U} + (\nabla \mathbf{U})^T)] + \rho \mathbf{g} + S \tag{3.3}$$

that can be interpreted as the general conservation equation 3.1 by using $\phi = \mathbf{U}$.

Here, the first 3 terms on the right hand side of the balance equation represent the divergence of the shear stress tensor $\nabla \cdot \mathbf{T}$ that are the surface forces acting on the fluid. The shear stress tensor represents the molecular rate of transport of momentum and for a newtonian fluid it is written as:

$$\mathbf{T} = (-p + \frac{2}{3}\mu\nabla\mathbf{U})\mathbf{I} + \mu[\nabla\mathbf{U} + (\nabla\mathbf{U})^T] = -p\mathbf{I} + \tau \tag{3.4}$$

where $\mathbf{I}$ is the identity tensor and μ is the cinematic viscosity. The shear tensor can be interpreted as the sum of a pressure contribution and a viscous part contribution τ. Finally, the term $\rho\mathbf{g}$ represents volume forces due to gravity (i.e. buoyancy).

Energy conservation

Conservation of energy can be written in several forms. In the classical form, the total energy is conserved:

$$\frac{\partial(\rho e)}{\partial t} + \nabla\cdot(\rho e\mathbf{U}) + \frac{\partial(\rho K)}{\partial t} + \nabla\cdot(\rho K\mathbf{U}) = -\nabla\cdot(\mathbf{T}\cdot\mathbf{U}) + \nabla\cdot\mathbf{q} + \rho r + \rho\mathbf{g}\cdot\mathbf{U} \tag{3.5}$$

that is the general conservation equation 3.1 with $\phi = e + K$, where e is the internal energy (thermodynamic term) and $K = \frac{1}{2}(\mathbf{U}\cdot\mathbf{U})^2$ is the kinetic energy (mechanical term). Terms on the right hand side represent respectively the thermodynamic power and the mechanical power applied on the control volume.

In case of incompressible flows the mass and momentum balance equations are decoupled from the energy equation, so they can be solved to obtain velocity and pressure fields that are used in the last equation to get the total energy.

In OpenFOAM the energy equation can be implemented in another form as conservation of total entalphy:

$$\frac{\partial(\rho h)}{\partial t} + \nabla\cdot(\rho h\mathbf{U}) + \frac{\partial(\rho K)}{\partial t} + \nabla\cdot(\rho K\mathbf{U}) - \frac{\partial p}{\partial t} = -\nabla\cdot(\mathbf{T}\cdot\mathbf{U}) + \nabla\cdot\mathbf{q} + \rho r + \rho\mathbf{g}\cdot\mathbf{U} \tag{3.6}$$

remembering the relation between entalphy and internal energy:

$$h = e + \frac{p}{\rho} \tag{3.7}$$

In OpenFOAM the energy equation is not in complete form, but some approximations are present: mechanical sources ($\nabla\cdot(\tau\mathbf{U})$, $\rho\mathbf{g}\cdot\mathbf{U}$) are neglected; a heat flux of the form $q = -\alpha\nabla e$ is assumed where the thermal diffusivity α is the sum of a laminar and turbulent contribution; the thermal source term ρr is modeled considering the specific solver.

3.2 Governing equation for reacting flow motion

Chemical thermodynamics of reacting flows

If the flow involved is a reacting flow (e.g. in combustion phenomena), chemistry has to be considered when solving the Navier-Stokes equations. In fact, a multicomponent fluid contains several chemical species that can interact one with the other producing new species and changing the thermal state. Usually combustion happens after reaching an energetic state called the activation energy; the system is then activated and the reaction can take place.

First of all, it is necessary to quantify the amount of each species when working with mixture. Since the mass is conserved by equation 3.2, the most appropriate quantity to describe the conservation of the species is the **mass fraction**, defined as:

$$Y_k = \frac{\rho_k}{\rho} \tag{3.8}$$

where ρ_k is the density of the k-species and ρ the total density. It is evident that:

$$\sum_{k=0}^{K} Y_k = 1 \tag{3.9}$$

since the summation of partial density is equal to the total density. Chemical reactions are usually described with moles instead of mass: moles of different species react together obtaining moles of products. Thus, the **mole fraction** X_k is also used to describe mixture, defined as:

$$X_k = Y_k \frac{W}{W_k} \tag{3.10}$$

where W_k is the molecular weight of species k and W is the mean molecular weight of the mixture. Another way to describe species composition in a mixture is by using the **molar concentration** $[X_k]$:

$$[X_K] = \rho \frac{Y_k}{W_k} = \rho \frac{X_k}{W} \tag{3.11}$$

For a mixture with N species a partial ideal gas law can be defined for species k:

$$p_k = \rho_k \frac{RT}{W_k} \tag{3.12}$$

where p_k is the partial pressure of k-species in the mixture and $p = \sum_k^N p_k$. The ideal gas law of the mixture $p = \rho \frac{RT}{W}$ can be derived by summing equation 3.12 for all the N components.

For chemical reacting flows, there are several ways to define energy and enthalpy:

- **sensible energy**: $e_{sk} = \int_{T_0}^{T} c_{vk}\, dT - R\frac{T_0}{W_k}$;
- **sensible enthalpy**: $h_{sk} = \int_{T_0}^{T} c_{pk}\, dT$;
- **sensible and chemical energy**: $e_k = e_{sk} + \Delta h^0_{f,k}$;
- **sensible and chemical enthalpy**: $h_k = h_{sk} + \Delta h^0_{f,k}$;

where $\Delta h^0_{f,k}$ is the mass formation enthalpy of species k at T_0 (usually it is taken at standard reference $T_0 = 298.15K$) that can be written also in molar values $\Delta h^{0,m}_{f,k}$ by multiplying it it by W_k. C_{vk} and c_{pk} are respectively the heat capacity at constant volume and constant pressure. The heat capacity at constant pressure of the mixture can be derived by summing the heat capacities of each species weighted with their own mass fractions. So C_p depends on the species composition and the temperature: as a matter of fact, for most hydrocarbon/air flames where the nitrogen is the most important species, it can be considered constant as it does not change excessively while T changes.

Chemical kinetics mechanism

Considering N species, a reaction is typically written with the following formalism:

$$\sum_{k=0}^{N} \nu'_k M_k \rightleftharpoons \sum_{k=0}^{k} \nu''_k M_k \tag{3.13}$$

where ν'_k and ν''_k are the i^{th} stochiometric coefficients of the reactants and the products respectively and M_k is the k^{th} chemical species of the mixture. Thus, a general reaction can be written as:

$$A + B \rightleftharpoons C + D \tag{3.14}$$

where A,B,C and D are generic species. A reaction can be **reversible** (as in equation 3.14) when reactants can become products and vice versa, coexisting in chemical equilibrium, while in a **irreversible** reaction the equilibrium is tended to reactants or to products. Combustion phenomena are a valid example of irreversible reactions, since their main products (i.e. H_2O and CO_2) do not react with each other.

Usually natural phenomena are not ruled by a single reaction but by a complex chain of multiple elementary reactions. It is clear that each reaction occurs at different rate that mainly varies the concentration of the species involved. So the rate of change of a specie is called **chemical reaction rate** $\dot{\omega}$ and it is defined as the time derivative of the specie concentration. Clearly, a negative reaction rate will indicate a species consumption (referred to reactant) while a positive reaction rate will indicate a specie production (referred to products). It has been discovered experimentally that reaction rate depends on five conditions of the system: concentration of the chemical species, temperature, pressure, presence of catalyst or inhibitor and radiation [18]. In general:

$$\dot{\omega} = \frac{1}{\pm\nu_i}\frac{d[M_k]}{dt} \tag{3.15}$$

defines the **law of mass action**, that was confirmed by several empirical observations.

Considering for instance a reversible reaction:

$$A + B \overset{k_f,k_r}{\rightleftharpoons} C \tag{3.16}$$

the rate of change of the species can be written as:

$$\frac{d[A]}{dt} = \frac{d[B]}{dt} = -k_f[A][B] + kr[C] \tag{3.17}$$

$$\frac{d[C]}{dt} = -k_r[C] + k_f[A][B] \tag{3.18}$$

This set of equations that can be built up for a multi-species flow represents a system of n-Ordinary Differential Equations (ODE) that has to be solved in order to find the n-reaction rate (one for each species). k_f and k_r are, respectively, the **forward** and **reverse reaction rate constants** that indicate the relation between the molar concentration and the reaction rate.

More general, the rate of production of the k^{th} species can be written as linear combination of the rate-of-progress variables of the reactions $\dot{\omega}_i$:

$$\dot{\omega}_k = \sum_{i=0}^{I} \nu_{ki}\omega_i = \sum_{i=0}^{I} (\nu_{ki}^{'} - \nu_{ki}^{''})\omega_i \tag{3.19}$$

The rate-of-progress variable for the i^{th} reaction is defined as:

$$\dot{\omega}_i = k_{f_i} \prod_{k=0}^{N} [X_k]^{\nu_{ki}^{'}} - k_{r_i} \prod_{k=0}^{N} [X_k]^{\nu_{ki}^{''}} \tag{3.20}$$

depending on the forward and reverse rate constants of the i^{th} reaction. The former value can be determined experimentally or using the Arrhenius Law, that gives a relation with the temperature:

$$k_f = AT_a e^{-\frac{E_a}{R_u T}} \tag{3.21}$$

where AT_a is the collision frequency (that describes the rate of collision between two or more molecules in a define volume) and E_a is the activation energy. The reverse reaction rate constant is derived from the forward reaction rate as follows:

$$k_r = \frac{k_f}{K_c} \tag{3.22}$$

where K_c is the equilibrium constant.

Mass conservation of species

Equations 3.19 represent a set of N ordinary differential equations, one for each species. The concentration of the species involved changes due to the progress in the reaction.

Even though the conservation of mass of the mixture is already enforced through 3.2, conservation of mass of each species in the mixture has to be applied due to the Lavoisier principle (*Matter is neither created or destroyed*).

The additional N-equations can be written in conservation form, so the k^{th} species relation is:

$$\frac{\partial \rho Y_k}{\partial t} + \nabla \cdot (\rho(\mathbf{U} + \mathbf{V}_k)Y_ik) = \dot{\omega} \tag{3.23}$$

for $i = 1, ..., N$. $\mathbf{V}_k$ is the diffusion velocity of the species k and accounts for diffusion phenomena. The source/sink term is represented by the reaction rate $\dot{\omega}$. Since combustion does not generate mass, total mass conservation is still valid. Summing the N-mass conservation equations of the species and assuming that $\sum_{i=1}^{N} \dot{\omega}_i = 0$, one obtains:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{U}) = -\nabla \cdot (\rho \sum_{i=1}^{N} \mathbf{V}_i Y_i) \tag{3.24}$$

Enforcing the right hand side of the equation to be zero as necessary condition to conserve the mass, equation (3.2) is obtained.

3.3 Numerical treatment of chemical ordinary differential equations

As mentioned above, fluid dynamics problems involving reacting mixture require the solution of a set of coupled ODEs:

$$\frac{d[X_i]}{dt} = \dot{\omega}_i \tag{3.25}$$

$$\dot{Q} = -\frac{1}{\rho}\sum_{i=1}^{N_S} h_i\dot{\omega}_i \tag{3.26}$$

Substituting the definition of species concentration in eq. 3.25 and remembering the definition of entalphy in eq. 3.26, the two ODEs can be rewritten in a more general form:

$$\frac{dY_i}{dt} = \frac{\dot{\omega}_i W_i}{\rho} \tag{3.27}$$

$$\dot{T} = -\frac{1}{c_p}\sum_{i=1}^{N_S} h_i\dot{\omega}_i \tag{3.28}$$

Typically species mass fractions and the temperature are combined together in a vector $u(t)$ and the system of differential equations becomes:

$$\begin{aligned} \dot{\mathbf{u}}(t) &= f(\mathbf{u}(t), \mathbf{q}) \\ \mathbf{u}(t_0) &= \mathbf{u}_0(\mathbf{q}) \end{aligned} \tag{3.29}$$

that is a initial value problem (IVP), with initial condition $\mathbf{u}_0$ and non integrated parameters $\mathbf{q}$. Ordinary differential equations can be solved numerically.

A first classification of the numerical methods for first-order IVPs can be made considering the number of steps used for advancing the solution. **Single-step methods** (like Euler's method) require only the solution at previous time t to compute the value at time $t + 1$. **Runge-Kutta methods** take some intermediate steps to compute the solution at time $t + 1$, but the intermediate steps are then discarded. On the contrary, **multi-step methods** use intermediate solutions for the next steps, gaining accuracy and efficiency. Single steps methods usually consider as Runge-Kutta methods of first order.

Another classification can be made regarding the approach used for computing the solution, talking about **explicit** and **implicit** methods. Explicit methods use the state of the system at current time t_n to calculate the state at a later time t_{n+1} while by means of implicit methods one has to solve a system of equations with both the current and the following time state. The former have a simpler mathematical structure: small time steps must be used in order to achieve convergence and great accuracy of results. Each time step have a low computational cost. They give high quality results with smaller number of computational cells combined with few chemical species and reactions. On the other side, implicit methods can use larger time steps and so less chemical time steps per fluid dynamics step, but increasing the cost per each one because they are based on the calculation of a Jacobian Matrix. The choice of the method is strictly dictated by the particular case. Anyway the best choice is the one that gives the most accurate results at smaller computational time.

One of the biggest problem is represented by **numerical stiffness**. Stiffness usually arises when a variable of the ODEs system changes rapidly during time, for instance in a transient state [19]. Several definitions of stiffness are present in literature. A common interpretation states as follows (Spijker, [20]):

For most explicit methods, stiffness occurs if the largest step size h_n^* guaranteeing numerical stability is much smaller than the largest step h_n for which the local discretization error is still sufficiently small (in norm), i.e., $h_n^* \ll h_n$.

Another simpler definition, but more general, states as follows:

Initial value problems are stiff if they are (exceedingly) difficult to solve by ordinary, explicit step-by-step methods, whereas certain implicit methods perform quite well

As can be seen, explicit methods can not work as well as implicit ones for stiff cases provided a large chemical sub-time step but they usually require smaller time steps that roughly increase the computational cost. Chemical kinetics can reach great sources of stiffness, as shown by Curtiss and Hirschfelder [21].

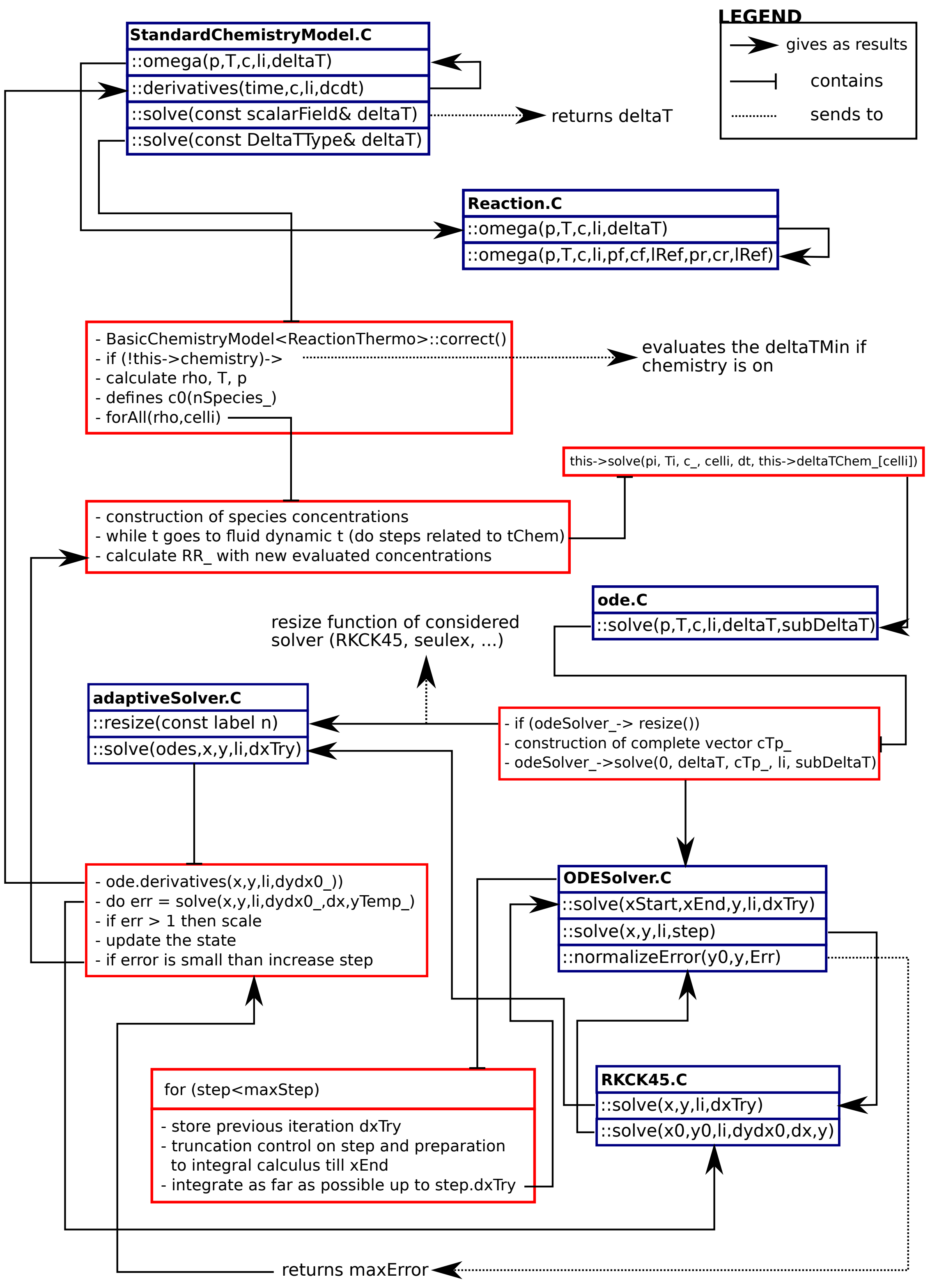

LEGEND
gives as results
contains
sends to
StandardChemistryModel.C
::omega(p,T,c,li,deltaT)
::derivatives(time,c,li,dcdt)
::solve(const scalarField& deltaT)
::solve(const DeltaTType& deltaT)
returns deltaT
Reaction.C
::omega(p,T,c,li,deltaT)
::omega(p,T,c,li,pf,cf,lRef,pr,cr,rRef)
- BasicChemistryModel<ReactionThermo>::correct()
- if (!this->chemistry)->
- calculate rho, T, p
- defines c0(nSpecies_)
- forAll(rho,celli)
evaluates the deltaTMin if chemistry is on
this->solve(pi, Ti, c_, celli, dt, this->deltaTChem_[celli])
- construction of species concentrations
- while t goes to fluid dynamic t (do steps related to tChem)
- calculate RR_ with new evaluated concentrations
ode.C
::solve(p,T,c,li,deltaT,subDeltaT)
resize function of considered solver (RKCK45, seulex, ...)
adaptiveSolver.C
::resize(const label n)
::solve(odes,x,y,li,dxTry)
- if (odeSolver_-> resize())
- construction of complete vector cTp_
- odeSolver_->solve(0, deltaT, cTp_, li, subDeltaT)
- ode.derivatives(x,y,li,dydx0_))
- do err = solve(x,y,li,dydx0_,dx,yTemp_)
- if err > 1 then scale
- update the state
- if error is small than increase step
ODESolver.C
::solve(xStart,xEnd,y,li,dxTry)
::solve(x,y,li,step)
::normalizeError(y0,y,Err)
for (step<maxStep)
- store previous iteration dxTry
- truncation control on step and preparation to integral calculus till xEnd
- integrate as far as possible up to step.dxTry
RKCK45.C
::solve(x,y,li,dxTry)
::solve(x0,y0,li,dydx0,dx,y)
returns maxError

Figure 3.1: RKCK45 OpenFOAM scheme [6]

.

OpenFOAM provides a set of different integration methods. In particular, in this work the explicit `RKCK45` and the implicit `seulex` have been used for testing. Moreover, `RKCK45` has been chosen to be rewritten in CUDA-C language, in order to achieve GPU parallelization. This choice has been made because of the specific architecture of the GPU: the explicit method can be parallelized, while the implicit method can not without introducing instabilities. In section 4.2 we will give a detailed explanation behind the adopted solution. `RKCK45` method implementation in OpenFOAM is presented in figure 3.1 and is described accurately in appendix B.

3.4 The Runge-Kutta Cash-Karp method

The Runge-Kutta Cash-Karp [22] is a Runge-Kutta explicit method developed by Cash and Karp in 1990. A Runge-Kutta method has the general form:

$$y_{i+1} = y_i + \phi(x_i, y_i, h)h \tag{3.30}$$

where the function ϕ is called *increment function* and it can be interpreted as the slope over the interval $[i, i+1]$. This function is written as follows:

$$\phi = \sum_{j=1}^{n} a_j k_j \tag{3.31}$$

where a_j are constants and k_j are evaluated with a recurrence relation:

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + p_1 h, y_i + q_{1,1} k_1 h) \\ &\dots \\ k_n &= f(x_i + p_{n-1} h, y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + \dots + q_{n-1,n-1} k_{n-1} h) \end{aligned} \tag{3.32}$$

where p_j and $q_{j,j}$ are constant and they are arrange in a table call *Butcher Tableau*. A Runge-Kutta method can be of various orders, that is given by n.

RKCK45 is a fourth-fifth order method: it uses six function evaluations to calculate fourth and fifth-order solutions. Butcher tableau for Runge-Kutta Cash-Karp method is:

0						
1/5	1/5					
3/10	3/40	9/40				
3/5	3/10	-9/10	6/5			
1	-11/54	5/2	-70/27	35/27		
7/8	1631/55296	175/512	575/13824	44275/110592	253/4096	
	37/378	0	250/621	125/594	0	512/1771
	2825/27648	0	18575/48384	13525/55296	277/14336	1/4

Table 3.1: Runge-Kutta Cash-Karp Butcher tableau.

The two solutions are then subtracted together and the difference is taken as the error on the fourth-order solution:

$$\Delta_{n+1} = y_{n+1} - y^*_{n+1} \tag{3.33}$$

and it is compared with a desired accuracy Δ_0. When the error is higher than the desire accuracy, the algorithm rejects the current solution and calculates another step size. When the error satisfies the desired accuracy, the algorithm accepts the step and calculates the next time step. For this reason RKCK45 is an *adaptive* method: the step size is not constant but changes due to great variation of the solution. In fact, when the slope becomes very steep, a shorter step size better catches the behaviour of the solution.

Chapter 4

Code implementation

The particular GPU structure gives the possibility to solve the chemical kinetics problem without impacting too much on the computational resources thanks to the massive parallelism that can be achieved. Benefits of GPGPU have been presented so far, as well as the physical problem of the combustion. In this chapter the adopted implementation is presented and a general description of the code is offered.

4.1 Combustion solvers in OpenFOAM

OpenFOAM provides multiple combustion solvers (`chemFoam`, `coldEngineFoam`, `fireFoam`, ...) and most of them are based on different combustion models. This work is focused on two solvers only that work through the resolution of chemistry kinetics ODEs:

- `chemFoam`, a single-cell solver. It is normally used to test the behaviour of untested chemical reactions. Before simulating a complete test case, chemistry is evaluated without considering other parameters (e.g. geometry). Similar software are Chemkin and Cantera;

- `reactingFoam`, a multi-cell solver. It is the most used OpenFOAM solver for combustion fluid dynamics. It does not extend its capability up to the spray treatment, as it is implemented in `sprayFoam` solver.

The first solver is a simpler version of the second one (figure 4.1). A single cell reactor is simulated and the fluid dynamics is not considered at all. Only chemical ODEs are solved. Species mass fractions and temperature are update at each chemical time step, until the end time is reached.

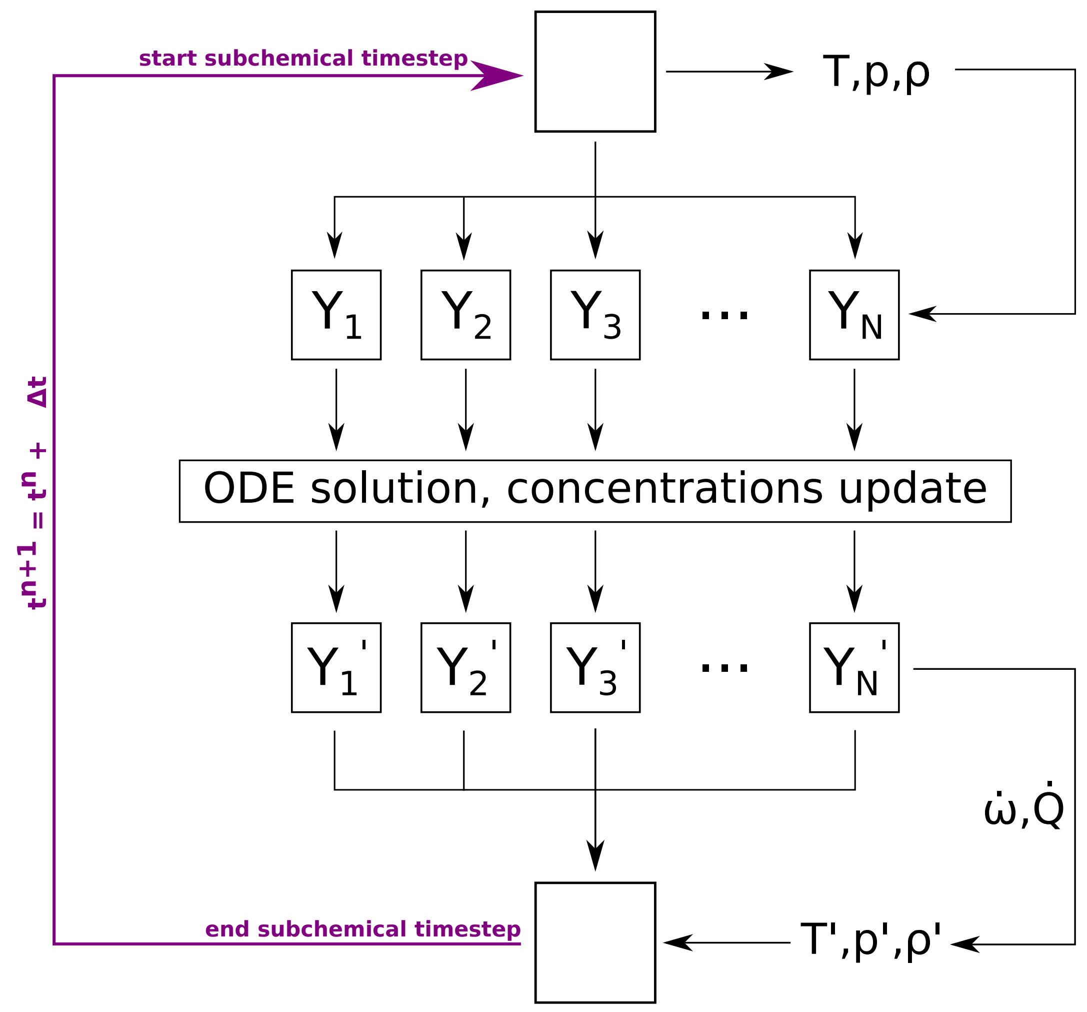


Figure 4.1: `chemFoam` structure.

In `reactingFoam` the algorithm becomes more complex. The solver uses the PIMPLE algorithm, that is a combination of SIMPLE and PISO algorithm. Chemical kinetics ODEs are solved for each mesh cell thanks to a `forAll(rho, celli)` cycle, that is contained in the `StandardChemistryModel.C` file. Every cell is independent from the others, so a set of n-cells independent ordinary differential equations is solved, each one with its own initial conditions. Furthermore, the solution is obtained from the current simulation time t to $t+\Delta t$, where Δt is the fluid dynamics time step. When the chemistry is solved and the reaction rates are computed, concentrations of the species and temperature are updated in adiabatic, fixed-volume condition ($\frac{dp}{dt} = 0$). Reaction rates and chemical heat source are then used in the conservation of mass species (equation 3.23) and energy (equation 3.5) and, at the end, fluid dynamics quantities are updated.

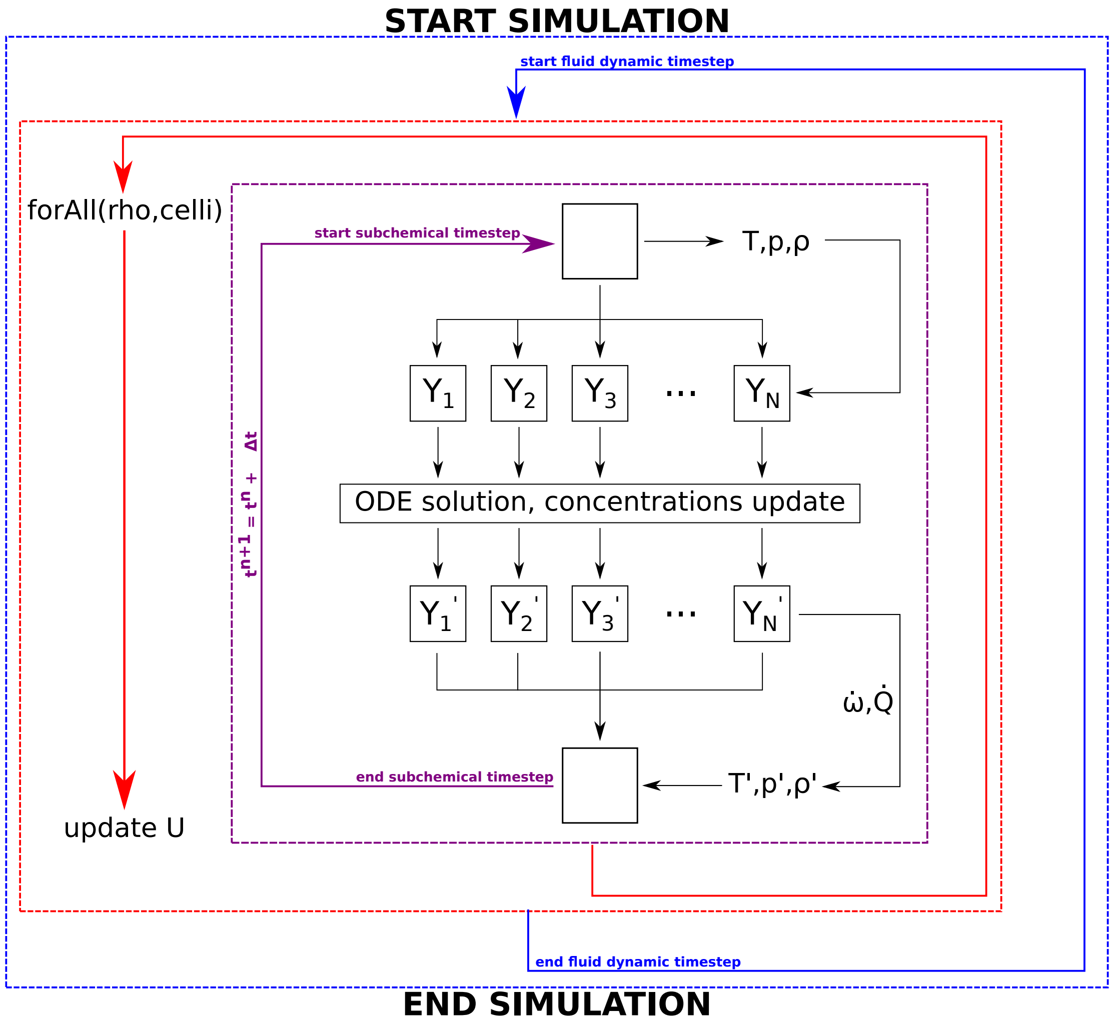


Figure 4.2: `reactingFoam` structure.

For a more detailed explanation of the implementation of these two solvers, see appendix A.

4.2 GPU chemistry model

Operations on chemistry in standard method on CPU are performed in a serial way. Giving initial conditions all ODEs are solved in the aforementioned `forAll(rho, celli)` cycle that can be very disadvantageous when solving a huge number of cells. Added to this, complexity of the reactions in terms of number of species and reaction chain affects the total computational time. These problems can be solved if all these operations could be done in parallel rather than one after another.

Two different strategies are possible to achieve parallelism [3]. The **one-thread** implementation is based on the solution of the differential equation on a single thread, that will perform also all the operations to get the chemical reaction rates for each species. This approach permits to solve a large number of ODEs at the same time, depends on the available hardware. However, it does not use shared memory but global memory only. The great amount of data per each ODEs that has to be used for computation cannot be stored in the limited shared memory. At the other end, in the **one-block** implementation the differential equation is solved among the threads in a single block, so a GPU computes several parallel blocks on each SM. In this case shared memory is largely used to let intra-block cooperation.

A sort of one-thread approach was chosen to implement the code. Every thread-block is associated with a mesh cell. The number of CUDA blocks to be used in the kernel function is set in the constructor of `GPUChemistryModel.C` considering the limitations in terms of the GPU compute capability and the hardware. The difference between single and multi-cell solver is processed by the use of a conditional statement on the variable `meshDimension_`. When this value is not equal to 1, the number of block is chosen considering memory availability and hardware limitations. On the other hand, each thread represents a single species so each of them is represented by an index thanks to the CUDA variable `threadIdx.x`. This implementation gives the possibility to use the limited amount of shared memory to perform computation among the species to solve the chemical ODEs. The large amount of data is stored in global memory, thus limitations on shared memory are reached only for extremely complex reactions and high number of species, while for simpler reactions this does not cause particular issues.

The implementation adopted is now shortly described. In addition to the chemistry model available on the last OpenFOAM version, the new *GPUChemistryModel.C* (figure 4.3) has been attached as an extern OpenFOAM library. `GPUChemistryModel` inherits the class `StandardChemistryModel` and rewrites the functions that the latter provides to adapt them for parallel computation.

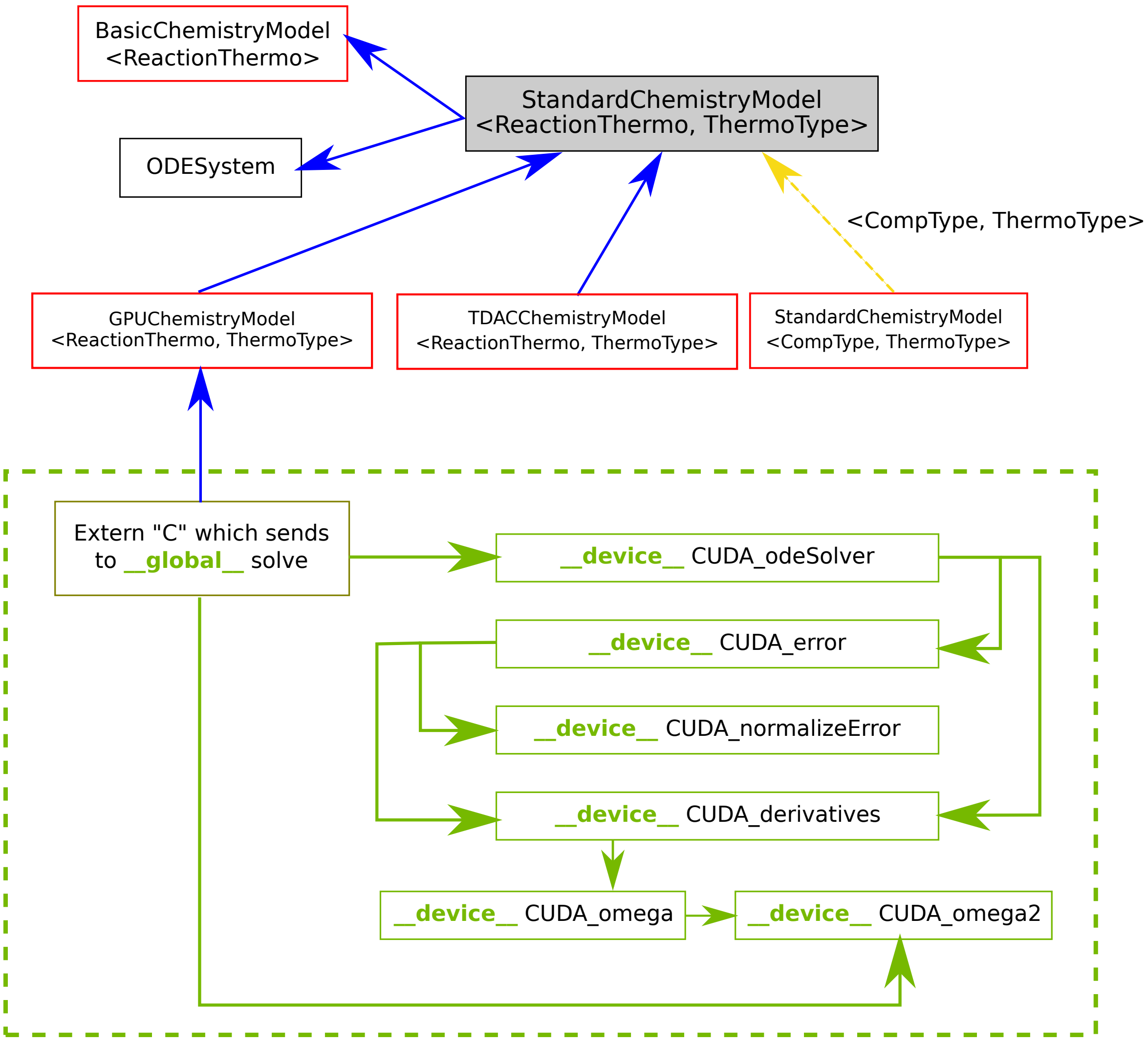

BasicChemistryModel <ReactionThermo>
ODESystem
StandardChemistryModel <ReactionThermo, ThermoType>
<CompType, ThermoType>
GPUChemistryModel <ReactionThermo, ThermoType>
TDACChemistryModel <ReactionThermo, ThermoType>
StandardChemistryModel <CompType, ThermoType>
Extern "C" which sends to __global__ solve
__device__ CUDA_odeSolver
__device__ CUDA_error
__device__ CUDA_normalizeError
__device__ CUDA_derivatives
__device__ CUDA_omega
__device__ CUDA_omega2

Figure 4.3: *GPUChemistry* model structure.

This adds a new `basicChemistryModel` type that defines a new combination method/solver, that has to be defined in `constant/chemistryProperties` as:

```
solver          gpu;
method          GPU;
```

This file defines a new `solve` function that overwrites the standard `solve` function and calls for particular `extern "C"` functions, that are necessary to compile extern CUDA "*.cu*" file on OpenFOAM. Data taken from thermophysical properties are stored in one dimensional arrays thanks to CUDA template class `thrust`. This particular one-dimensional structure is necessary due to indexing offered by CUDA: as shown in chapter 2, it is practice in CUDA implementation to create a global index `tid` that takes in account of the possibility to parallelize the code, so a linear array can be managed more easily than a multi-dimensional array. The class `thrust` is based on `C++` Standard Template Library (STL) and gives the possibility to construct vectors of data in a simple way like template class `vector` in `C++` (it shares together some objects and functions). Data are stored in CPU memory `host_vector` and then are copied into device global memory thanks to a similar container named `device_vector`. The construction is developed thanks to a classical iterative `for` cycle in CPU. As an illustration, the stoichiometric coefficients are referred for each species in each reaction, so they need a double species-reaction `for` cycle; on the contrary, the temperature varies cell by cell, so a different vector is needed.

At the end of this script, a `this->solve()` is present, sending on *gpu.C* file, where another solve function sends to an extern CUDA file. Here the `extern "C"` function is defined, sending to the `__global__` function where the main computation takes place. The kernel function `globalCudaOdeSolver` is launched. Considering the one-thread implementation adopted, the kernel is invoked as:

```
globalCudaOdeSolve<<<nCells,nSpecies,nSpecies*sizeof(double)>>>
        (arguments);
```

associating one block for one computational cell and one thread for one species. Shared memory is dynamically allocated because the number of species is not defined at compile time but depends on the particular test case. As mentioned in chapter 2, a typical CUDA program works following a common path:

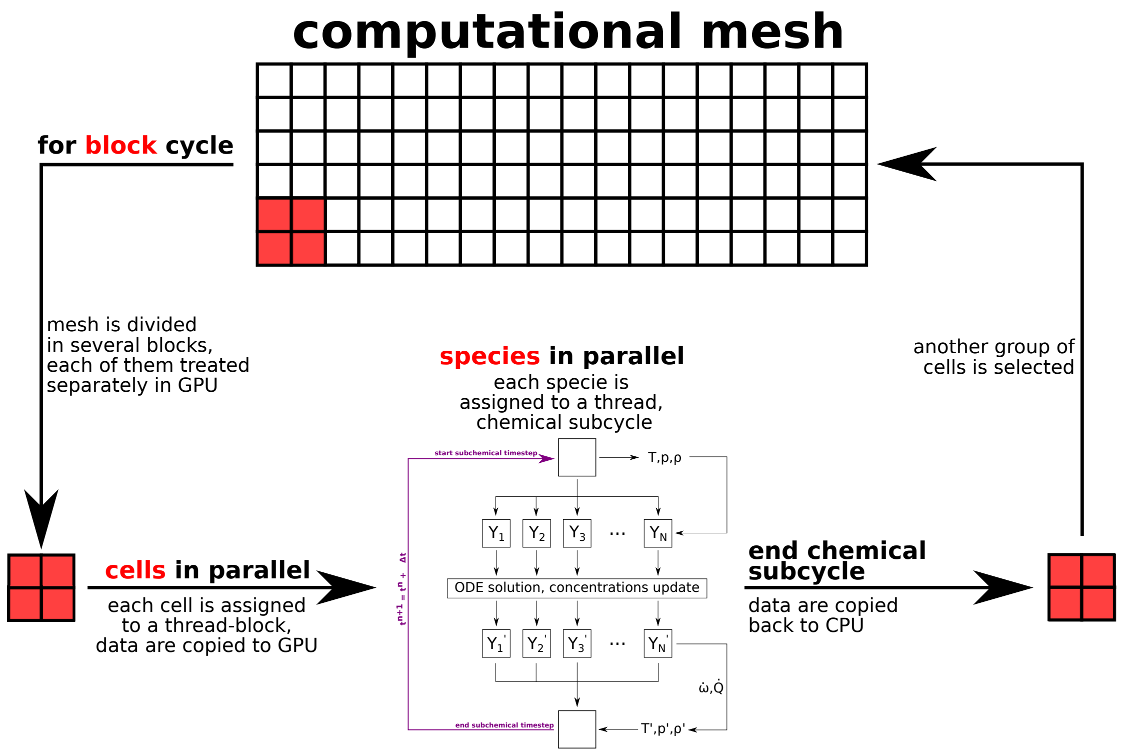

computational mesh
for block cycle
mesh is divided in several blocks, each of them treated separately in GPU
species in parallel
each specie is assigned to a thread, chemical subcycle
another group of cells is selected
start subchemical timestep
T,p,ρ
Y_1 Y_2 Y_3 ... Y_N
ODE solution, concentrations update
Y_1' Y_2' Y_3' ... Y_N'
$\dot{\omega},\dot{Q}$
T',p',ρ'
end subchemical timestep
$t^{n+1} = t^n + \Delta t$
cells in parallel
each cell is assigned to a thread-block, data are copied to GPU
end chemical subcycle
data are copied back to CPU

Figure 4.4: *GPUChemistry* structure.

memory is allocated in the device where data are copied from CPU and then the kernel is invoked thanks to pointers to the allocated data. Arrays of data are copied in global memory by using the copy assignment operator "=". Then, a pointer to the device vector is created and used as argument in kernel function. Operation of memory allocation and data copying are largely simplified thanks to the CUDA `thrust` template class. First of all, memory allocation is included in the copying operation from `host_vector` to `device_vector`. Second, the template class gives built-in functions for creation of pointers like `raw_pointer_cast`. A different construction has to be made for chemical arrays of data that contain informations for species or reactions for each cell: the entire vector was divided in sub-vectors (one per each cell); a pointer was assigned to every portion; a pointer to these pointers was created and used by the kernel.

In the kernel function the shared memory variable `shared` is defined dynamically with type `extern __shared__`. Then, the solution cycle starts after a conditional checking for each block: the reaction happens only if the temperature of the cell exceeds the reaction temperature. Finally, there is the call to `CUDA_odeSolver` where computation takes place in the device (thus beinf `__device__` type). This function is defined in an extern "*.cu*" file that in turn sends to other .cu file where other device functions are defined. The code is based on the `StandardChemistryModel` and `RKCK45` ODE explicit method adapted to work in parallel. Shared memory is used to allow species data communication among the threads for the solution of the chemical ODEs, while there is no intra-block communication and mesh cells perform the computation independently. After calculating the species mass fractions and temperature rate of variation over time, the error is computed in order to check the goodness of the results. If the normal error exceeds the unity, the algorithm finds a new chemical time step and the computation restarts until convergence is reached. The algorithm then progresses until chemical time has reached the fluid dynamics time step.

After the computation is over, the chemical reaction rate is computed and stored in a device vector. The `extern "C"` function ends with the operation of copying the chemical reaction rate and the chemical time step for each cell back on CPU. Usually a CUDA program requires operations of deallocating memory previously allocated in the device. Vectors created with thrust template class are deleted automatically when the function execution is stopped, freeing the memory. As might be expected, without the `cudaFree` functions a great amount of time can be saved. These functions would be necessary if the `thrust` library was not used in the program implementation. Finally, data calculated are sent back to OpenFOAM and used in the species and energy equations. This procedure is done several times concurrently depending on the number of cells computed in parallel.

The final version of *GPUChemistryModel* was obtained after a great work of code optimization. In fact, the original version worked well for single-cell cases only, thus the first changes were aimed to make the code working for multi-cell simulations. Since one of the goals of a GPGPU implementation is to achieve better performances in terms of computational speed, the code has been optimized in order to get better speed-ups while preserving the accuracy of the results. All these changes will not be presented in this dissertation. However, one needs to know that up to a 2x speed-up was achieved on the final code version with respect to the original one.

Finally, the implementation of the explicit Runge-Kutta method instead of an implicit one is soon explained. In a SIMT execution model (see section 2.3) all the blocks and threads do their tasks simultaneously without gathering. The only possibility to communicate is given by the shared memory, that allows communication amongst threads of the same block only. Thus, there is the necessity to decrease branching of the tasks and the requests of informations among different blocks to build a performing code. The `RKCK45` algorithm meets these requirements and it can be easily parallelizable. On the other side, an implicit integration method works totally different: it requires the construction of a Jacobian matrix and the solution of the system of equations must be found with an LU decomposition and a backward substitution. To build the corresponding triangular and diagonal matrices, each row of the matrix must know data of other rows and large amount of data has to be shared amongst the matrix parts. This matrix increases in shape with the mesh dimension, so in an hypothetical CUDA implementation each element should be stored in a block rather than in a thread (you should remember that the maximum number of threads in a block is limited to 1024). Since communication amongst block is not possible, data should be copied back and forth from global memory, causing a huge computational effort since its high latency and low throughput. One might think to construct only the Jacobian matrices of the chemical ODEs, one for each computational cell, and store them into the blocks since they are much smaller for simple chemistry mechanisms. Nevertheless, the solution of n different systems could increase the computational time needed instead of decreasing it. Furthermore, an implicit method is used for its extended stability as the linear system is completely solved considering every cell at the same time. Splitting the matrix could produce numerical errors in the solution and the method could become unstable.

Chapter 5

Results and discussion

In the last chapter the code has been analyzed exhaustively and all the main parts of the code have been described qualitatively. Nothing has been mentioned about the performance of *GPUChemistryModel* with respect to the original CPU version. In this chapter, CPU and CPU/GPU version of OpenFOAM chemical reaction solver will be studied with the aim of understanding whether or not the hybrid code can indeed decrease the computational time without any lost in performance. Consequently, results will be compared with CPU counterparts to validate the accuracy of the result. Then, speed up will be analyzed.

To do the best analysis, both (`chemFoam` and `reactingFoam`) solvers were used. In particular, the following simulations were performed:

- hydrogen combustion, single-cell;
- methane combustion (*GRI-Mech 3.0*), single-cell;
- simple methane combustion, multi-cell;
- simple syngas combustion (*CRECK syngas*), multi-cell;
- methane combustion (*GRI-Mech 3.0*), multi-cell.

Every test case is taken from OpenFOAM tutorial folder except the syngas mechanism. Therefore, GPU results were compared with data validated with OpenFOAM original chemistry model. Furthermore, chemical kinetics software Cantera [23] was used to better analyzed single-cell tutorial cases.

5.1 Single cell tutorials

As previously mentioned, the single-cell case consists in a single element mesh where fields are created from initial condition. There are no boundary conditions, since it is a pure virtual simulation that can not be represented experimentally. Fluid dynamics is not present at all, while temperature T and mass fractions of the chemical species Y_k are the only changing physical values.

5.1.1 Hydrogen combustion mechanism

The hydrogen/oxygen combustion is one of the most studied reactions system: in fact, the utilization of hydrogen as fuel is very important. First of all, it has a simple oxidation kinetics, with very fast mass diffusivity and low molecular weight. Second, this fuel is very powerful as clean burning alternative due to the fact that it does not produce CO_2 as pollutant [24]. Third, molecular hydrogen is easy to find in nature. It is widespread in space industry for rocket propulsion applications due to its high specific impulse, i.e. the thrust per unit bi-propellant flow rate [18, 25].

The hydrogen/oxygen chemical system is quite complex, although only two species are involved. This test case is based on a reduced mechanism, containing 5 elements, 10 species and 27 reactions. H_2/O_2 mechanism can be summarized by the simple reaction:

$$H_2 + O_2 \rightarrow 2H_2O + energy \tag{5.1}$$

The reduced kinetic system considered has low numerical stiffness, so an explicit integration method performs accurately. All the informations about chemistry, as well as the other `chemFoam` tutorials, are written in the Chemkin II format. Initial conditions and chemistry properties are presented in table 5.1 and 5.2.

T [K]	p [atm]	X_{H_2}	X_{O_2}	X_{N_2}
1000	2	1.00	1.00	3.76

Table 5.1: Initial data, hydrogen combustion mechanism.

Δt [s]	t_{end} [s]	$absTol$	$relTol$
1e-3	1e-7	1e-12	1e-1

Table 5.2: Simulation setup, hydrogen combustion mechanism.

Results are showed in figure 5.1 and 5.2. Here the evolution of temperature and mass fraction of the species involved is represented. The explicit CPU/GPU solution is compared with the explicit CPU counterpart, using the RKCK45 ODE scheme. The two results are then compared with the Cantera solution, assumed as reference. As it might been seen the accuracy of the results is preserved, confirming the goodness of the work. Nevertheless, the three curves do not perfectly overlap when the solution begins to vary strongly.

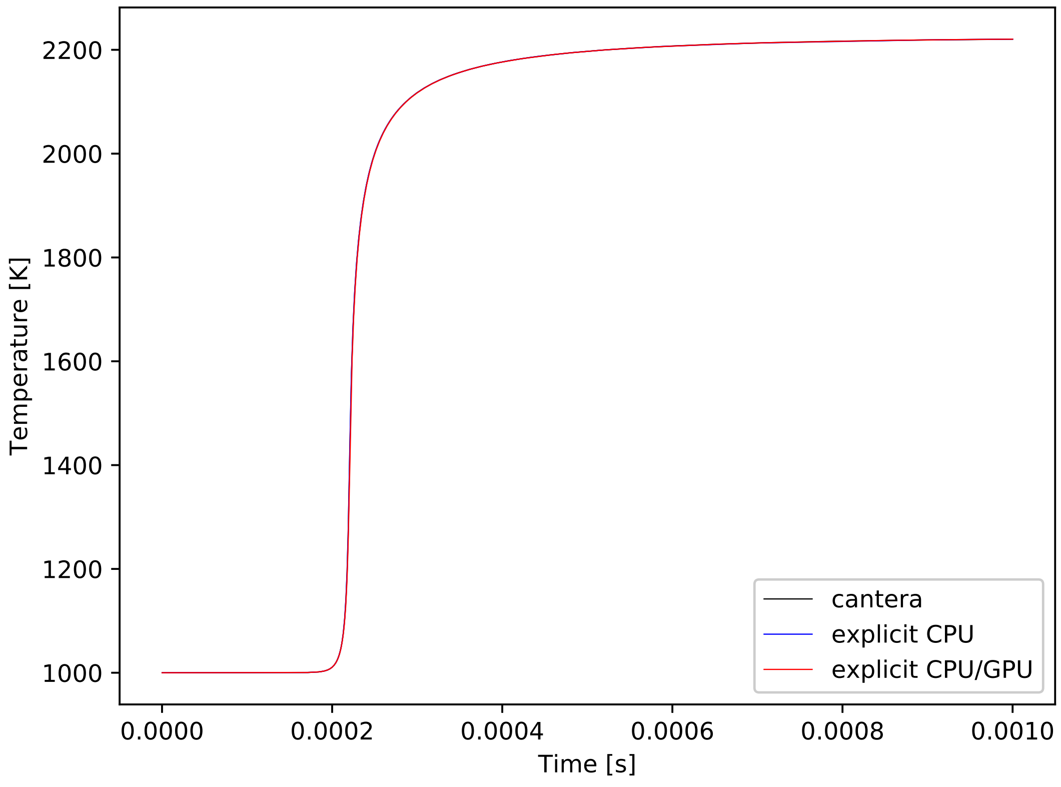
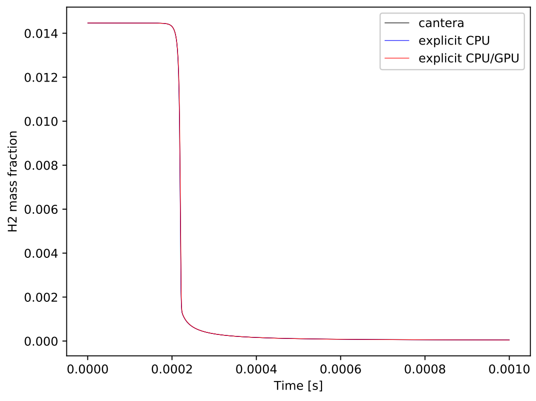
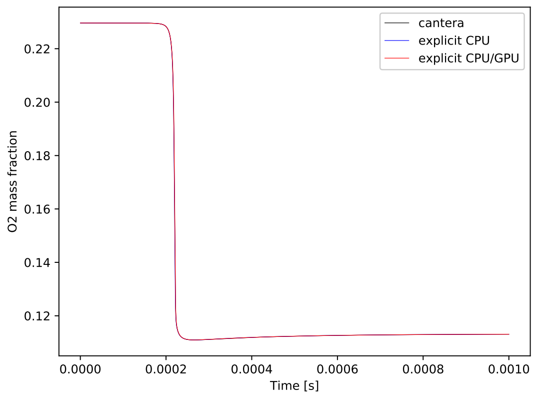


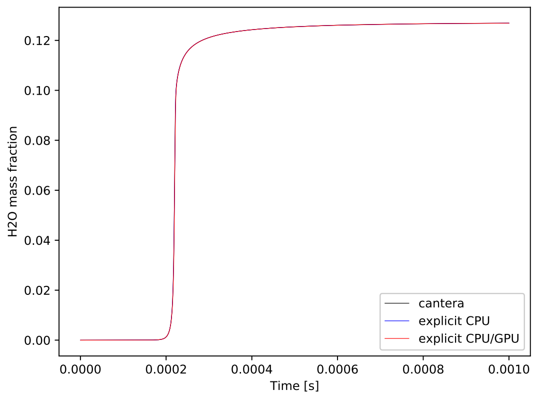
Figure 5.1: Time evolution of temperature, single-cell hydrogen combustion.



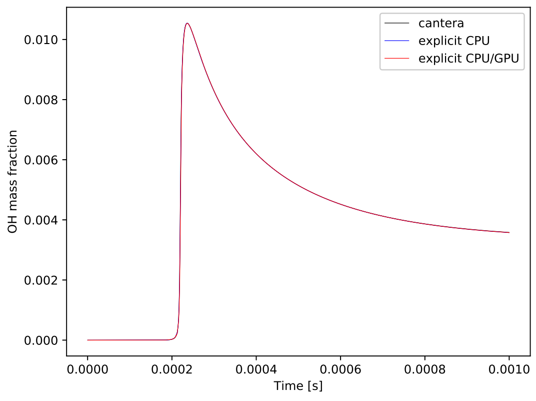
(a) H_2



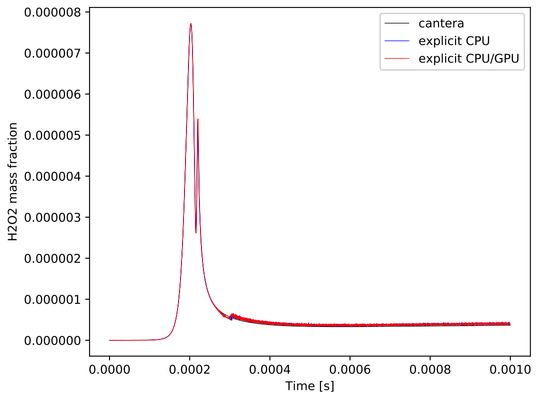
(b) O_2



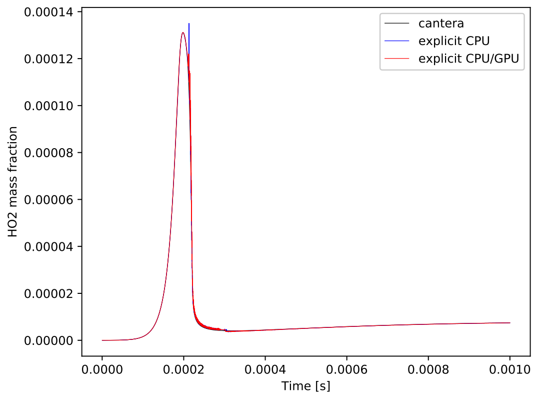
(c) H_2O



(d) OH



(e) H_2O_2



(f) HO_2

Figure 5.2: Time evolution of main species mass fractions, single-cell hydrogen combustion.

Figures 5.3 and 5.4 focus their attention to the slope of the curves, from $t = 2 \times 10^{-4}$ to $t = 2.5 \times 10^{-4}$. In particular, the curve is simply translated and one methods anticipates the other by few fraction of seconds (as the temperature, figure 5.3). The HO_2 mass fraction evolution behaves slightly different: again, it is noticeable the time translation between explicit CPU and explicit CPU/GPU method but there is also an oscillatory behaviour at the slope start where the two methods are different.

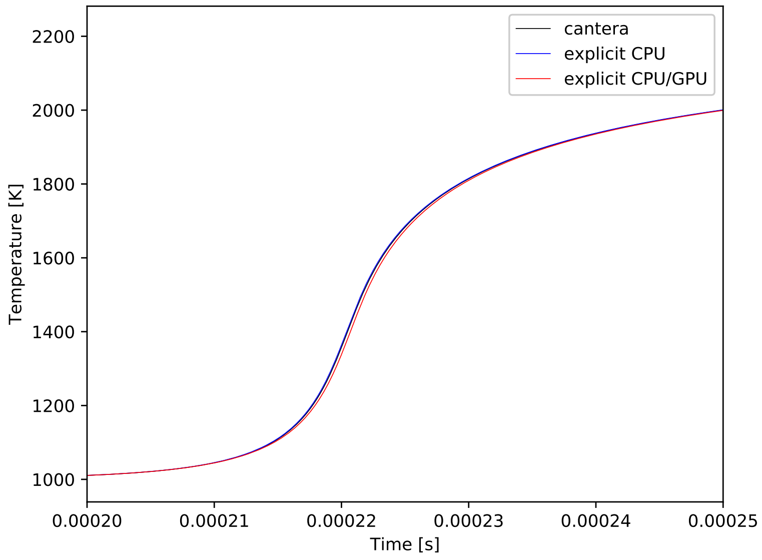


Figure 5.3: Time evolution of temperature, single-cell hydrogen combustion: detail.

This is substantially due to the parameters used to compute the steps of the chemical ODEs. In particular, CPU and GPU versions of RKCK45 execute the same tasks but have a different truncation error, depending on the particular hardware used. Thus, the same result can not be achieved with the same set of ODE parameters, that might be slightly different in this case. This difference is evident when the variable of the differential equation starts changing, while it is not present when the solution stays quite constant.

This phenomenon is even more visible when plotting the relative error amongst the solutions. In particular, all the data set were interpolated between two values in order to have the results on the same time. Results are shown in figures 5.5 and 5.6. For temperature and most of the species involved, the relative error

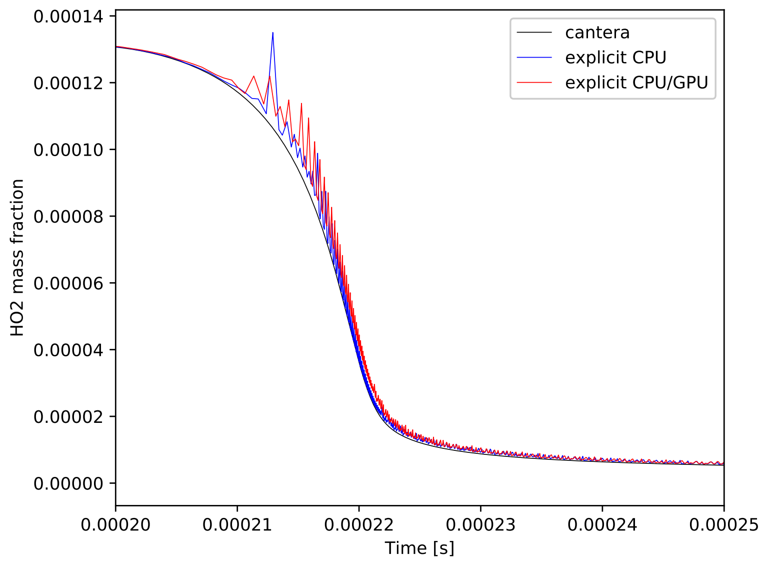
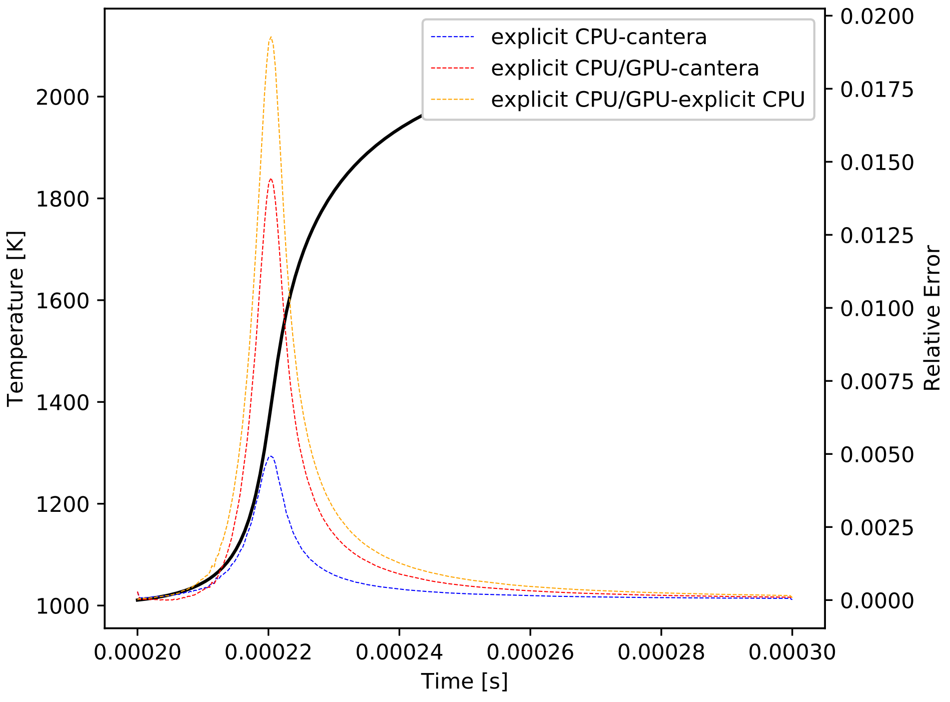


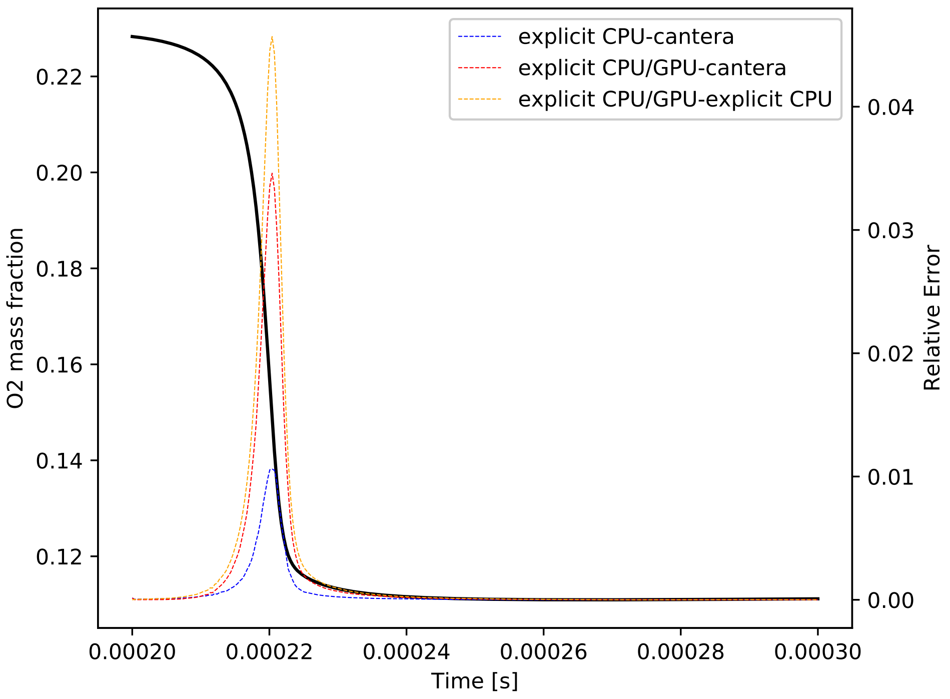
Figure 5.4: Time evolution of HO_2 mass fraction, single-cell hydrogen combustion: detail.

has its peak in the middle of the slope, as it can be seen for temperature and O_2 mass fraction. In particular, we can see that the max relative error of the explicit CPU/GPU with respect to the Cantera solution reaches low values in this case (up to 4% for oxygen, 2% for temperature). Furthermore, the max error is even lower referring to the explicit CPU method (less than 1%), while also the OpenFOAM `RKCK45` does not match perfectly the reference solution.

Much more interesting is the behaviour of HO_2 and H_2O_2, as highlighted in figure 5.6. Here the relative error has an oscillatory behaviour and reaches higher values (up to 20% for OH). Furthermore, also the CPU solution gives similar errors with respect to the reference solution, revealing that the inexact behaviour could be a consequence of the explicit ODE solution method adopted. One should note that the highest relative errors are obtained when the value of the solution is very small (less than $1e-5$ for OH and than $1e-6$ for H_2O_2 mass fraction). Thus, the weight of these errors on the solution is minimal and it can be caused by the different machine arithmetic of the GPU as well as the oscillations for small values of the solution. Anyway, the error does not seem to propagate, especially when the solution reaches higher values.

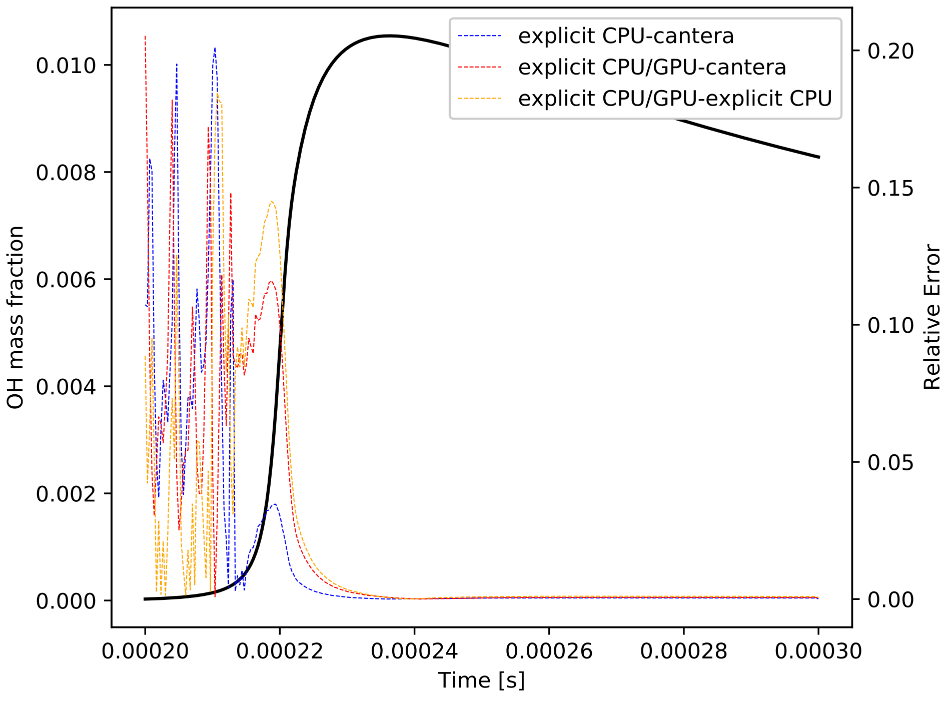


(a) Temperature

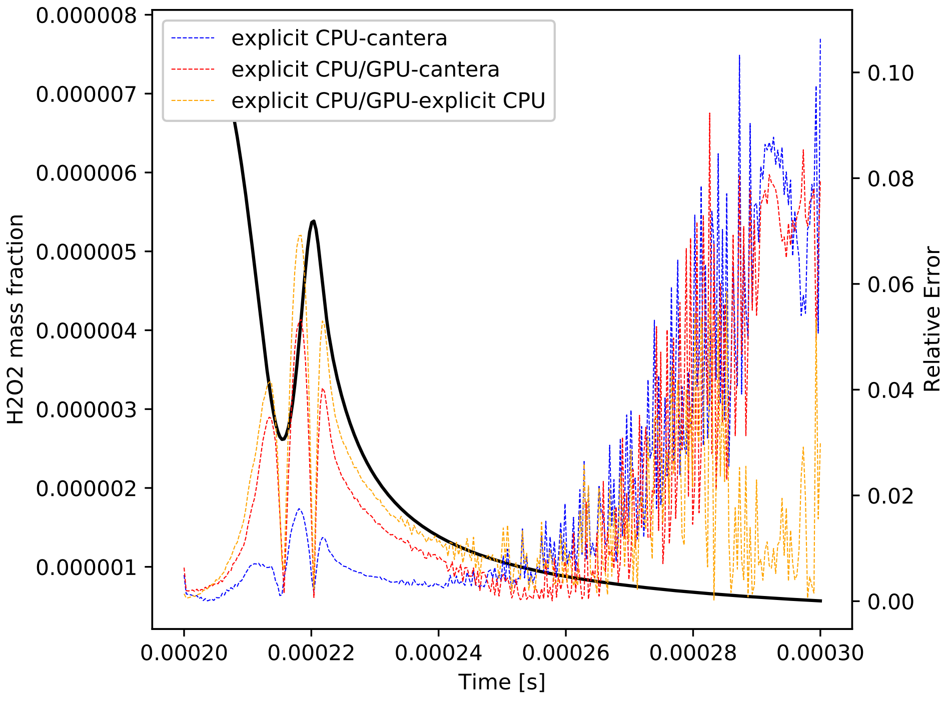


(b) O_2

Figure 5.5: Relative error of the time evolution of temperature and species mass fraction, single-cell hydrogen combustion.

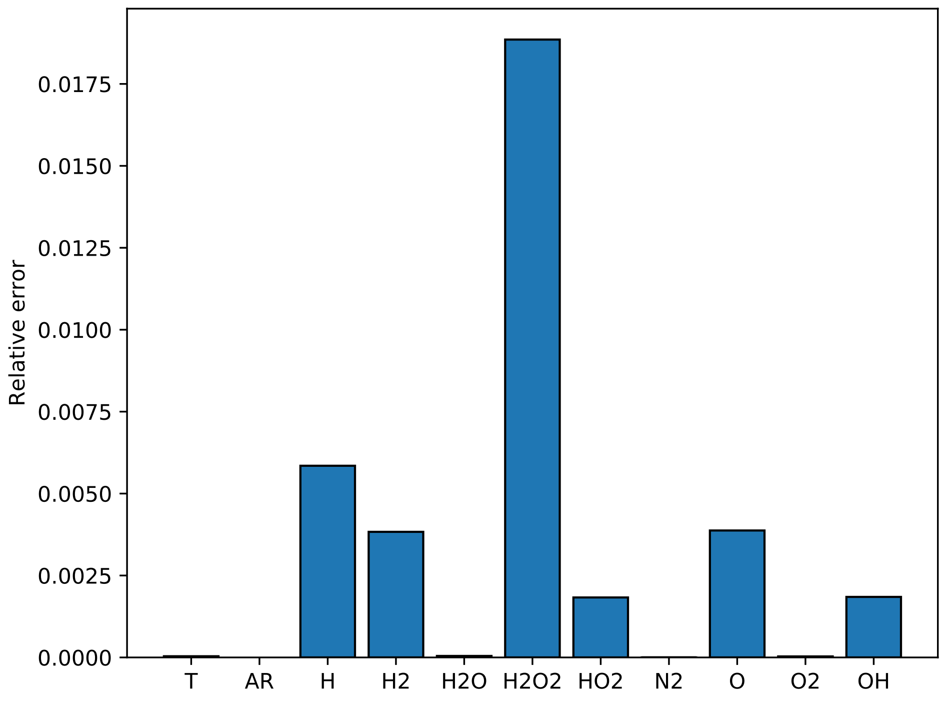


(a) OH

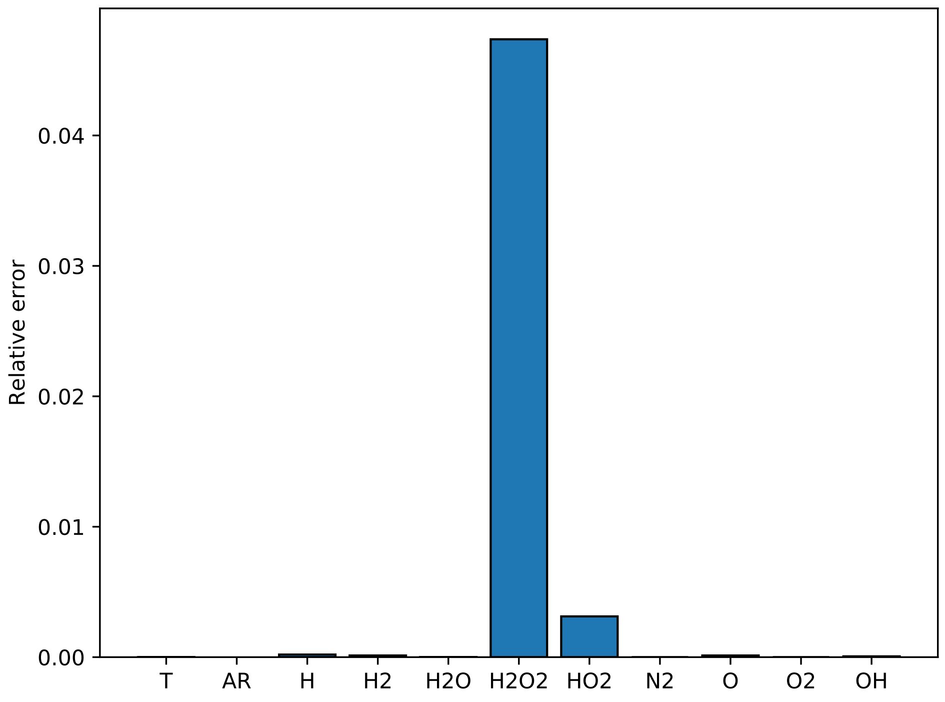


(b) H_2O_2

Figure 5.6: Relative error of the time evolution of species mass fraction, single-cell hydrogen combustion.



(a) Explicit CPU/GPU VS Cantera



(b) Explicit CPU/GPU VS explicit CPU

Figure 5.7: Relative error of the result for temperature and species mass fractions, single-cell hydrogen combustion.

Since it is important that the implementation gives the right final result, a comparison between explicit CPU/GPU and the other two solution final results (i.e. at end time) has been conducted. Figure 5.7 shows relative errors amongst the end values. The highest relative error is related to H_2O_2; it is noticeable that the error on the final value does not exceed 2% with respect to the Cantera result, while it is slightly higher compared to the CPU result. Furthermore, one can clearly see that the error does not propagate since its final value is smaller than the intermediate one.

One can conclude that the explicit CPU/GPU method works correctly in terms of accuracy of the results for low-stiffness chemical mechanism and single-cell cases.

5.1.2 Methane combustion mechanism

The combustion of an aliphatic hydrocarbon consists in several reactions where the original fuel is disintegrated in intermediate species, that can be fuel themselves. Final products usually are H_2O and CO_2, while other species (H_2, CO, OH, ...) are frequently present [18].

The following test case is based on the reduced chemical kinetics system GRI-Mech, an optimized mechanism created to model natural gas combustion [26]. This model was designed by the Gas Research Institute (GRI) and it was updated several times. The current version is the 3.0 release. Methane is largely present on Earth, making it one of the cheapest fuels, thus very attractive. It has two main problems: first, it is challenging to manage it due to its gaseous state at ambient temperature and pressure; second, it produces several pollutants, such as CO_2 or NO_2, that are responsible of greenhouse effect. Thus, a key research point in last years is the reduction of the emission of carbon dioxide, aiming to delay climate changes effects as much as possible [27].

Unlike the hydrogen combustion mechanism, GRI-Mech is much more stiff. Then, an implicit method should be much more suited for this mechanism ODEs solution. It contains 5 elements, 53 species and 325 reactions, therefore its complexity is higher than the H_2 combustion mechanism. Initial conditions and chemistry properties are presented in table 5.3 and 5.4.

The same analysis applied to the hydrogen combustion simulation has been

T [k]	p [atm]	X_{CH_4}	X_{O_2}	X_{N_2}
1000	13.5	0.50	1.00	3.76

Table 5.3: Initial data, single-cell methane combustion.

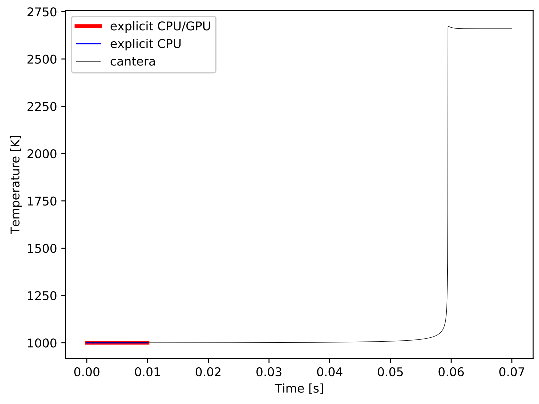
Δt [s]	t_{end} [s]	$absTol$	$relTol$
1e-5	7e-1	1e-16	1e-6

Table 5.4: Simulation setup, single-cell methane combustion.

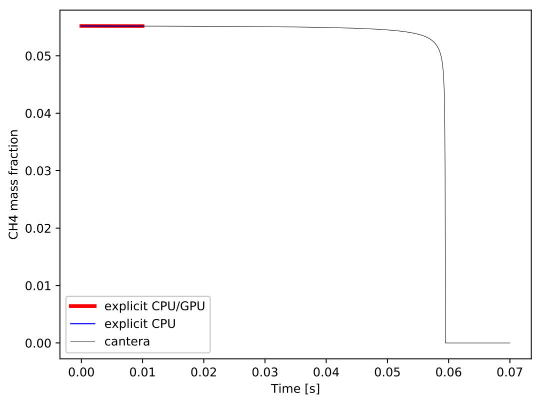
carried out for this case. Figure 5.8 shows the time variation of temperature and some of the most important species mass fractions.

In particular, explicit CPU/GPU and explicit CPU results are compared up to $t = 1 \times 10^{-2} s$ since the computational cost for this simulation is very high, as it will be highlighted in subsection 5.1.3. It is noticeable that the two curves overlap and they follow the reference Cantera solution. This observation is strengthened by figure 5.9 that shows the relative error between the two methods for temperature and some species mass fractions. As one can see, the error stays small (it does not exceed 0.2% for species considered). Much more interesting is the error trend, less regular than in the H_2/O_2 combustion. For instance, temperature and O_2 mass fraction do not show a well-defined trend but the error oscillates reaching small instantaneous peaks.

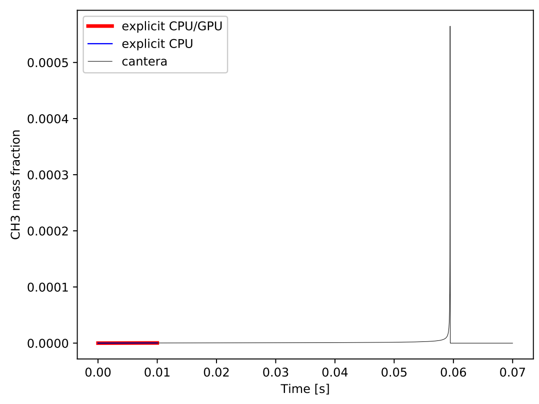
This behavior can be caused from the adopted interpolation as well as the fact that we are considering a time range when the reaction has not begun to develop. Finally, figures 5.10 show the relative error calculated on the final value obtained with the explicit CPU/GPU solution with respect to the Cantera reference solution. Figure 5.10a presents the relative error of temperature and principal species mass fractions, while figure 5.10b shows only those species with the highest error on the final value. It is worth noting that the error does not exceed 5% and the highest values are obtained for intermediate species like C_3H_7 and C_3H_8. Errors are much lower for main species: an error less than 1.5% is obtained for CH_4.



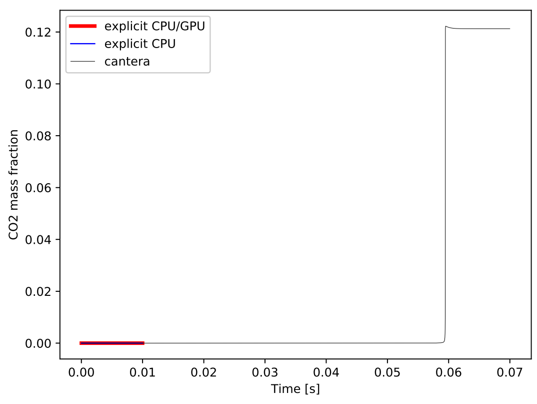
(a) Temperature



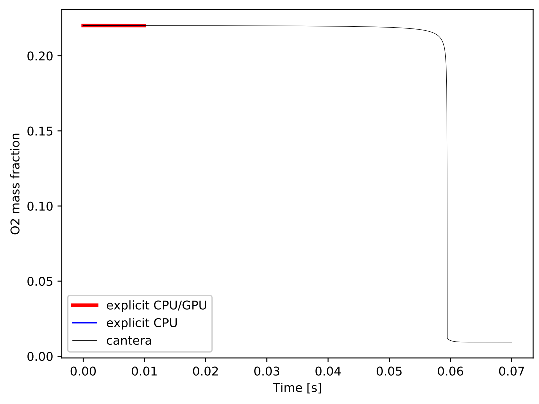
(b) CH_4



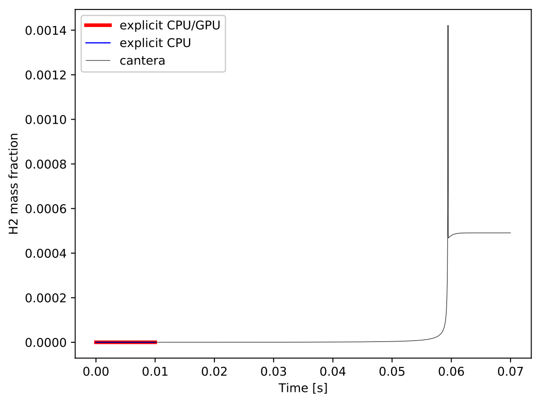
(c) CH



(d) CO_2

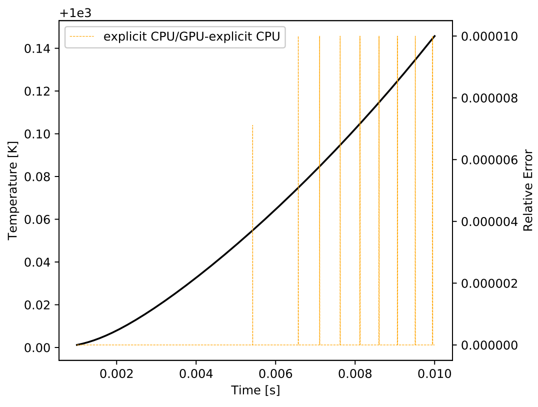


(e) O_2

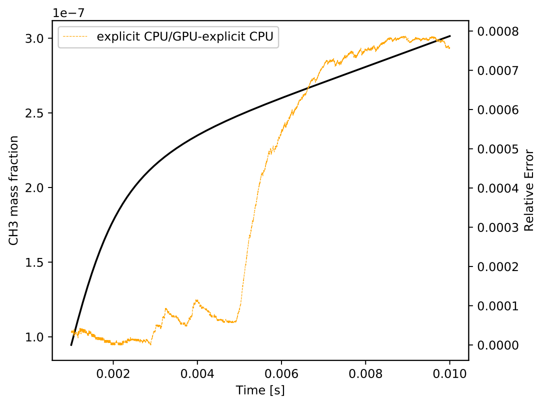


(f) H_2

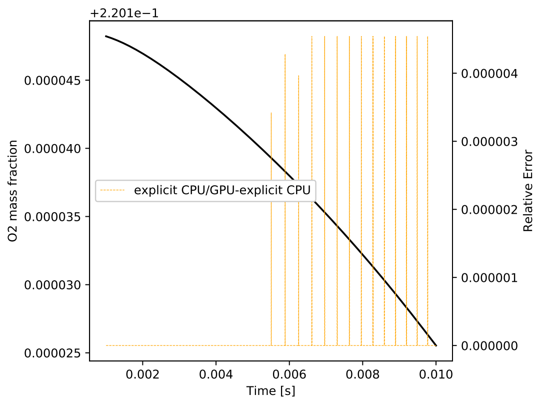
Figure 5.8: Time evolution of temperature and main methane combustion species mass fractions, single-cell methane combustion.



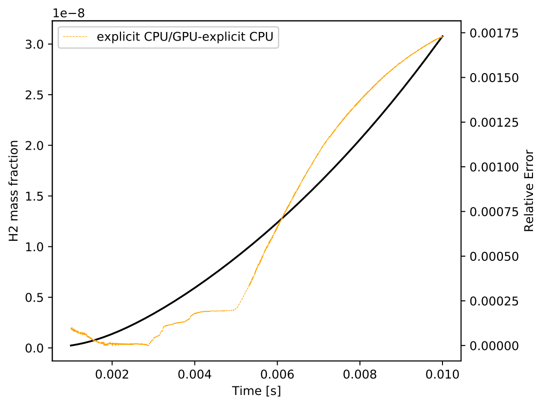
(a) Temperature



(b) CH_3

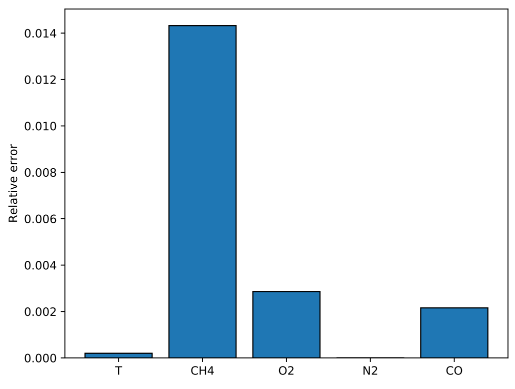


(c) O_2

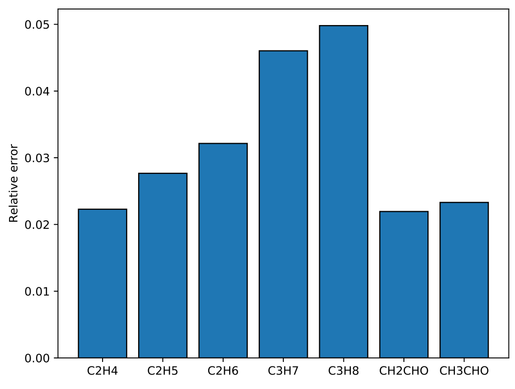


(d) H_2

Figure 5.9: Relative error of the time evolution of temperature and main methane combustion species mass fractions, single-cell methane combustion.



(a) Temperature and main species



(b) Highest error on final value

Figure 5.10: Relative error of the result for temperature and species mass fractions with respect to reference solution, single-cell methane combustion.

5.1.3 Time performance

In this section, computational time for single cell simulations is presented. Figure 5.11 shows time taken to complete H_2 combustion simulation (about 3829 chemical time steps for explicit CPU/GPU, 3831 for explicit CPU).

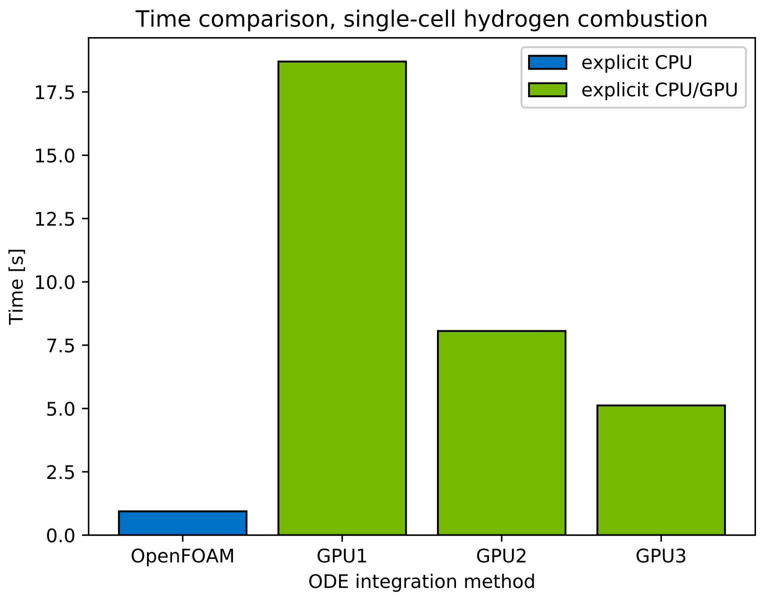


Figure 5.11: Time comparison, single-cell hydrogen combustion.

It is clearly evident that the simulation performs much faster with default OpenFOAM Runge-Kutta Cash-Karp method (less than 1 second), while execution with GPU takes much more time (i.e. five more times with faster NVIDIA TITAN V and even eighteen more times with NVIDIA GeForce 930MX). This confirms the assumption made before: GPU gets its best result when working with a system that can be parallelized, while in this case only a single cell is treated, thus a single block is created by CUDA. Furthermore, it is worth noting the differences between the three GPU hardware: a 3.6x speed-up is achievable with the NVIDIA TITAN V with respect to a low-end hardware.

Figure 5.12 shows time taken to complete methane combustion simulation. Here the computational effort is much higher than the other test case since the complexity of the mechanism. For this reason, we recorded the simulation time with *GPU3* only. It is noticeable that both the simulations take a large amount of time to complete, but the original explicit CPU version takes an order of magnitude lower than the CPU/GPU version to end.

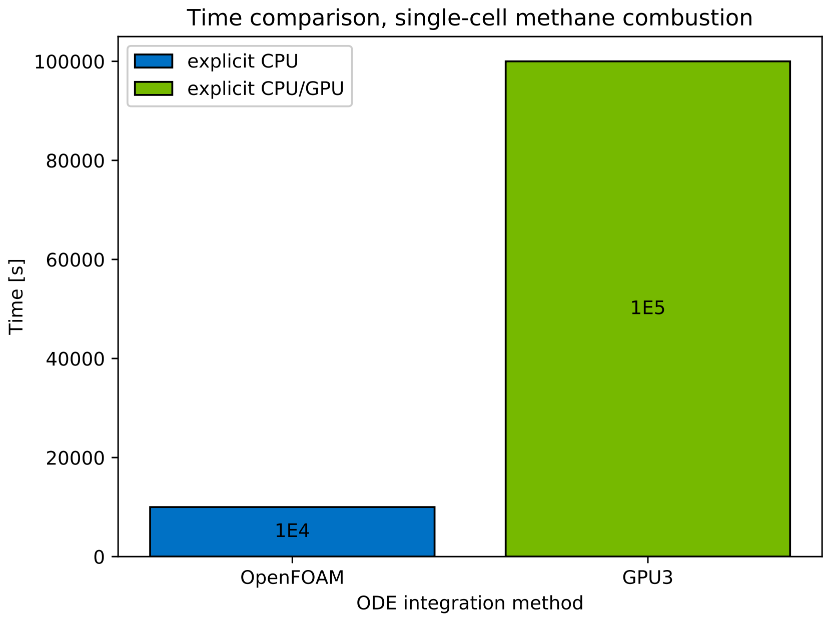


Figure 5.12: Time comparison, single-cell methane combustion.

Both the test cases highlight that, in single cell simulation, GPGPU computing seems that can not achieve an increment in performance. A large amount of time of the simulation is used for the operations of copying data from host to device and vice versa, for allocation on memory on GPU and for the initialisation of the libraries. In addition to this, the parallelization offered by the GPU is not exploited at all.

Method	H2		GRI	
	Avg [ms]	Std [ms]	Avg [ms]	Std [ms]
Implicit CPU	0.0451	0.0159	0.5621	0.0099
Explicit CPU	0.0334	0.0193	0.2171	0.0295
Explicit CPU/GPU	0.8110	0.4205	9.3243	1.2786

Table 5.5: Average and standard deviation of the computing time for all the ODEs treatment adopted.

Table 5.5 shows the time taken for each integration method to solve a single chemical time-step. We add a timing function to the code, before and after the implementation of the integration cycle, excluding all the operations of memory allocation, data copying and vector construction. Results exhibit clearly that explicit CPU/GPU method would perform much worse than its CPU counterparts (explicit and implicit). In particular, the hybrid CPU/GPU implementation is more than 24 and 43 times slower with respect to the explicit CPU method in both the test cases and more than 18 and 16 times slower than the implicit CPU integration method. The comparison with the implicit method is less indicative. In fact, an explicit method requires very small time steps, particularly in region of high numerical stiffness, to fulfill convergence and accuracy of the results. This leads to a low cost per time step but also to a greater number of steps, while in an implicit method the discretize time step can increase in size, leading to a smaller number of high-cost time steps. However, the comparison between the explicit methods clearly underlines why latency hiding is very important on GPUs. During CUDA kernel operations, a large amount of data are read on the global memory and copied in registers or cache memory. These global memory operations, as discussed in section 2.3, take more clock cycles to complete with respect to arithmetic operations or other memory operations. This large amount of time can be hidden when using lots of blocks, warps and threads in parallel: when a warp is copying data from device memory, other warps hide the latency by doing other operations. This procedure can not be exploited when launching few blocks and threads, as in these single-cell simulations. Thus, latency can not be hidden and the total computational time is higher than the original one.

Anyway, if it was possible to avoid the large amount of data transfer between host and device memory, the hybrid integration method would perform better than the CPU counterpart. A potential solution can be solving the numerical integration of the chemical kinetics ODEs in a single step and then giving the data back to OpenFOAM. In this way, the code would display the final result only, without showing intermediate values, and data would be stored in low-latency memory and latency would be hidden. Another possible implementation could be the size adjustment of the fluid dynamics time step based on whether or not the simulation is advancing in a low-stiffness region. In this case, the fluid dynamics time step can be increased about 10-15 times with respect to the chemical time step, while in a high-stiffness region the solver has to choose the original chemical time step. Hence, the number of iterations would decrease and so the computational effort. Due to the particular OpenFOAM implementation, this data transfer is necessary. However, a new `chemFoam`-based solver can be built in order to modify it and a speed-up might be reached in single-cell simulations, too.

5.2 Multi cell tutorials

The parallelization offered by GPGPU computing can not give the best results in terms of computational time when simulating a chemical mechanism in a single cell reactor, even though it has the same accuracy with respect to the OpenFOAM CPU version. A consistent improvement regarding the simulation time can be achieved when treating a multi-dimensional geometry: here the possibility to execute several cells at the same time can be a great advantage.

To better compare the effects of a GPU architecture on reactive flows simulations, the same geometry has been used for all the tutorials. Thereby, it is possible to analyze the effect of the CUDA parallelization only, removing other time consuming effects such as different complexities of the mesh. The geometry chosen was the simplest possible: a bi-dimensional square in xy-plane with rectangular mesh elements. Mesh could have different refinements on the two dimensions. Since OpenFOAM is based on a finite volume method, the bi-dimensional geometry is shaped as a regular hexaedra with a single cell in the z-direction.

The physical phenomenon examined is a laminar diffusion counter flow flame. A diffusion flame consists in a particular combustion class of phenomena where the fuel and the oxidizer (in general air) are not mixed together, but they are released from different location in the combustion chamber. The reactants have to be released into the reaction zone fast enough to let the combustion to proceed. This class is different from the *premixed flame*, where the species are combined in a mixture before reaching the flame. Diffusion flames have a recurrent structure, shown in figure 5.13.

It is noticeable the presence of two diffusion zones, where the fuel (on the left) and the oxidizer (on the right) are released; here temperature and mass fraction are constant due to the boundary conditions. Between them, the reaction zone

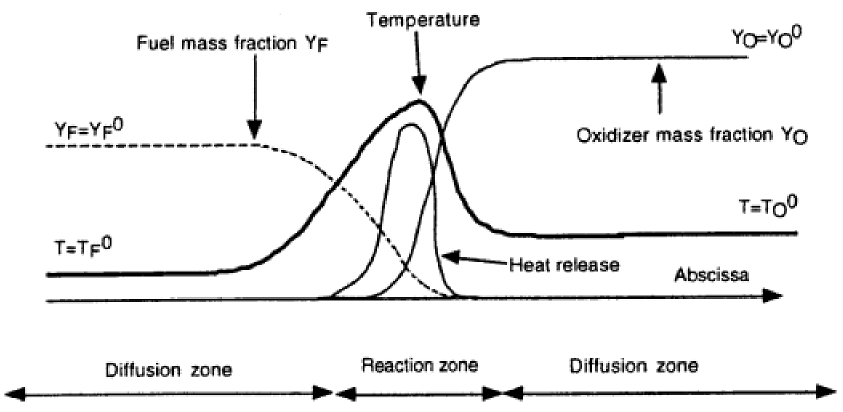


Figure 5.13: Diffusion flame structure [28].

is characterized by a peak on the temperature profile (and the heat generated). The maximum of these quantities outlines the *flame front*, defined as a discontinuity surface that separates what is burnt from the unburnt. Of course, the fuel decreases when moving through the oxidizer and vice versa, showing the consumption of the reactants for the combustion; on the other hand, products of the reaction (usually CO_2 or H_2O) show their maximum concentration in the core of the combustion.

Figure 5.14 shows the boundary condition applied on the geometry. Three patches are applied:

- `fuel`, on the left wall, where the fuel is released;
- `air`, on the right wall, where the oxidizer is released;
- `outlet`, on the top and bottom wall.

In addition to this, an `empty` condition is imposed on the front and the back wall since equations along the third dimension are not solved. Species mass fractions are fixed on the lateral walls, based on the structure of the diffusion flame previously explained. A fixed value velocity boundary condition is enforced, too: the two gases have a speed of $0.1m/s$ in opposite direction, due to the counter flow configuration. Applied boundary conditions are described in table 5.6.

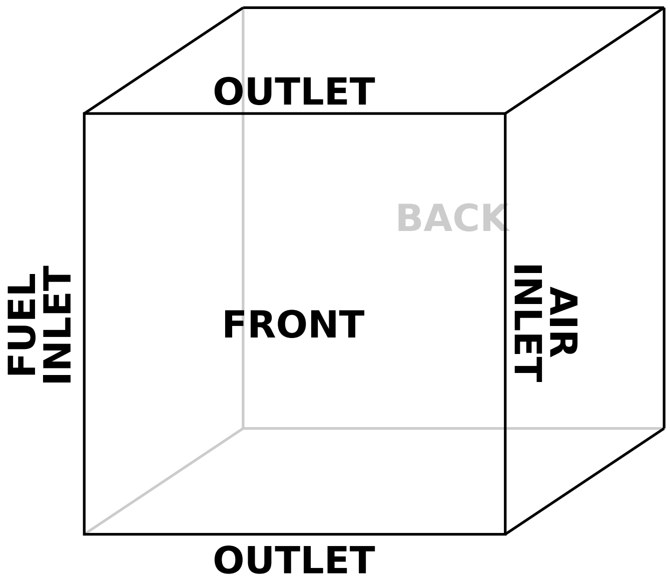


Figure 5.14: Counter-flow flame boundary patches.

	fuel	air	inlet/outlet
T	`fixedValue`	`fixedValue`	`inletOutlet`
p	`zeroGradient`	`zeroGradient`	`zeroGradient`
Y_k	`fixedValue`	`fixedValue`	`inletOutlet`
U	`fixedValue`	`fixedValue`	`pressureInletOutletVelocity`

Table 5.6: Counter-flow flame boundary conditions types.

5.2.1 Simple methane combustion mechanism

The simple methane combustion mechanism was created by Bui-Pham [29] to simulate this particular counter-flow configuration, so the mechanism has to be considered for research uses only. The mechanism is elementary and consists on 5 species, 3 elements and a single reaction:

$$CH_4 + 2O_2 \rightarrow CO_2 + 2H_2O \tag{5.2}$$

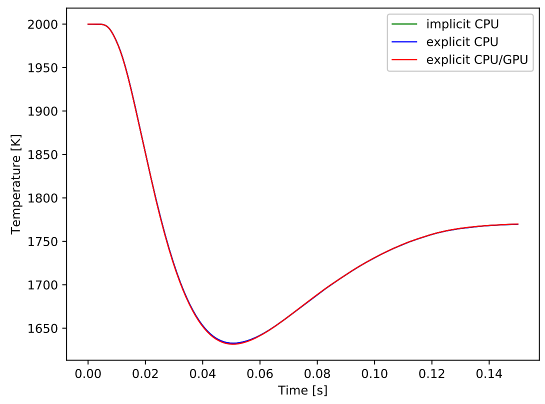
Due to the low complexity of the reaction, the purpose of this simulation was to validate the multi-cell solver in terms of results more than showing a performance improvement. The best effects of the GPU architecture occur when complex reaction are treated; thus, no sensible speed-up is expected from a single reaction mechanism. Boundary conditions and initial values of the test case are presented in table 5.7.

	fuel	air	inlet/outlet	internal field
T [K]	293	293	293	2000
p [atm]	`zeroGradient`			1e-5
Y_{CH_4}	1	0	0	1
Y_{O_2}	0	0.23	0	0
Y_{N_2}	0	0.77	1	1
U_x [m/s]	0.1	0	0	0

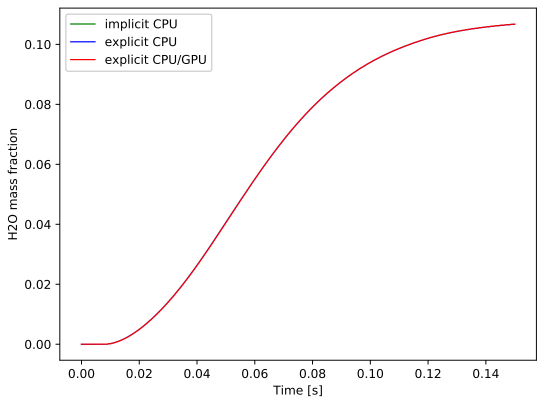
Table 5.7: Methane combustion boundary conditions.

Figure 5.15 shows a comparison amongst the integration methods on different locations. Two probes are considered: the former near the fuel inlet, the latter next to the oxidizer inlet. Temperature is analyzed on both the location (5.15a, 5.15b), while CH_4 and O_2 are compared respectively on their own inlet(5.15c, 5.15d). You can notice that the three results taken from explicit CPU, explicit CPU/GPU and implicit CPU (taken as reference) are overlapping.

Results overlap when checking other mesh location too, as it is shown in figure 5.16. Here a probe in the middle of the geometry is chosen while temperature (5.16a) and H_2O (5.16b) are compared.



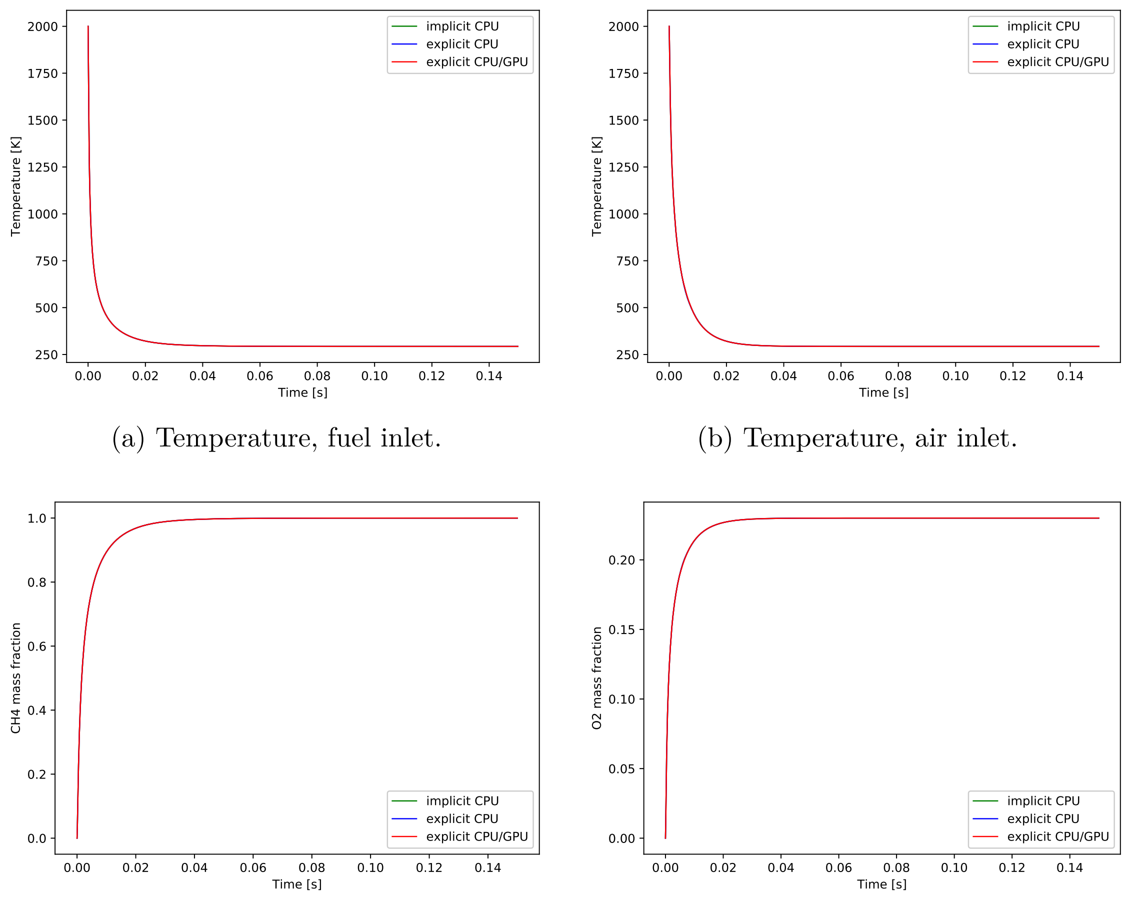
(a) Temperature, center location.



(b) H_2O, center location.

Figure 5.16: Time evolution of temperature and H_2O mass fraction on central probe, multi-cell simple methane combustion.

Another interesting comparison is about the spatial variation of the different



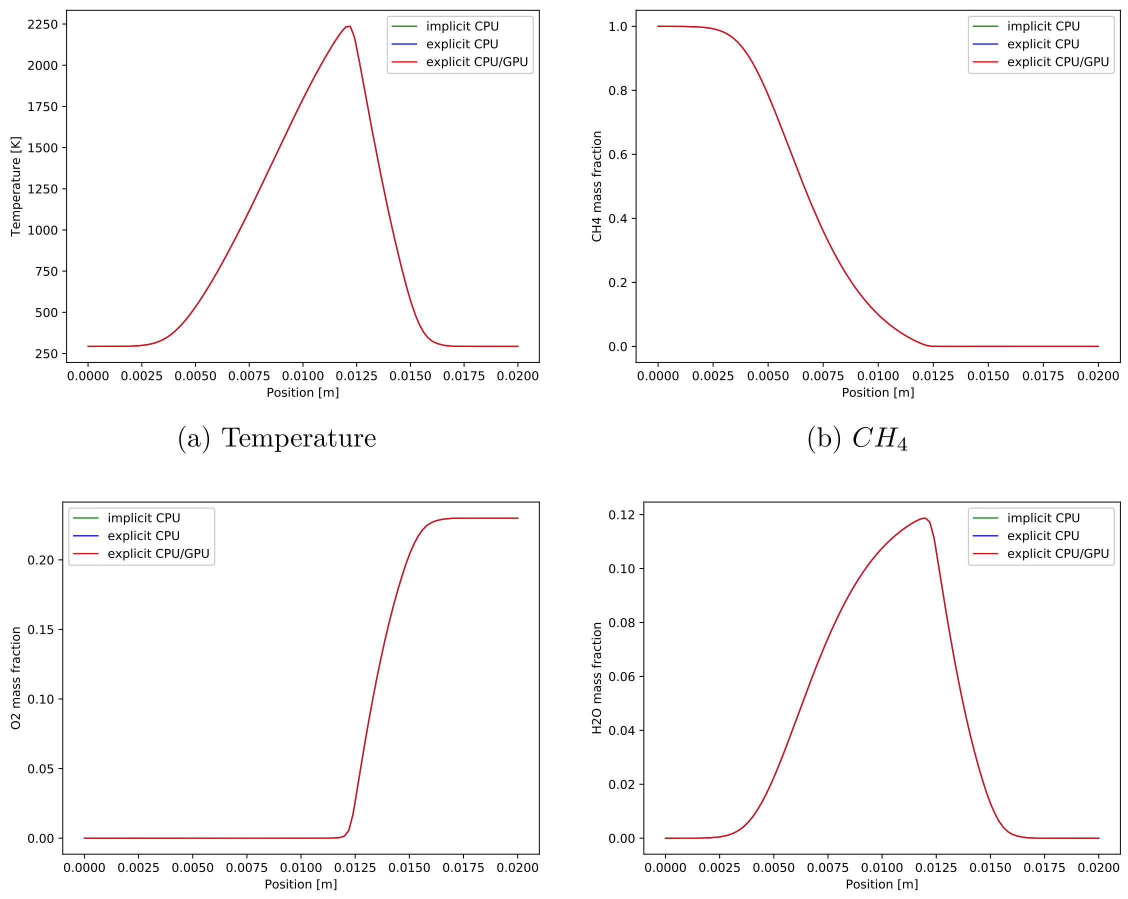
(a) Temperature, fuel inlet. (b) Temperature, air inlet.

(c) CH_4, fuel inlet. (d) O_2, air inlet.

Figure 5.15: Time evolution of temperature, CH_4 and O_2 mass fraction on fuel and air inlet, multi-cell simple methane combustion.

quantities. This type of analysis was not possible in single-cell tutorials, while it shows interesting features in multi-cell simulations. Figure 5.17 represents comparisons of temperature and species variation along x-direction for different integration methods. Data have been taken at time $t = 0.15s$. As happened for the temporal variation, these plots highlight the perfect overlapping of the three methods.

Finally, the front flame obtained with the hybrid code is represented in figure 5.18. The front flame surface is positioned at $x = 0.012m$, confirming the OpenFOAM results with explicit and implicit integration methods.



(a) Temperature

(b) CH_4

(c) O_2

(d) H_2O

Figure 5.17: Spatial variation of temperature and mass fractions at time $t = 0.15s$, multi-cell simple methane combustion.

5.2.2 Syngas combustion mechanism

The term *syngas* derives from the union of two words, **Syn**tethic **gas**. It is a mixture of molecular hydrogen H_2 and carbon monoxide CO, with the presence of methane CH_4 and carbon dioxide CO_2. It is used for producing synthetic natural gas (SNG), but also as fuel in internal engines; the process where the syngas is created is called gassification. As previously mentioned in 5.1.1, hydrogen combustion is a very important phenomenon because of its efficiency as propellant and the environmental impact that it has with respect to classical

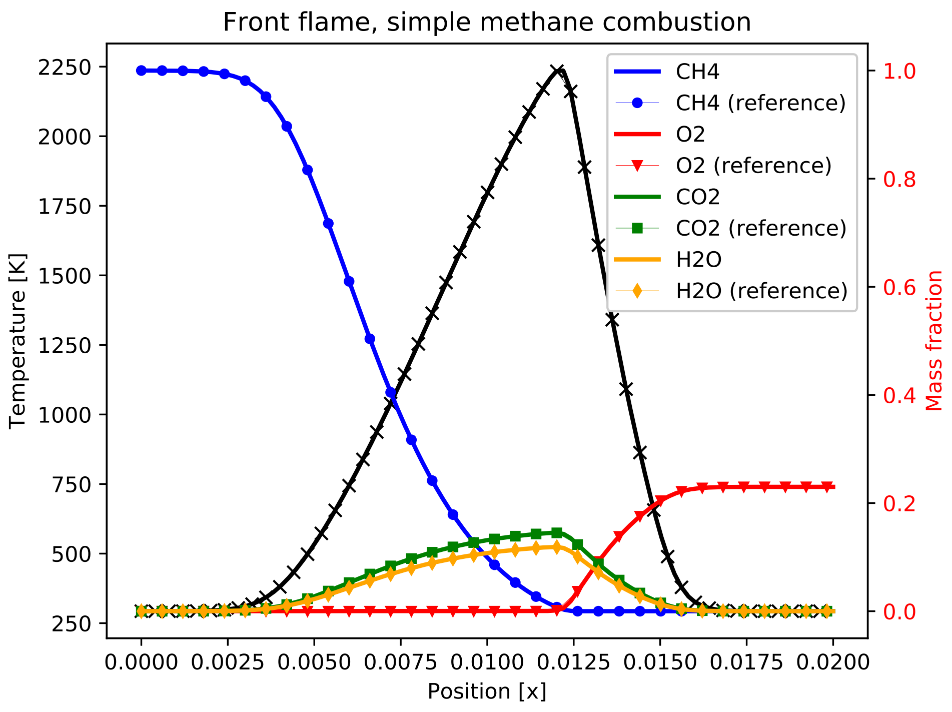


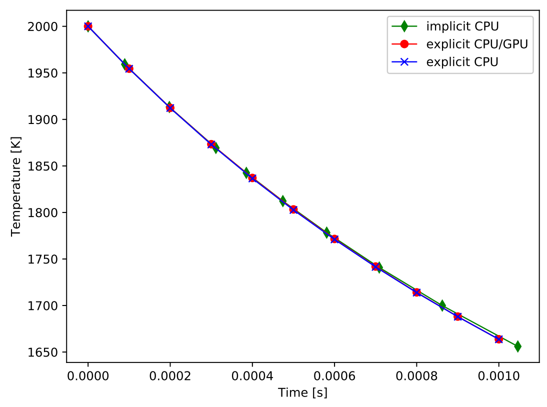
Figure 5.18: Front flame, multi-cell simple methane combustion.

fossil fuels [30].

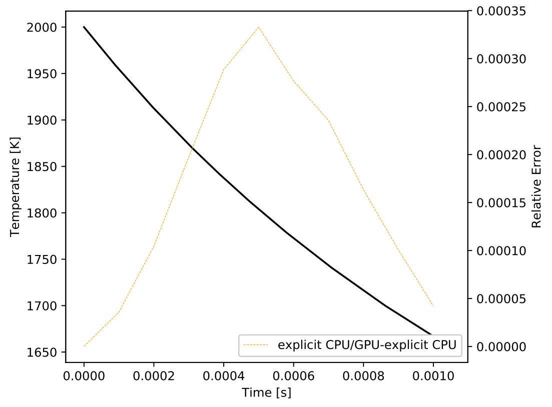
The reduced chemical mechanism used in this test case was developed by the Chemical Reaction Engineering and Chemical Kinetic (CRECK) lab from Politecnico di Milano [31–33]. It consists on 6 chemical elements, 21 species and 62 reactions and it is a medium-stiffness mechanism. The boundary conditions and the initial values of the simulation are showed in table 5.8.

	fuel	air	inlet/outlet	internal field
T [K]	1500	1500	1500	2000
p [atm]	`zeroGradient`			1e-5
Y_{H_2}	1	0	0	0
Y_{CO_2}	1	0	0	0
Y_{O_2}	0	0.53	0	0
Y_{N_2}	0	0.47	1	1
U_x [m/s]	0.1	0	0	0

Table 5.8: Syngas combustion boundary conditions.

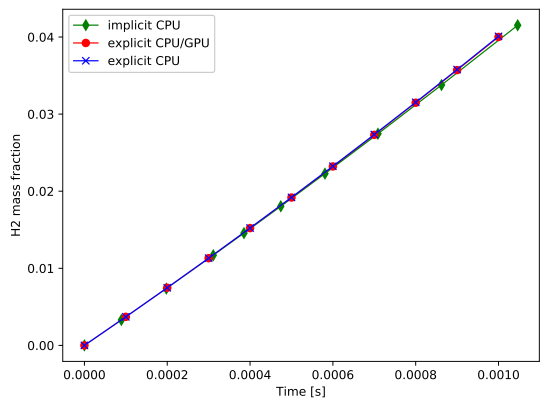


(a) Temperature

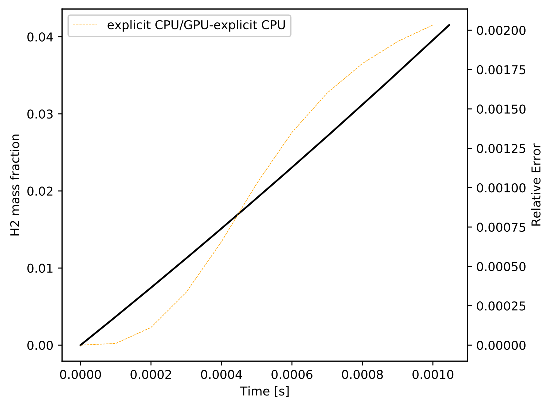


(b) Temperature, relative error

Figure 5.19: Time evolution of temperature on fuel inlet, multi-cell syngas combustion.



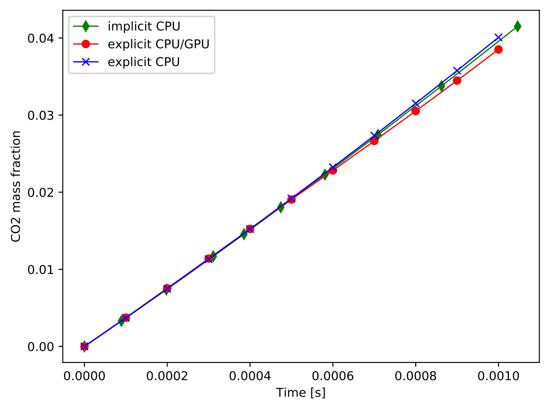
(a) H_2



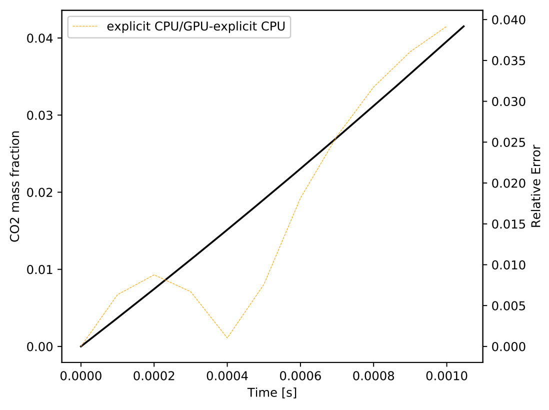
(b) H_2, relative error

Figure 5.20: Time evolution of H_2 mass fraction on fuel inlet, multi-cell syngas combustion.

Since the GPU code has already been validated, it is not essential to develop the combustion until the end. Moreover, the complexity of this test would have caused a sensible increase in computational cost to reach the final time. Thus, the simulation was stopped at $t = 1 \times 10^{-3} s$ and results have been compared.

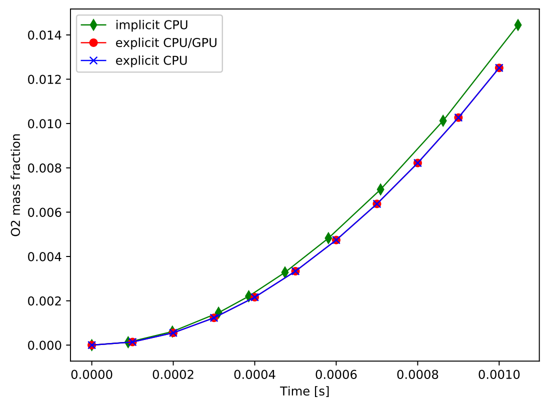


(a) CO_2

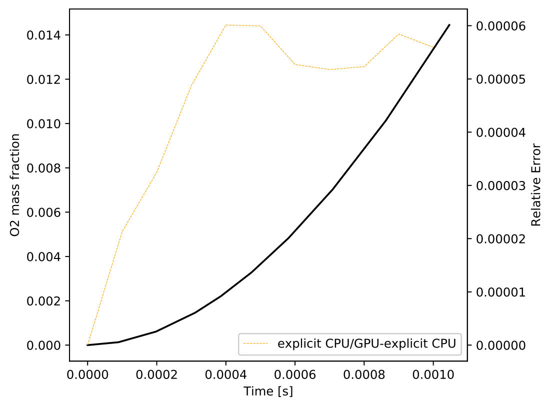


(b) CO_2, relative error

Figure 5.21: Time evolution of CO_2 mass fraction on fuel inlet, multi-cell syngas combustion.



(a) Temperature



(b) Temperature, relative error

Figure 5.22: Time evolution of O_2 mass fraction on air inlet, multi-cell syngas combustion.

Figures 5.19, **??** and 5.22 show the time evolution of temperature and most important species mass fractions and the respective relative errors between explicit CPU and explicit CPU/GPU methods. Temperature (figure 5.19), H_2 and CO_2 mass fractions (figure **??**) were recorded at the fuel inlet, while O_2 mass fraction (figure 5.22) was taken at air inlet. One can note that the behavior is respected with explicit CPU/GPU method even though the results are not perfectly overlapping. This can be caused by the tolerances used for the ODEs solution that do not allow the implicit and the explicit methods to give the same result. In fact, the two explicit methods give overlapping solutions and this is confirmed by the relative error computed on these two integration methods. This does not exceed the 4% on CO_2 (figure 5.21b) and it keeps low with the other quantities, validating the accuracy of the results.

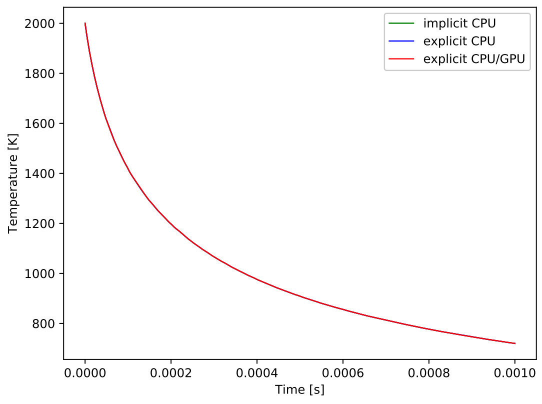
5.2.3 Complex GRI-Mech 3.0 mechanism

The following case is based on the GRI-Mech 3.0 combustion, already presented in section 5.1.2 [26]. For the same reasons presented in subsection 5.2.2, the simulation was stopped at $t = 1 \times 10^{-3}s$ and results have been compared. Even though the simple methane combustion (section 5.2.1) has been totally simulated until the right end time, we did not find it necessary to do that with this test case. In fact, the cost of this simulation is much higher than the previous two multi-cell cases and it is possible to demonstrate the accuracy of the result with the available data only.

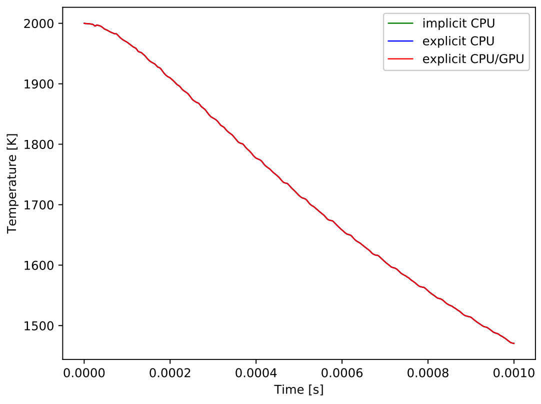
Figure 5.23 shows a comparison among the integration methods on different positions: fuel inlet, air inlet and a central position. Again, it is clearly noticeable that the results are perfectly overlapping.

Figure 5.24 shows the relative error computed on fuel and air inlet, for CH_4 and O_2 mass fraction respectively. It is clearly evident that the error is higher when the corresponding gas has not reached the probe location yet, thus for lower time, then decreases to zero. The maximum error is less than 0.5% in both cases, probably meaning that at smaller quantities values the GPU can not have the same precision as the CPU.

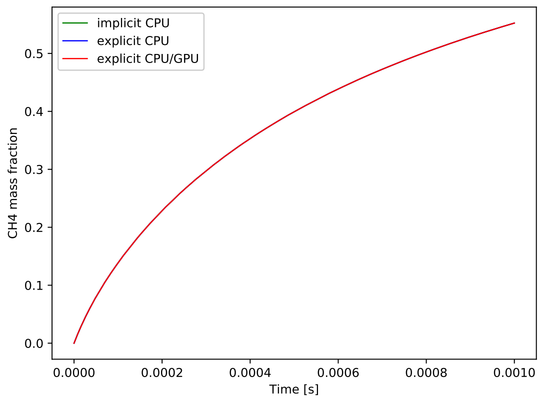
A comparison was made also for the different methods along the x-direction, at end time $t = 1 \times 10^{-3}s$. Results are shown in figure 5.25, highlighting good overlapping. The relative error has been calculated (figure 5.25b and 5.25d): again, biggest errors manifest when the quantities are smaller, confirming the previous assumptions.



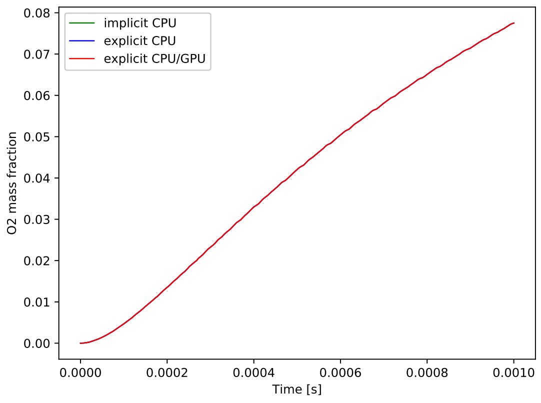
(a) Temperature, fuel inlet.



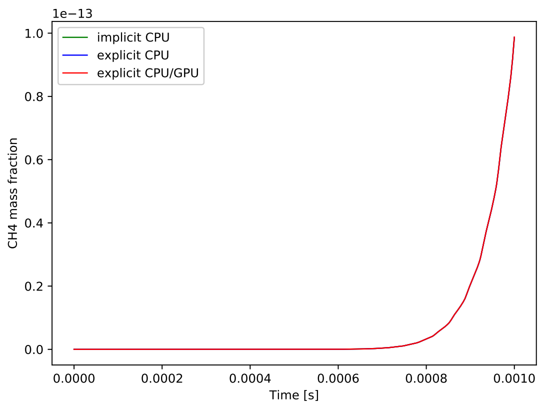
(b) Temperature, air inlet.



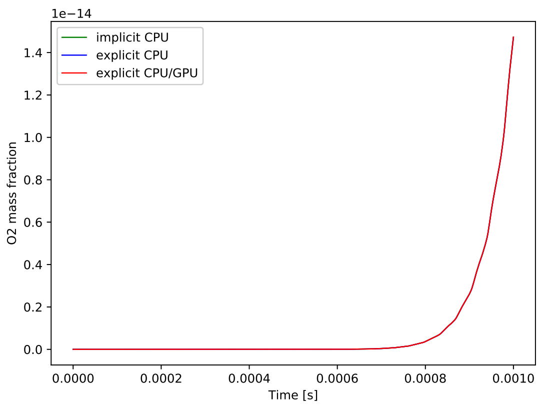
(c) CH_4, fuel inlet.



(d) O_2, air inlet.

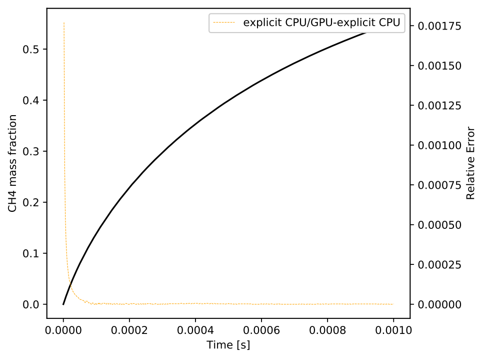


(e) CH_4, central inlet.

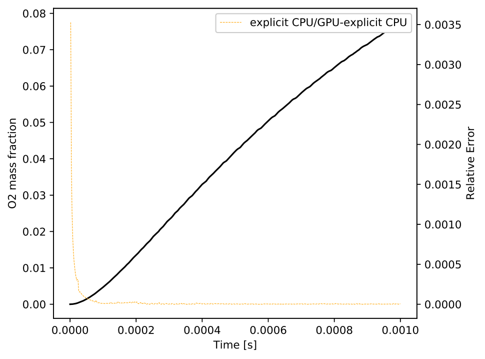


(f) O_2, central inlet.

Figure 5.23: Time evolution of temperature, CH_4 and O_2 mass fraction on different positions, multi-cell methane combustion.

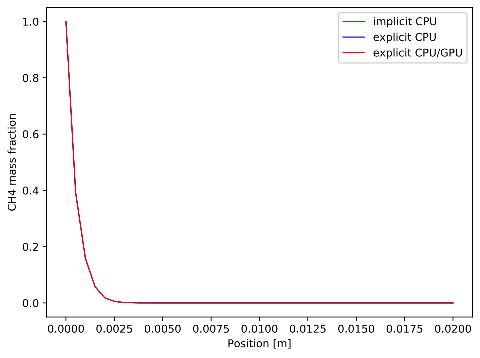


(a) CH_4, fuel inlet.

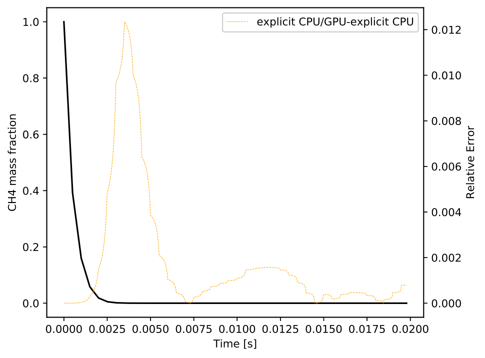


(b) O_2, air inlet.

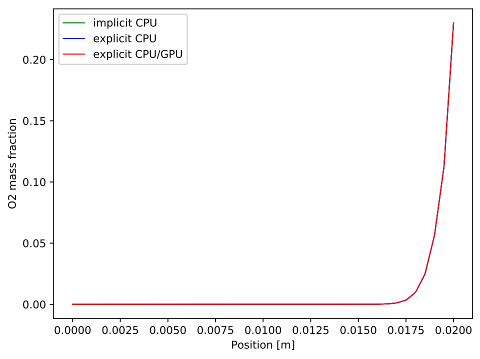
Figure 5.24: Relative error on time evolution of CH_4 and O_2 mass fractions on fuel and air inlet, multi-cell methane combustion.



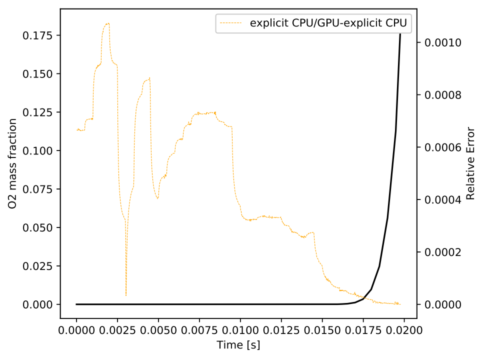
(a) CH_4



(b) CH_4, relative error.



(c) O_2

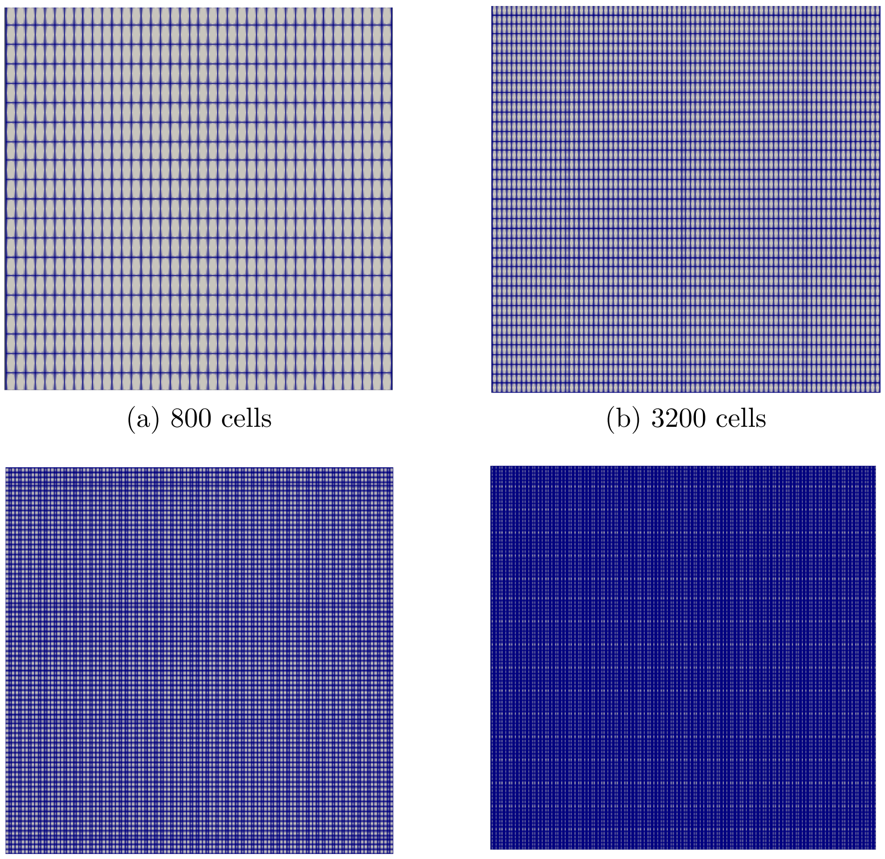


(d) O_2, relative error.

Figure 5.25: Spatial variation of CH_4 and O_2 mass fractions at time $t = 1\times10^{-3}s$, multi-cell methane combustion.

5.2.4 Time performance

Computational time analysis for multi-cell simulations is now presented. Due to the GPU architecture, a great speed-up can be achieved when treating multiple cells with respect to the single-cell cases. Tests were made on the NVIDIA TITAN V (*GPU3*) to have the maximum improvement possible. All the simulations were carried out until the same end time $t = 1 \times 10^{-5}$. Different number of cells have been tested, as shown in figure 5.26.


(a) 800 cells

(b) 3200 cells

(c) 9600 cells

(d) 20000 cells

Figure 5.26: Computational mesh for different number of cells.

Figure 5.27 shows computational time of the explicit CPU/GPU integration method compared with the explicit RKCK45 implemented in OpenFOAM of the simple methane combustion (section 5.2.1). This case was tested up to 20000 cells. It is clearly evident that time decreases when using the GPU even though there is not a great improvement.

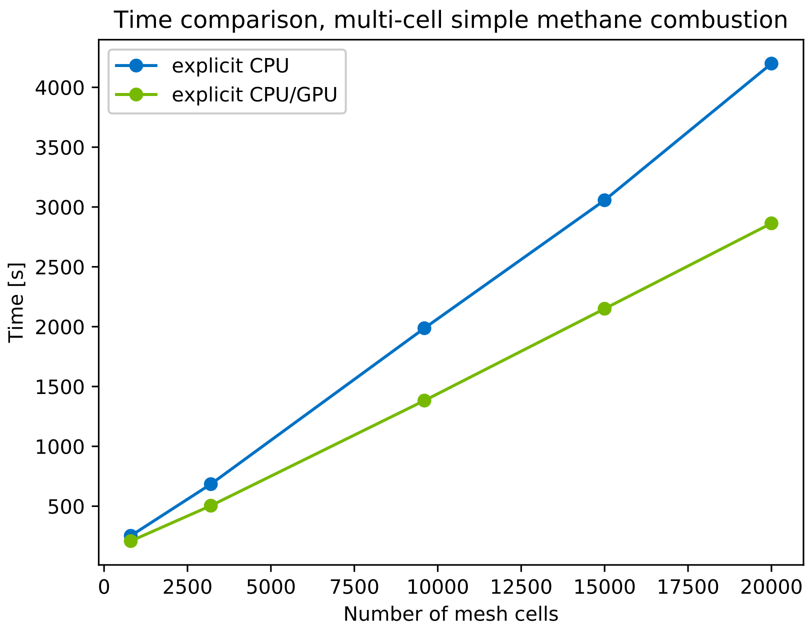


Figure 5.27: Time comparison, multi-cell simple methane combustion.

This is confirmed by figure 5.28 that shows the speed-up obtained when varying the number of cells. Some considerations can be made. A maximum 1.47x speed-up can be reached. Thus, performance improvements can be achieved even with a very simple reaction mechanism. This mostly confirms the goodness of the optimization work, since no improvements were obtained in this test case with previous versions of the code. Third, there is not a great improvement on the speed-up when changing the number of cells (from 1.21x to 1.47x) due to the low complexity of the chemical mechanism. Finally, an upper limit is reached when increasing the number of cells; above that, the speed-up seems to become uniform.

Figures 5.29 and 5.30 show the time comparison between the explicit CPU and the explicit CPU/GPU methods and the speed-up at various number of cells for the syngas combustion. Again, some interesting considerations can be made. First of all, the maximum speed-up is 3.55x reached at 30000 cells, more than twice the value obtained with the simple methane combustion. This confirms that better results can be achieved when the involved chemistry mechanism

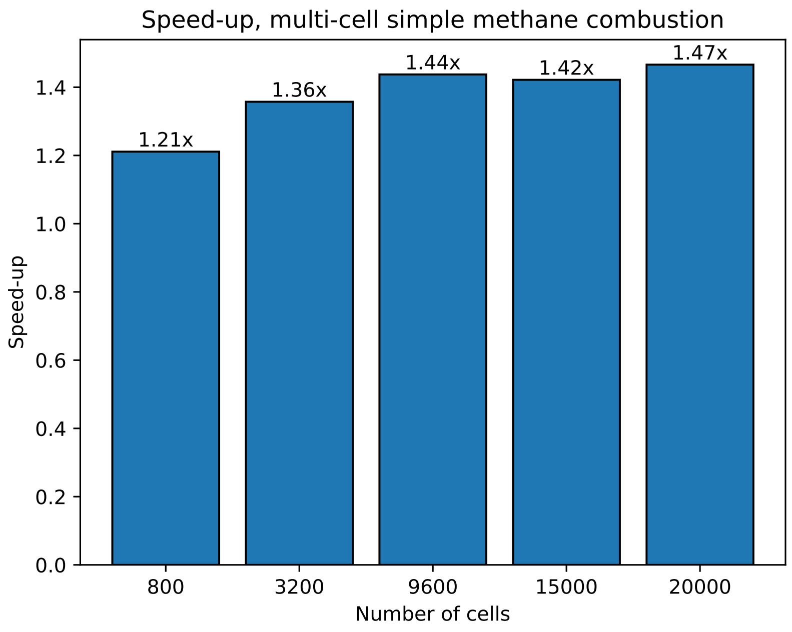


Figure 5.28: Speed-up, multi-cell simple methane combustion.

increases in complexity, while for simpler ones the CPU and the GPU take about the same time to perform. Second, when treating complex chemical chains the number of cells affects the speed-up much more than how it happens with simpler ones. One can clearly see a wider range of speed-ups, from 1.12x to 3.55x, with respect to the simple methane mechanism.

Figures 5.31 and 5.32 show the same syngas combustion mechanism, but with different ODEs integration tolerances: in particular, in this second case the tolerances are coarser than the first case. It is worth noting that the maximum speed-up is slightly lower than the previous case and, in general, the performances seem to be worst. Furthermore, speed-ups are much lower when treating a low number of cells, at the point that explicit CPU is faster than explicit CPU/GPU at 800 cells. Thus, we should say that at lower number of cells the simulation is tolerance-dependent, i.e. a change in the tolerance can improve or not the simulation time more than a different method can do. This is not happening at higher number of cells, where tolerances change the results but up to a less extent.

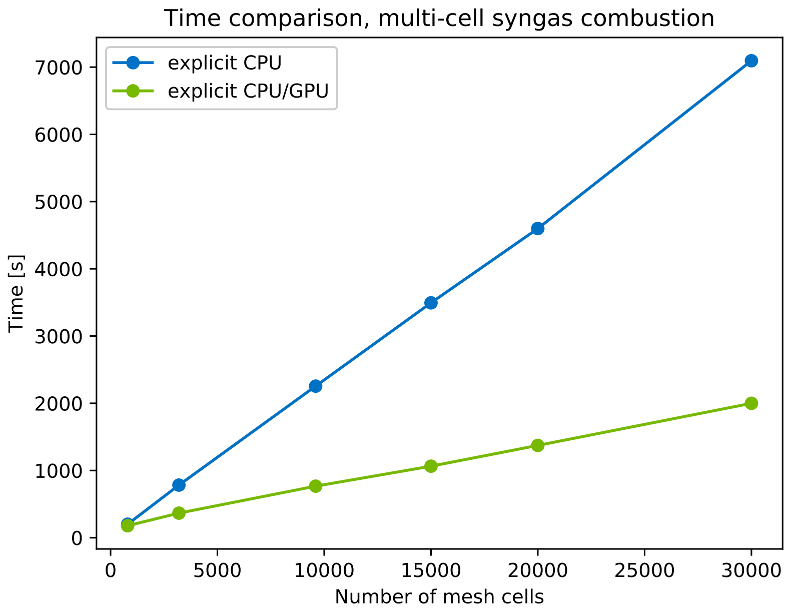


Figure 5.29: Time comparison, multi-cell syngas combustion ($absTol = 1 \times 10^{-13}$, $relTol = 1 \times 10^{-4}$).

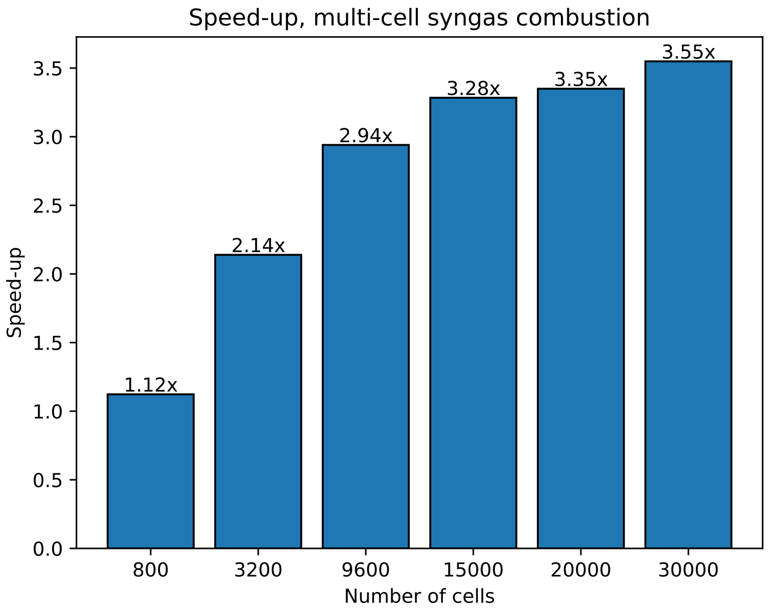


Figure 5.30: Speed-up, multi-cell syngas combustion ($absTol = 1 \times 10^{-13}$, $relTol = 1 \times 10^{-4}$).

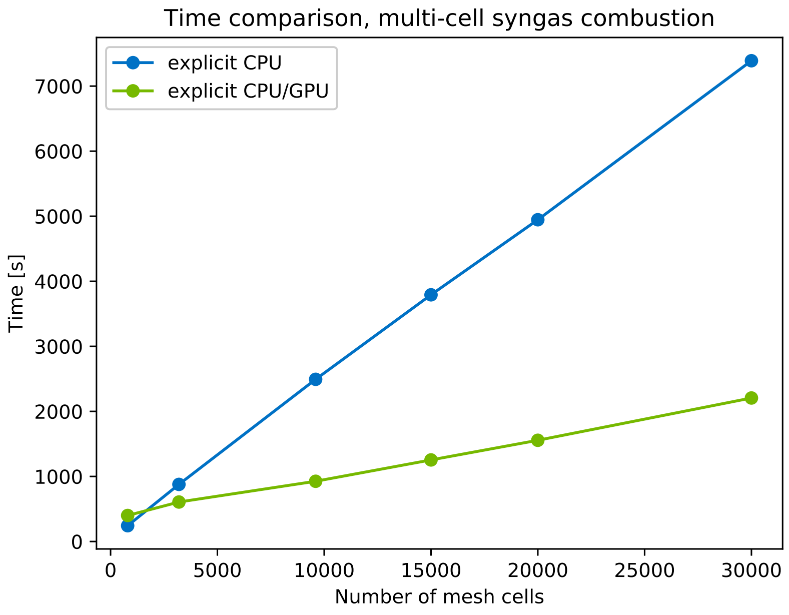


Figure 5.31: Time comparison, multi-cell syngas combustion ($absTol = 1\times10^{-10}$, $relTol = 1 \times 10^{-1}$).

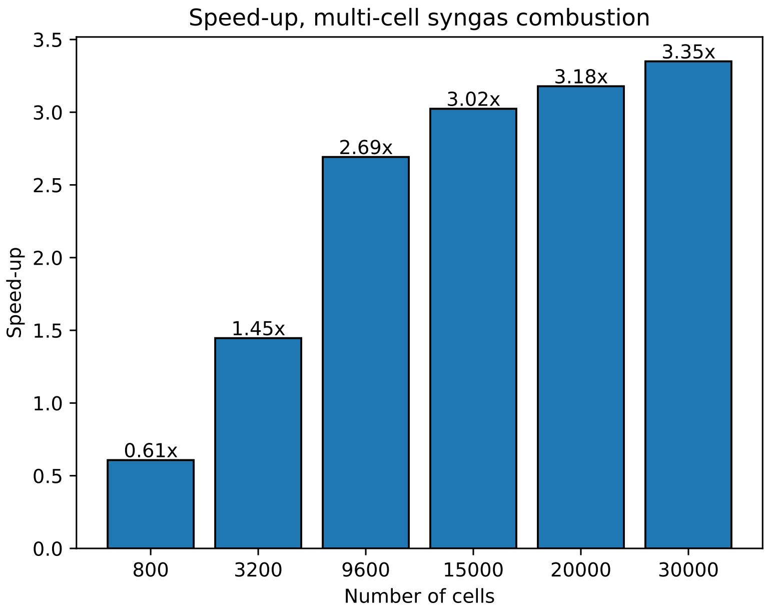


Figure 5.32: Speed-up, multi-cell syngas combustion ($absTol = 1 \times 10^{-10}$, $relTol = 1 \times 10^{-1}$).

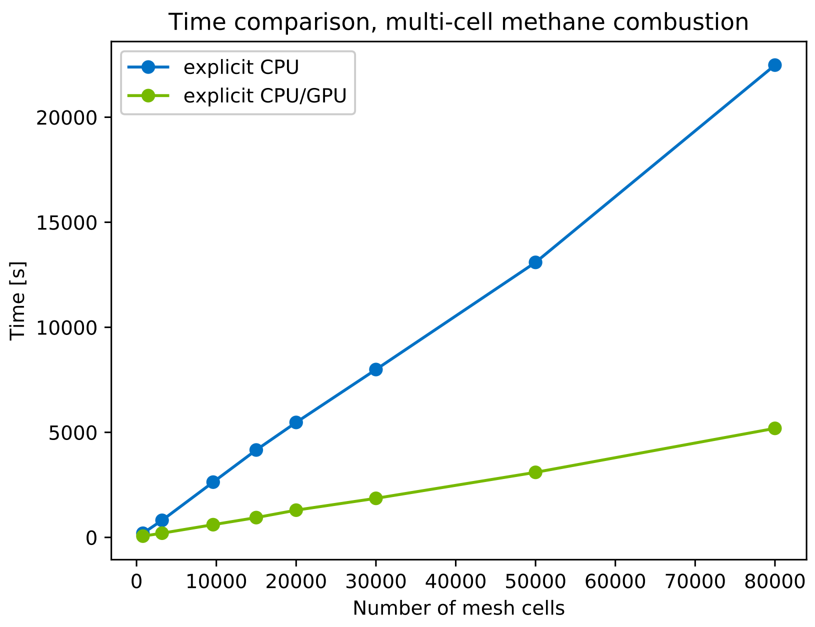


Figure 5.33: Time comparison, multi-cell methane combustion.

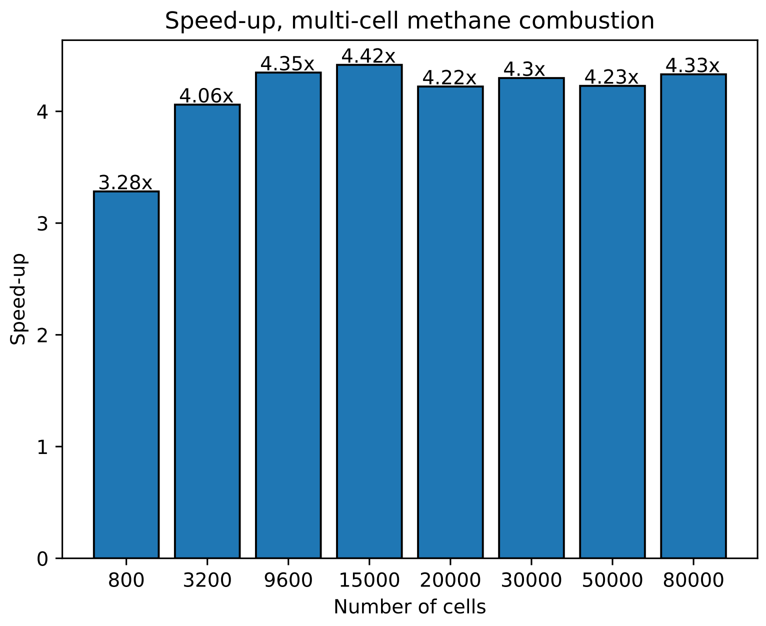


Figure 5.34: Speed-up, multi-cell methane combustion.

Finally, the complex GRI mechanism results are presented in figures 5.33 and 5.34. Here the number of cells tested was increased up to 80000 to show that, after a certain mesh dimension, the speed-up does not increase anymore. The maximum speed-up achieved is about 4.42x with 15000 cells, but for higher number of cells the results are still the same. It is noticeable that also for coarser meshes the reached speed-up is high (greater than 3x). This highlights the difference between explicit CPU/GPU and explicit CPU when simulating complex mechanisms.

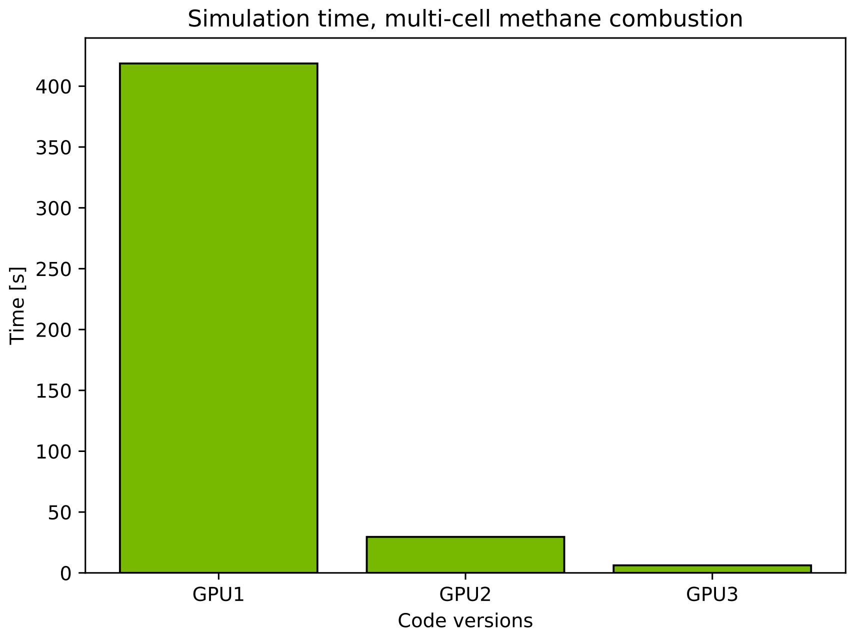


Figure 5.35: Time comparison, multi-cell methane combustion, different GPUs.

A comparison among different GPUs was made for the complete methane combustion mechanism, too. The simulation was run up to $t = 1 \times 10^{-6}s$ in order to execute a sufficiently large amount of time steps. As in the single-cell case, one can see a huge difference among the three devices in terms of computational effort. In this case, the NVIDIA GeForce GTX 1660 SUPER and the NVIDIA TITAN V are 15x and 68x faster than the NVIDIA GeForce 930MX. This confirms that the results obtained are strictly related to the hardware used, since the low-end *GPU1* does not give any improvement in the performance with respect to the CPU original version as *GPU3* and *GPU2* do.

Chapter 6

Conclusion

The project presented in this dissertation had the purpose of showing the potentiality of GPGPU computing applied on CFD, particularly on combustion problems. These class of fluid dynamics phenomena is very complex to simulate numerically. In addition to the fact that the involved chemistry kinetics mechanism can be very complicated and have great mathematical stiffness, the solution of the chemical ordinary differential equations has to be repeated for all the computational cells of the mesh. This can cause huge demand of computational resources, considering also the ODEs integration method used for chemistry treatment. Thus, we wanted to demonstrate that a GPU implementation of the Runge-Kutta Cash-Karp integration method could improve the performance with respect to the original OpenFOAM counterpart, without affecting the accuracy of the results.

The proposed implementation of the explicit RKCK45 CPU/GPU method has revealed itself appropriate to reach the aforementioned goal. After a general overview about how GPGPU computing works, this implementation was described accurately highlighting its main features. In particular, the `thrust` library provided by CUDA allowed to manipulate large arrays of data in a simple manner: for instance, memory is freed automatically at the end of the code thanks to this library and that gives a large benefit in terms of overall computational cost. Other implementation choices were particularly important, resulting in a great increment of the performances. A general code optimization reduced the computational effort, too. Very different test cases were simulated in order to evaluate the code. Hydrogen and methane combustion mechanisms were used for single-cell simulations. Accuracy of the results was checked and small errors on the final solutions were obtained (less than 5%). A general speed-up could not be achieved in these cases since it is not possible to hide memory latency due to the fact that only a single CUDA block is used. On the contrary, an increment of the performance has been obtained in all the multi-cell cases simulated for different number of mesh cells and different stiffness-level chemical mechanisms. With the low-stiffness simple methane combustion we reached a 1.5x speed-up with respect to the CPU original version. Better speed-ups were achieved with the medium-stiffness syngas combustion (CRECK syngas mechanism), up to 3.6x. Finally, a 4.4x speed-up was obtain with the high-stiffness methane combustion (GRI-Mech 3.0).

A natural conclusion can be that GPGPU computing applied on numerical simulations of reactive flows combustion can really decrease the computational effort and preserve the accuracy of the results at the same time. Surely, the explicit method adopted has well-defined mathematical features and it is better applied to low-stiffness-level chemical reaction problems. Thus, an implicit method could be better in terms of time saving for stiff problems with respect to the solution proposed since it uses less high-cost chemical time step per fluid dynamics one.

6.1 Future developments

Despite the implementation adopted showed its potential and good speed-up can be achieved, further developments are needed for improving the code and exploiting all the advantage of GPGPU computing. Here the main features that should be implemented are briefly presented.

1. **`chemFoam` implementation**. It has been underlined that the computational effort can not be reduced with the single-cell chemical kinetics solver. In fact, data are copied to the device and back to host every single chemical time step and the kernel has to access to high-latency global memory several times during the computation since an explicit integration method is used. This causes the code to perform slower than the original explicit

CPU method. A possible solution could be solving all the chemical problem without passing data every time step, giving the final result only. Data will be stored in low-latency memory (registers, shared) that could decrease the computational effort. Another possible solution could be to increase the time step about 10-15 times with respect to the chemical time step in region of low stiffness, while using the chemical time step for high stiffness regions. Thus, the number of iteration would be reduced and the total simulation time could be decreased.

2. **Species limitation.** As showed in 2.2, the maximum number of threads per block is 1024. This means that we can treat reaction mechanisms that have up to 1024 species. In addition to this, shared memory constraint adds a further limitation in the number of species to be processed. A possible solution could be to use more blocks for a single cell or to assign multiple species to a single thread. The latter idea must be implemented considering shared memory and registers limitations.

3. **Multi-CPU.** One might think to increase the speed-up of the simulation by using the further parallelization offered by a multi-core CPU. The computational mesh is divided in multiple sections, one per each core, thanks to the OpenMPI library. Then, each core communicates with the GPU separately and these groups of cells are further divided among the CUDA blocks. Nevertheless, the achievable speed-up is quite low or insignificant due to the fact that data transfer between host and device are limited by the PCIe throughput. Solving this problem could cause a sensible increasing in the performance.

4. **Multi-GPU.** Another way to improve performance by enhancing parallelism could be a multi-GPU system. As with a multi-core CPU hardware, the code should be modified in order to implement multi-GPU usage.

Appendix A

Solvers in OpenFOAM

A.1 chemFoam

`chemFoam` is a solver used for chemistry problems. It is designed for testing chemical mechanisms in a single cell geometry and to compare them with other chemistry solvers (*Chemkin*, *Cantera*, ...). The resolutive process is based on the numerical integration of the chemical kinetics ordinary differential equations; as opposed to other solvers, there is not a fluid dynamics field motion so the momentum conservation has not to be enforced. Furthermore, the pressure stays constant. The principal steps are now presented:

1. update time;

2. solve chemistry ODEs, returning the chemical time step and the heat source $\dot{Q}$ (the algorithm used for the integration of the chemical differential equation it will be presented accurately in appendix B);

3. calculate the integrated heat:

$$Q = \int_0^{t_{end}} \dot{Q} dt \approx \sum_{t=0}^{t_{end}} \dot{Q}_t \Delta t \tag{A.1}$$

4. solve the species conservation equations 3.23 without the convective term;

5. solve the conservation of total entalphy:

$$h = h_0 + Q \tag{A.2}$$

6. if the volume is constant, update the universal gas constant:

$$R_{spec} = 1000R \sum_{k=0}^{N} \frac{Y_k}{W_k} \tag{A.3}$$

   and the pressure:

$$p = \rho R_{spec} T \tag{A.4}$$

A.2 reactingFoam

`reactingFoam` is a transient compressible solver for combustion with chemical reactions. It is very similar to another solver, `rhoReactingFoam`, except for the `psiThermo` treatment of reactions and heat exchange. In fact, the thermopyhsical model for the mixture accounts for compressibility:

$$\Phi = \frac{1}{RT} \tag{A.5}$$

and density is calculated from a closure relation (e.g. the equation of state). On the other hand, `rhoReactingFoam` uses density to compute all thermopyhsical properties. Differences are also on the application: the former is used for complex combustion problems, the latter for heat exchangers.

`reactingFoam` is based on the PIMPLE algorithm, that is a combination of SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) and PISO (Pressure Implicit with Splitting of Operator) algorithms. Both the procedure are based on two main points: the derivation of an equation for pressure from continuity and momentum equations; a corrector step for the velocity field in order to satisfy the continuity constraint [34, 35].

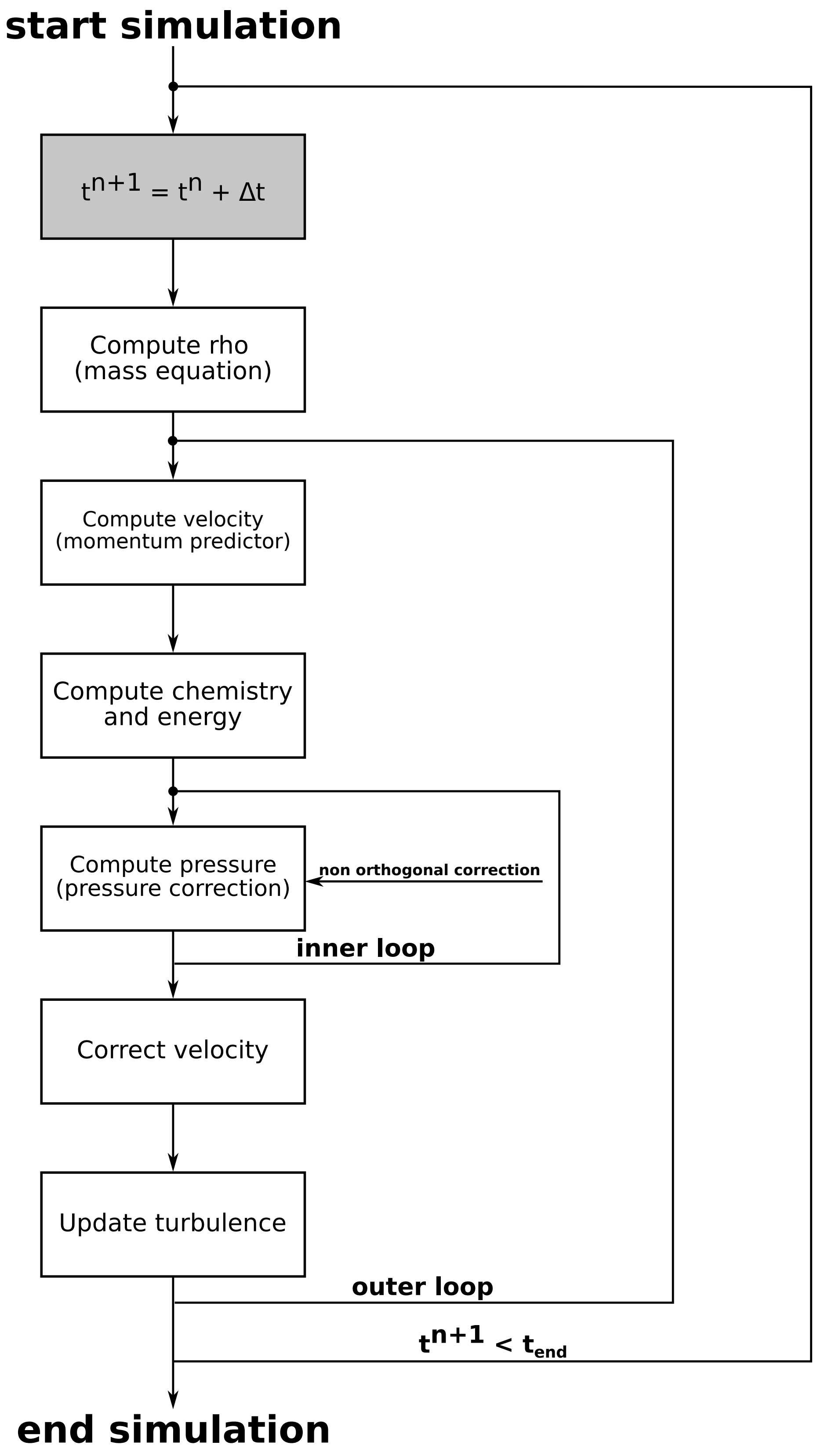

start simulation
$t^{n+1} = t^n + \Delta t$
Compute rho (mass equation)
Compute velocity (momentum predictor)
Compute chemistry and energy
Compute pressure (pressure correction)
non orthogonal correction
inner loop
Correct velocity
Update turbulence
outer loop
$t^{n+1} < t_{end}$
end simulation

Figure A.1: PIMPLE loop.

While the SIMPLE loop restarts the iteration from the momentum predictor in order to guess the velocity field with the pressure update, the PISO loop solves again the pressure equation only. The PIMPLE algorithm (figure A.1) can be imagined as a SIMPLE algorithm for every time step. First of all, the momentum predictor equation is solved and the initial prediction of the velocity field is computed. Then the pressure equation, derived from continuity and momentum equation, is solved and the velocity field is corrected. An inner corrector loop (i.e. the PISO algorithm feature) is used to update the pressure until it converges. On the other side an outer corrector loop is used to control the field velocity convergence. When the outer loop converges, time is updated. The number of the inner and outer loops are defined in *system/fvSolution* through the dictionaries `nCorrectors` and `nOuterCorrectors`. Another correction can be made on pressure when the computational mesh is not orthogonal (`nNonOrthogonalCorrectors`).

The procedure for $t < t_{end}$, available in the file `reactingFoam.C`, is the following:

1. evaluate the time step;
2. enforce the mass conservation and compute an initial density:
$$\frac{d\rho}{dt} + \nabla\Phi = fvOptions(\rho) \tag{A.6}$$
3. start the outer loop;
4. solve the momentum equation;
5. solve the chemical ordinary differential equation, calculate the chemical reaction rates and the integrated heat;
6. if the chemical species is reactive, enforce the conservation of mass species by using the reaction rates previously calculated and updating the mass species:
$$\frac{d\rho Y_i}{dt} + \nabla(\Phi Y_i) - \nabla^2(\mu Y_i) = RR(Y_i) + fvOptions(\rho, Y_i) \tag{A.7}$$

For inert species:

$$Y_{inert} = 1 - Y_{reactive} \tag{A.8}$$

7. solve the energy equation (the form is chosen in *constant/thermophysical-Properties*);
8. start the inner loop (PISO);
9. solve the pressure equation, that can also treat transonic conditions (in this case the equation to be solved is slightly more complex);
10. solve again the mass equation since density has changes due to the variation on p and *phi*;
11. update turbulence parameters;
12. update velocities and fluxes. If p does not converge, solve again the pressure equation (PISO loop);
13. if U does not converge, guess another velocity value and restart the cycle.

Appendix B

Chemistry solution in OpenFOAM

The resolution of chemistry in the two solvers `chemFoam` and `reactingFoam` takes place in different ways even though the resolution method is the same for both the OpenFOAM solvers. In `chemFoam`, this takes place in the *solve-Chemistry.H* file, before the imposition of the species conservation in *yEqn.H* file that requires the solution of the chemical ODEs. In the above file there is `chemistry.solve()`, where `chemistry` is a reference to `pChemistry` in *create-FieldRefs.H* file (that creates new variable referenced by other variables created in *createFields.H* file). `pChemistry` is in turn associated with the *Basic-ChemistryModel* template and by means of `rhoReactionThermo` is generated as `New(const ReactionThermo& thermo)` by the selector in *BasicChemistry-Model.C*. A selector is defined as a virtual function named `New` and it looks through the function parameters to determine the `typeName` of the derived to be constructed, then it uses it to look up in the hask table to return the constructor point. This file in turn invokes *basicChemistryModel.H* file that by *basicChemistryTemplates.C* reads and uses all the thermodynamic and chemistry information contained in the `constant` case.

On the other side, in `reactingFoam` chemistry ODEs is resolved into the PIMPLE loop, in particular through the presence of `reaction->correct()` contained in the *yEqn.C* file. As the above `pChemistry`, `reaction` is defined in *createFields.H* and associated with the `CombustionModel` template. Then, it is generated as:

```
New(
    const ReactionThermo& thermo,
    const compressibleMomentumTransportModel& turb,
    const word& combustionProperties
   )
```

by means of `psiReactionThermo`, with the selector contained in line 47 of *CombustionModel.C*. The class `CombustionModel` is then inherited by the class `ChemistryCombustion`, that uses the class `BasicChemistryModel` through pointer `chemistryPtr_`.

Focusing on the aforementioned `constant` folder, there is the presence of the `chemistryProperties` file where the method for solving the chemical ordinary differential equations is defined. First of all, the `method` is chosen between two possibilities:

- `standard`, that is selected by default and treats the ODEs using the required resolution method, providing the relative coefficients through the dictionaries;
- `TDAC`, that is based on tabulated quantities through which a linear interpolation is made.

Second, a `solver` can be chosen between `EulerImplicit` and `ode`: in particular, the second dictionary collects all the procedure to solve ordinary differential equation (`RKCK45`, `seulex`, `ROsenbrok34`, ...) and the tollerance parameters (`absToll`, `relTol`, ...).

Referring to the latter solver, the connection between the ODE solver and the chemistry model is made in the *standardChemistryModel.C* file where a chemistry variable is built thanks to the connection between the template instance `StandardChemistryModel<CompType,ThermoType>` and the template class from which it was instantiated from. In the above file there is also the presence of three `solve` functions, of which the first has a dependence with *BasicChemistryModel.C* through the function `correct()`. Furthermore, this `solve` function calls the `ode.C` and then the *odeSolver.C* files, showing the aforementioned connection.

B.1 RKCK45

The main steps of the chemistry treatment with explicit RKCK45 method are now presented, referring to figure 3.1.

1. in *StandardChemistryModel.C* file, the second of the three `solve` functions creates the time step that is returned by the last `solve` function.

2. in the same file, the first `solve` is used. Thermodynamic quantities are copied from `thermo` class, then in a `forAll(rho,celli)` cycle species concentrations are built (doing this operation once per cell). After the vector constructions, the computation enters in a `while(timeLeft>small)` cycle, where `this->solve` is present.

3. the above function sends to *ode.C*, where a `cTp_` vector is created with informations about concentrations, temperature and pressure. At the end, an `odeSolver->solve` is present.

4. this function sends to the last `solve` function in *ODESolver.C* file. Here, the chemical time step is created and another `solve` is present.

5. this sends to the penultimate `solve` function in the same file, that in turn sends to *RKCK45.C* file, second `solve` function.

6. another `solve` sends to *adaptiveSolver.C*, where the main core of the integration method starts. The `ode.derivatives` function sends back to *StandardChemistryModel.C* file, where the function for the computation of the derivatives dy/dx, needed for the advancing of the integration method in time, is implemented.

7. in the `derivative` function, there is the presence of `omega` that sends to another function of the standard chemistry file, that in turns sends to *Reaction.C* file where `omega` is finally implemented.

8. after the computation of the derivative, a `do-while` cycle is performed using the error as convergence parameter. In particular, the iteration takes place until the normalized max error is less than unity. Inside the cycle, another

`solve` sends again to *RKCK45.C* file, where the Runge-Kutta algorithm takes place.

9. in *RKCK45.C*, first `solve` function, several `ode.derivatives` are implemented, that sends back to step 6. At the end, *normalizeError* function is returned.
10. this sends again to *ODESolver.C*, where the normalized error that is needed as convergence parameter is returned.
11. this completes first `solve` in *RKCK45.C*.
12. this in turn completes second `solve` function in *adaptiveSolver.C*
13. last two `solve` functions in *ODESolver.C* are completed, as well as *ode.C*.
14. finally, in *StandardChemistryModel.C* the chemical time step and the reaction rates can be constructed and the minimum time step is returned.

Bibliography

[1] Kyle Spafford et al. “Accelerating S3D: a GPGPU case study”. In: *European Conference on Parallel Processing*. Springer. 2009, pp. 122–131.

[2] Kyle E Niemeyer and Chih-Jen Sung. “Accelerating moderately stiff chemical kinetics in reactive-flow simulations using GPUs”. In: *Journal of Computational Physics* 256 (2014), pp. 854–871.

[3] Christopher Stone and Roger Davis. “Techniques for solving stiff chemical kinetics on GPUs”. In: *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*. 2013, p. 369.

[4] Yu Shi et al. “Redesigning combustion modeling algorithms for the Graphics Processing Unit (GPU): Chemical kinetic rate evaluation and ordinary differential equation integration”. In: *Combustion and Flame* 158.5 (2011), pp. 836–847.

[5] Nicholas Curtis. “Accelerating Reactive-Flow Simulations via Vectorized Chemical Kinetic Evaluation”. In: (2019).

[6] Federico Ghioldi. “Fast algorithms for highly underexpanded reactive spray simulations”. In: (2019).

[7] “CUDA C programming guide”. In: *Nvidia Corporation* 107 (2012).

[8] “CUDA C Best practices guide”. In: *Nvidia Corporation* 107 (2012).

[9] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[10] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[11] Gordon E Moore. “Cramming more components onto integrated circuits”. In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[12] Maurice Herlihy. “The multicore revolution”. In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 2007, pp. 1–8.

[13] Karl Rupp. “40 years of microprocessor trend data”. In: *GitHub*. 2018.

[14] John D Owens et al. “GPU computing”. In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[15] Michael J Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[16] Michael J Flynn. “Some computer organizations and their effectiveness”. In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960.

[17] Vasily Volkov. “Better performance at lower occupancy”. In: *Proceedings of the GPU technology conference, GTC*. Vol. 10. San Jose, CA. 2010, p. 16.

[18] Kenneth K Kuo. *Principles of combustion*. TJ254. 5 K85 2005. 2005.

[19] Steven C Chapra, Raymond P Canale, et al. *Numerical methods for engineers*. Boston: McGraw-Hill Higher Education, 2010.

[20] Marc Nico Spijker. “Stiffness in numerical initial-value problems”. In: *Journal of Computational and Applied Mathematics* 72.2 (1996), pp. 393–406.

[21] Charles Francis Curtiss and Joseph O Hirschfelder. “Integration of stiff equations”. In: *Proceedings of the National Academy of Sciences of the United States of America* 38.3 (1952), p. 235.

[22] Jeff R Cash and Alan H Karp. “A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides”. In: *ACM Transactions on Mathematical Software (TOMS)* 16.3 (1990), pp. 201–222.

[23] David G. Goodwin et al. *Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes*. `https://www.cantera.org`. Version 2.4.0. 2018. DOI: `10.5281/zenodo.1174508`.

[24] NM Marinov, CK Westbrook, and WJ Pitz. *Detailed and global chemical kinetics model for hydrogen*. Tech. rep. Lawrence Livermore National Lab., CA (United States), 1995.

[25] Robert J Kee, Michael E Coltrin, and Peter Glarborg. *Chemically reacting flow: theory and practice.* John Wiley & Sons, 2005.

[26] Gregory P Smith. "GRI-Mech 3.0". In: *http://www. me. berkley. edu/-gri_mech/* (1999).

[27] J Bibrzycki and T Poinsot. "Reduced chemical kinetic mechanisms for methane combustion in O2/N2 and O2/CO2 atmosphere". In: *Working note ECCOMET WN/CFD/10* 17 (2010).

[28] Thierry Poinsot and Denis Veynante. *Theoretical and numerical combustion.* RT Edwards, Inc., 2005.

[29] Mary N Bui-Pham. "Studies in structures of laminar hydrocarbon flames." In: (1993).

[30] Alan Kéromnès et al. "An experimental and detailed chemical kinetic modeling study of hydrogen and syngas mixture oxidation at elevated pressures". In: *Combustion and Flame* 160.6 (2013), pp. 995–1011.

[31] Eliseo Ranzi et al. "Reduced kinetic schemes of complex reaction systems: fossil and biomass-derived transportation fuels". In: *International Journal of Chemical Kinetics* 46.9 (2014), pp. 512–542.

[32] Eliseo Ranzi et al. "New reaction classes in the kinetic modeling of low temperature oxidation of n-alkanes". In: *Combustion and flame* 162.5 (2015), pp. 1679–1691.

[33] ELISEO Ranzi et al. "Hierarchical and comparative kinetic modeling of laminar flame speeds of hydrocarbon and oxygenated fuels". In: *Progress in Energy and Combustion Science* 38.4 (2012), pp. 468–501.

[34] Joel H Ferziger, Milovan Perić, and Robert L Street. *Computational methods for fluid dynamics.* Vol. 3. Springer, 2002.

[35] Tobias Holzmann. "Mathematics, numerics, derivations and OpenFOAM®". In: *Loeben, Germany: Holzmann CFD* (2016).

[36] Calvin Lin et al. *Principles of parallel programming.* Pearson Education India, 2008.

[37] Gerassimos Barlas. *Multicore and GPU Programming: An integrated approach.* Elsevier, 2014.