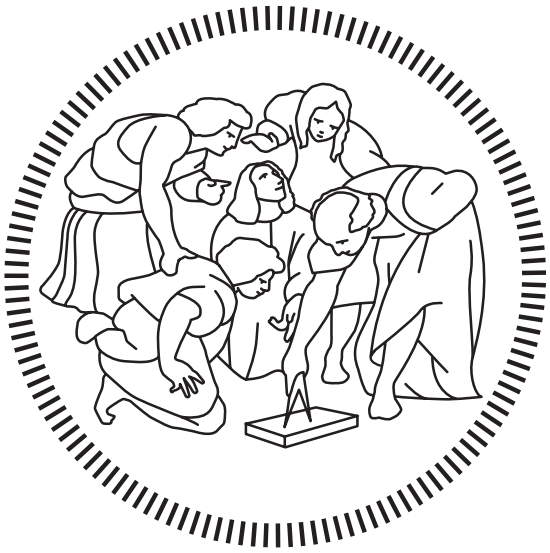
Politecnico di Milano

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING
Master of Science – Aeronautical Engineering



Development of a multi-GPU Navier-Stokes solver

Supervisor
Prof. Maurizio Quadrio

Candidate
Jimmy Vianello – 919832

Academic Year 2019 – 2020

Ringraziamenti

Il ringraziamento più grande va alla mia famiglia, fonte di supporto inesauribile. Le gioie e le delusioni di chi lavora ad un codice sono così altalenanti, ogni giorno un nuovo problema che ieri non c'era va risolto. La vostra pazienza deve essere stata immensa per sopportare tutti i miei cambi improvvisi d'umore, e per questo motivo vi ringrazio. Sorrido mentre ripenso a tutte le volte che avrete sentito ripetere GPU o CPU in questi mesi. Alla fine noi ci raccontiamo proprio tutto, a volte anche troppo. Questo è il nostro modo di fare le cose in famiglia ed è la nostra forza, spero che andremo sempre avanti così. Un grazie anche per avermi fatto passare tanti momenti importanti di spensieratezza in questo anno di convivenza forzata difficile un po' per tutti.
Ringrazio il Prof. Maurizio Quadrio per aver creduto in me da subito e per avermi affidato un argomento che si è rivelato adatto a me e che mi ha appassionato. Ringrazio anche per la grande disponibilità ad aiutarmi in altri progetti, al di fuori della tesi, che altrimenti non sarei riuscito a fare. Un ringraziamento va anche al Dr. Alessandro Chiarini e al Prof. Franco Auteri, che hanno condiviso il loro lavoro con me e hanno reso possibile questa tesi.
Infine ringrazio tutti i miei amici, quelli di università e quelli dell'isola. I 5 anni al Politecnico sono stati veramente impegnativi e senza amici non penso ci sia possibilità di successo. Ammetto che ci sono state tante occasioni limitate per colpa dello studio, ma posso assicurarvi che ho comunque tantissimi ricordi incredibili che mi hanno aiutato quando meno ve l'immaginate.

Sommario

Un nuovo solutore alle differenze finite collocate per le equazioni di Navier-Stokes incomprimibili, che sfrutta il metodo del direction-splitting proposto da Guermond e Minev nel 2010, sviluppato da Chiarini A., Quadrio M. e Auteri F., è stato portato su GPU clusters per poter sfruttare la potenza computazionale dei più recenti supercomputers. Lo sviluppo del codice è stato eseguito utilizzando CUDA Fortran e prendendo come target il nuovo cluster Marconi100, accelerato dalle NVIDIA Tesla V100 GPUs, presente al CINECA. La caratteristica principale del solutore è quella di effettuare l'intero time-loop esclusivamente sulle GPUs utilizzando kernels implementati ad hoc per ottenere il massimo rendimento possibile nelle parti computazionalmente intensive dell'algoritmo. La comunicazione è stata gestita attraverso l'utilizzo della libreria NCCL, ottimizzata da NVIDIA per aumentare la portabilità e scalabilità delle applicazioni multi-GPU. I risultati ottenuti sono stati confrontati con la versione CPU e sono identici a precisione macchina, ciò indica che le due versioni del codice sono consistenti. Infine è stato eseguito uno studio di scalabilità utilizzando una soluzione "costruita" delle equazioni di Navier-Stokes, i cui risultati sono stati integrati con quelli precedentemente ottenuti al CPU cluster Galileo (CINECA).

Parole Chiave: GPU; CUDA; computazione parallela; calcolo ad alte prestazioni; Navier-Stokes; Differenze finite; metodo a passi frazionari; Complemento di Schur;

Abstract

A new co-located finite-difference solver for the incompressible Navier-Stokes equations, which exploits the direction-splitting method proposed by Guermond and Minev in 2010, developed by Chiarini A., Quadrio M. and Auteri F., has been ported to GPU clusters in order to harness the computational power of the most recent supercomputers. The development of the code was performed using CUDA Fortran and targeting the new Marconi100 cluster, accelerated by NVIDIA Tesla V100 GPUs, present at CINECA. The main feature of the solver is to perform the entire time-loop exclusively on the GPUs using kernels implemented *ad hoc* to obtain the maximum possible performance in the computational-intensive parts of the algorithm. Communication was managed through the NCCL library, optimized by NVIDIA to increase the portability and scalability of multi-GPU applications. The results obtained were compared with the CPU version and are identical to machine precision, which indicates that the two versions of the code are consistent. Finally, a scalability study was performed using a manufactured solution of the Navier-Stokes equations, integrating the results with those previously obtained on the CPU cluster Galileo (CINECA).

Key Words: GPU; CUDA; Parallel Computation; High Performance Computing (HPC); Navier-Stokes; Finite-Differences; Fractional-step method; Schur-Complement;

Contents

List of Figures

List of Tables

Introduction

The scientific and engineering necessity to model increasingly complex systems and phenomena requires a continuous increase in the computational power available. This is particularly true in the case of fluid dynamics simulations. When it comes to extensive computational demands in fluid dynamics, the typical application that is taken as an example is the *direct numerical simulation* (*DNS*) of turbulence. It consists in solving the Navier-Stokes equations, resolving all the scales of motion without using a model of turbulence. It has been shown that the cost of this simulation increases with the Reynolds number, according to the estimate Re^3. It is clear that for real flows with high Re numbers such a stringent demand for hardware computational capacity is still out of reach for many researchers and industries. This stringent demand for hardware computational capacity makes the study of many real-world flows at high Re still out of reach even for powerful supercomputers.
Before the start of 2000, computer software was primarily written for serial computing, as a serial stream of execution of one or more algorithms that had to be executed on a central processing unit CPU. Moore's observation stated that every 18 months the number of transistors in a microprocessor would double, providing a speedup for the application developed in that time without further implementation effort. This increase in computational power has been at the heart of many advances in different fields. Since 2003, the industry has reached a limit in improving the processor as a single processing unit due to power consumption and heat dissipation issues. It was no longer possible to continue on the path of merely increasing the clock frequency. From this moment forward, the industry has turned in the direction of *parallelism*; rather than trying to develop faster monolithic processors, manufacturers started putting multiple processors on a single integrated circuit. In particular they concentrate on two different concepts for the design of subsequent microprocessors: the *multi-core*, where 2 or more processing units are integrated in the same chip, and the *many-thread* processor, focus on optimizing the execution throughput of a huge number of threads. It was a very important change for software developers: simply sequential program will only run on one of the processor cores, which will not become significantly faster from generation to generation. Rather, applications that will benefit an increase in performance will be those that will adopt a parallel programming model.
Graphics processing units (GPUs) are an example of a multi-threaded processor. They were originally developed for graphics/video processing and displays, where millions of pixels have to be displayed on screen simultaneously. The fact that they required special programming skills, related exclusively to the world of graphics, had discouraged most programmers interested in their computational power for general

purposes. In 2007, NVIDIA introduced the CUDA programming model and toolkit to allow developers to harness the power of GPUs to dramatically speed-up computing applications. The ratio between peak floating-point computation throughput of multi-threaded GPUs and multi-core CPUs has remained roughly constant in recent years and around a factor 10. Such a gap potentially allows a computation to be concluded earlier using the GPUs and therefore translates into energy savings. Electricity savings, together with development of CUDA and other programming models, are the main reasons that have pushed computer centers to invest more in GPU acceleration.
For all the above reasons, knowing how to develop applications that are able to exploit the potential of GPUs has become essential, especially now that they are such a popular accelerator technology in supercomputing. It is also the reason that prompted the following thesis work to develop a GPU version of the solver implemented by A.Chiarini, M.Quadrio and F.Auteri presented in [1]. It is a co-located finite difference solver for the incompressible Navier-Stokes equations based on a new direction-splitting method proposed for the first time by Guermond and Minev [2]. The main feature is to use the direction-splitting technique, similar to ADI, on the pressure step; the Laplacian operator in the Poisson equation is replaced with a more general operator. The advantage of this technique is that the algorithm requires only the solution of 1D tridiagonal linear systems, improving code efficiency. The solver has been parallelized using the Schur-complement method and already achieves high performance on thousands of processors.
The use of GPUs should further increase the performances respect to the CPU version and should provide excellent performances in large-scale grids that would be difficult to reach, unless a large number of CPUs are used.
The thesis is organized as follow:

- *Chapter* 1 contains a general description of parallel computation technology used to develop the solver. It describes the architecture of a GPU, detailing the differences from CPU. The CUDA programming model is presented. The material is adapted from these books: [3], [4], [5], [6];
- *Chapter* 2 contains an overview of the numerical methods underlying the solver. Specifically, the algorithm and the parallel implementation at the base of the CPU version, following [1];
- *Chapter* 3 is the heart of the work and contains a description of the software code. The main parts of the program are discussed with an explanation of the choices that were made during development to extract the best performance from GPUs;
- *Chapter* 4 contains an overview of the results and performances of the code compared with the CPU version. A 3D driven cavity test case is simulated and the obtained solution compared with the CPU-version solution. The results of the code scalability tests performed at Cineca on supercomputer Marconi100 are also presented;

Chapter 1

Parallel Computing

Parallel computing is a computation method that consists in performing many calculations or many processes simultaneously, based mainly on the concept of *divide et impera*. A computational task is divided into many smaller independent sub-tasks that run simultaneously on multiple processors.
There are two widely used approaches to partition the work to be done between multiple processors: *task-parallelism* and *data-parallelism*. Data-parallelism occurs when we can simultaneously perform calculations around small pieces of different data; what we want to do is divide the data among the various cores. Task-parallelism occurs when there are many tasks or functions that can be performed independently; its focus is therefore to distribute the various tasks among the cores. While data-parallelism consists in executing the same instruction simultaneously for different components of the data, task-parallelism executes different processes simultaneously on the same or different data.
Parallelism was exploited at the hardware level through the development of parallel computer architectures. The *Flynn's taxonomy* is frequently used in order to classify the different possible architectures. It classifies a system in terms of the number of concurrent instructions (or controls) and data streams available in the architecture. A classic Von Neumann system is of the *Single Instruction, Single Data* (SISD) type since it executes a single instruction at a time and fetch or store one data at a time. *Single Instruction, Multiple Data* (SIMD) describes, on the other hand, systems that have multiple processing units capable of executing the same instruction on different elements of the data simultaneously. It is therefore a class of parallel computers suitable for exploiting data-parallelism but limited to a single instruction stream. Machines that support multiple simultaneous instruction streams operating on multiple data streams are classified as *Multiple Instruction, Multiple Data* (MIMD) architectures. They are systems based on a collection of fully asynchronous and independent processing units or cores, each of which has its control unit and its arithmetic logic unit.
There are mainly 2 types of MIMD systems: *shared-memory systems* and *distributed-memory systems*. A shared-memory system consists of a set of processors connected to a memory system through an interconnection network. They share the same physical memory, therefore they implicitly communicate with each other by access-

ing shared data-structures. On the contrary, in a distributed-memory system each processor is coupled with its own memory. An interconnection between the various processor-memory pairs of the system is required, which is generally developed by sending messages, i.e. explicitly communicating, or by using special direct access to the memory of another processor.
Hardware parallel technologies, which are not directly under the programmer's control, have also been developed: such as *instruction-level parallelism* (ILP) and *thread-level parallelism* (TLP). Both for this reason can be seen as extensions of the basic von Neumann model. The ILP attempts to improve processor performance by having multiple functional units that simultaneously execute instructions. *Pipelining* is a way to perform ILP and consists of dividing the instruction flow into stages each performed by different functional units in parallel. The TLP, on the other hand, provides parallelism through the simultaneous execution of different threads. It is a coarser-grained parallelism with respect to the ILP because it is logically structured on threads, which are real processes with their own instructions and data. *Hardware multithreading* is a possible strategy for exploiting TLP. It consists in quickly switching threads that are waiting for a time-consuming operation, with threads that are ready for execution.
The GPU is an architecture that supports various types of parallelism that have been presented previously: instruction-level parallelism, multi-threading, Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD).

1.1 Architecture of a modern GPU

NVIDIA has summarized the execution model of a general purpose GPU with the expression *Single Instruction Multiple Thread* (SIMT). There is a subtle important difference between SIMD and SIMT. SIMD is typically characterized by a scalar thread executing the same operation in parallel across many data elements. In a SIMT architecture, rather than a single thread issuing vector instructions applied to data vectors, multiple threads issue common instructions to arbitrary data. The big advantage over SIMD is that each thread can access its own registers, can load and store from divergent addresses, and can follow divergent control paths.
Fig.(1.1) shows a high level view of the architecture of a typical CUDA-capable GPU.

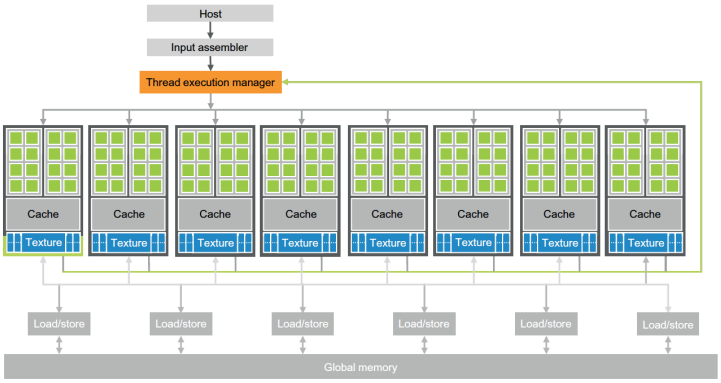


Figure 1.1. Architecture of a CUDA-capable GPU. [4]

It is structured in an array of highly threaded *streaming multiprocessors* (SMs) designed to support the execution of multiple threads simultaneously. Each SM contains: thousand of registers that can be partitioned among threads of execution, several caches, warp schedulers that can quickly switch context between threads, and floating-point execution cores. Each GPU currently comes with gigabytes DRAM, referred to as *Global Memory* in Fig.(1.1). For computing, global memories function as very high-bandwidth off-chip memories, characterized by longer latency than typical system memory. The communication with the CPU (host) is performed via the PCIe bus. Through PCIe, an application can transfer data from the system memory to the global memory. Peak bandwidth guaranteed by PCIe is generally much lower than that of global memory (PCIe Gen2 or Gen4 supports 8-16 GB/s in each direction). Newer architectures also support NVLink, which is a CPU-GPU or GPU-GPU interconnect that guarantees higher bandwidth than PCIe (40GB/s per channel).

1.1.1 Volta GPU architecture

The main feature of the Volta architecture is the new concept of Streaming Multiprocessor (SM) developed to guarantee an improvement in performance by simplifying programmability. Tesla V100 contains 80 SMs, each divided into 4 processing blocks consisting of: 16 FP32 Cores, 8 FP64 Cores, 16 INT32 Cores, two mixed precision Tensor Cores for matrix arithmetic, a new $L0$ cache, one warp scheduler, one dispatch unit and 64 KB register file.
Each SM features 128KB of $L1$ data cache combined with shared memory. Merging $L1$ and shared memory together allows $L1$ cache operations to benefit from shared memory performance. Shared memory provides high bandwidth, low latency, no cache misses but must be explicitly controlled by the programmer. By combining L1 with shared memory, Volta architecture aims to reduce the gap between applications that explicitly use shared memory and applications that access data in device memory directly. The amount of cache dedicated to shared memory can be set at runtime and range of available capacities goes from 0 to 96 KB for SM. There is also a 6144 KB unified $L2$ cache for data, instructions, and constant memory.
Compared to the GPUs based on GDDR5 as global memory, the $V100$ thanks to HBM2 memory has a much higher bandwidth. It provides 900 GB/s of peak memory bandwidth and reduces the gap between actual and theoretical performance compared to what experienced on Pascal architecture. These are some of the main features of the Volta, it is recommended viewing [7] and [8] for a complete discussion.

1.2 Heterogeneous Computing

Homogeneous computing uses multiple processors of the same architecture to execute a program. On the contrary, *heterogeneous computing* makes use of different processor architectures to run an application, taking advantage of the architecture that performs best depending on the task.
The GPU is a device that is placed side by side with the CPU and which is not suitable for working independently. It essentially performs a co-processor function in a heterogeneous environment, accelerating the pieces of code that are compute-intensive.

To understand this usage difference between many-threaded GPUs and multi-core CPUs, it is necessary to compare the two different design philosophies. As shown

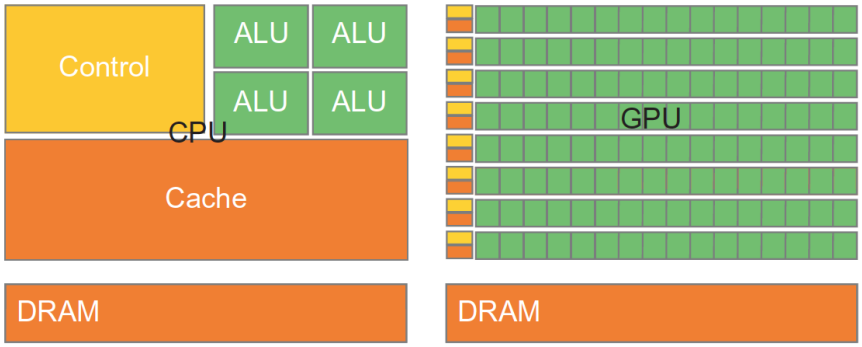


Figure 1.2. CPUs and GPUs different architectures. [4]

schematically in Fig.(1.2), the CPU is equipped with a large cache memory and a sophisticated logic control unit unlike the GPU. The cache reduces the long latency, due to memory accesses, by saving data that is accessed frequently. The latter directs instructions inside a processor arriving from a program. Neither the control logic unit nor the cache memory contribute to peak calculation throughput. The CPU is in fact designed following a *latency-oriented* vision. Both components are also present in the GPU, as we can see from Fig.(1.2), but in a reduced way, leaving the chip area to the arithmetic logic units. This is an example of *throughput-oriented* design, because it acts to maximize the throughput rather than investing in latency. The hardware takes advantage of the large number of threads to find work to do, even when some are waiting because of long latency memory accesses.
Due to these differences it is clear that both microprocessors will perform well in different situations and neither of them will replace the other in the near future. While the CPU is great for control-intensive tasks, the GPU is great for data-parallel computation-intensive tasks. Nowadays, many of the HPC clusters have multiple hosts and devices in each node. The possibility to take advantage of this was a reason that brought NVIDIA to develop CUDA.

1.3 CUDA

CUDA (acronym for Computer Unified Device Architecture) is a general purpose parallel computing platform developed by NVIDIA to allow programmers to exploit the computing power of CUDA-enables graphics processing unit (GPU). The CUDA platform is accessible through CUDA-accelerated libraries (such as cuBLAS, cuSOLVER, cuFFT,...), compiler directives (such as OpenACC) and extension to most utilized industry-standard programming languages (such as C, C ++, Fortran).
CUDA provides an application programming interface model (API) which consists of both a low level API (CUDA driver API) and a higher level API (CUDA Runtime API). An API is a computing interface that allows the programmer to perform functions without knowing exactly the underlying hardware implementation. While the

CUDA driver API allows for more extensive control of the GPU but requires more programming effort. The runtime API has been implemented on top of the driver API and each of its functions is divided into many basic operations performed by the driver API.

1.3.1 CUDA Thread Organization

Within CUDA programming model, the core element is the *kernel*, i.e. the portion of code that runs exclusively on the GPU. When a kernel is called, a grid of threads is launched, each running the same kernel. A good application typically runs 5000 to 12000 threads simultaneously on the GPU.
To exploit such an extensive execution activity, the CUDA programming model provides an organizational structure of the threads. They rely on coordinates to distinguish themselves and identify the appropriate portion of the data to be processed. As shown in Fig.(1.3), threads are organized in a 2-level hierarchy: a grid consisting of blocks, and blocks consisting of threads. Threads belonging to the same block have the same *blockIdx* value. While each thread in the block has its own thread index,

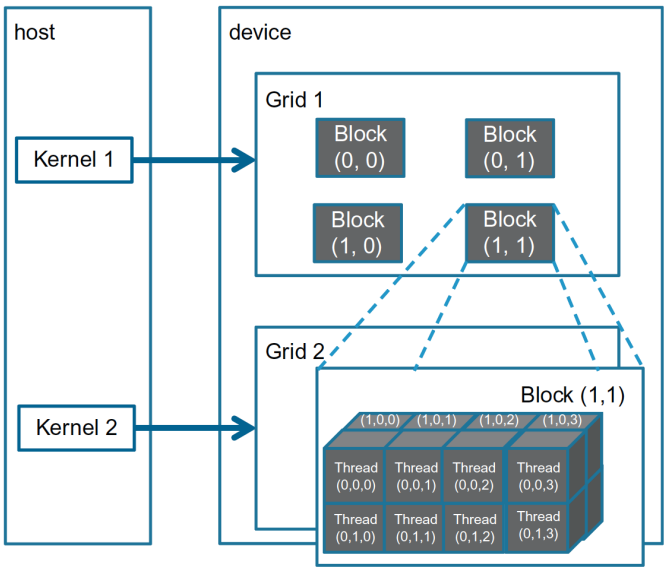


Figure 1.3. Threads hierarchy; 2D and 3D blocks grid. [4]

which is saved in the value of the *threadIdx* variable. When a thread executes in a kernel function, we can use blockIdx and threadIdx to get the thread coordinates and use them to exploit data parallelism. On the other hand, it is not possible to make assumptions about the order of execution of the blocks. Each block operate on a different part of the data and is executed in arbitrary order.
The size of the grid and the number of threads for each block are parameters that can be changed through the *execution configuration parameters*, as indicated in the following kernel call:

kernel_name <<< grid, block, [sharedSize], [stream]>>> (args)

where:

- **grid**: data structure that specifies the number of blocks in each dimension;
- **block**: data structure that specifies the dimension and size of each block, i.e. the number of threads in each dimension;
- **sharedSize**: optional argument that specifies the number of bytes of shared memory dynamically allocated per block in addition to the statically allocated memory;
- **stream**: optional argument that specifies the stream (if missing the default value is 0). A stream is simply a sequence of operations that are performed in order on the device. Operations in different streams can be interleaved and in some case overlapped, property useful for improving performances.

A block is also divided into *warps* generally consisting of 32 threads. A multiprocessor streaming is designed to execute all threads in a warp according to the SIMT model. In general, each multiprocessor streaming is overloaded with a higher number of warps than its streaming processors. By assigning a large number of warps, the hardware will more easily find a warp that is not waiting due to latency. This ability to tolerate long-latency operations is the main reason why GPUs have not dedicated chip areas to cache memories and branch prediction mechanism like the CPU does. This area was rather used to have more floating-point execution resources.

1.3.2 CUDA Memory Organization

Although the multitude of threads is able to hide latency, it is still possible to run into a congestion situation in the access path to the global memory. Programs whose execution is limited by memory access throughput are called *memory-bound* programs. CUDA architecture provides a number of resources and methods to get around this problem. CUDA-capable devices are designed with a memory hierarchy of progressively lower-latency but lower-capacity memories to optimise performance, as indicated in the Fig.(1.4). This is combined with greater exposure of the programmer to memory control by the CUDA programming model. Even if each GPU generation presents some peculiarities on the memory hierarchy, this paragraph is intended to provide general information.
Through the declaration of the variables we can use keywords (*device*, *shared*, *constant*, etc.) that allow us to decide in which memory the data will reside. Furthermore, the declaration to a given memory gives the variable a lifetime and a scope. The *Scope* identifies the range of threads that can access the variable: a single threads, a block, a grid of threads. *Lifetime* indicates the execution portion of the program in which the variable is usable (a kernel, the entire program, etc.).
Let's look more specifically at the various types of memory that exist in the GPU:

- **Global Memory**, technically device random access memory (DRAM). it's a high-bandwidth, long latency off-chip memory. It is the largest memory in the GPU. It can be read and written by both hosts and devices. It is accessible from all threads that are launched on the device for the entire duration of the application (or at least until a variable is deallocated);
- **Constant Memory**, used to save data which does not change over the life of

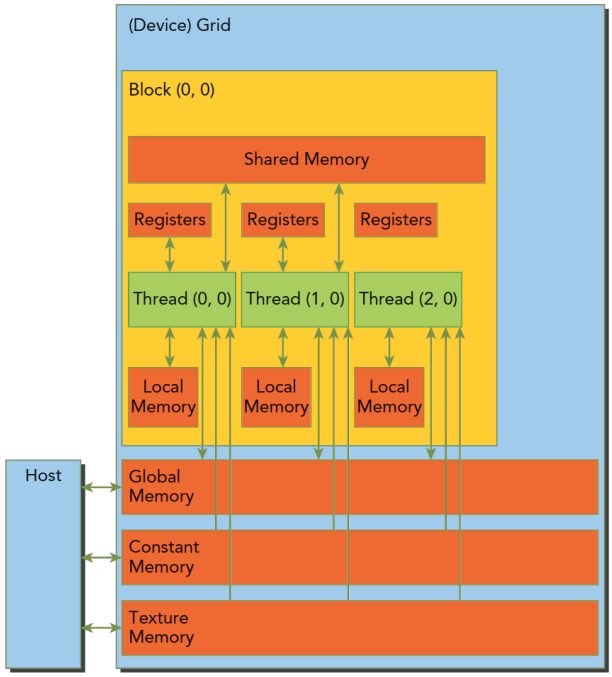


Figure 1.4. CUDA schematic memory view.

the program. It resides in device memory, but is cached in a read-only cache optimized to broadcast the results of read request to threads that all reference the same memory location;

- **Texture Memory**, it allows us to access global memory in a read-only fashion through the texture cache. Textures turn out to be very advantageous when sequential threads access strided data. The texture cache is optimized for 2D and 3D data locality;
- **Shared Memory**, unlike global memory, is on-chip. This means that it is a memory characterized by a higher bandwidth and lower latency than global memory. It is the memory that is mainly used when the global memory access patterns are inefficient. Each SM has a limited amount of shared memory which is split between blocks. Its lifetime and scope is the block. It is arranged as interleaved banks and generally optimized for 32-bit access. If more than one thread references the same bank, a bank conflict occurs, and the hardware manages the accesses sequentially;
- **Registers**, is a small on-chip memory that is divided between threads when the kernel is launched. It is the most plentiful and efficient memory of the entire device, it is used to hold data that is exclusive to each threads (such as the threadIdx and blockIdx indexes);
- **Local Memory**, is the memory used to contain the register spills, i.e. variables that need to be saved when registers reach storage limit. It is an off-chip memory with characteristics similar to those of global memory.

Chapter 2

Numerical Methods

The code considered in the following work is a finite difference solver for the incompressible Navier-Stokes based on a new fractional step method proposed by Guermond and Minev. In the literature, fractional step methods are often known as projection methods, and can be classified into 3 classes: the Pressure-Correction, the Velocity-Correction, and the Consistent Splitting methods. The chapter begins with an overview of the pressure-correction methods underlying the proposed new algorithm, following [9]. Next, the key elements of the solver will be covered, in accordance with [1]: the time discretization adopted, the fractional step algorithm, the formulation of the non-linear term, the Schur's complement method for parallelization.

2.1 Pressure-Correction methods

One of the major problems for numerical simulations of incompressible flows is that velocity and pressure are coupled by the incompressibility constraint. The characteristic of projection methods is to separate the incompressibility constraint from the momentum equation. This splitting operation is a consequence of the following theorem:

Theorem 2.1.1 (Helmholtz decomposition) *Let* $\Omega \in \Re^d$ *simply connected, then any functions* $\tilde{\mathbf{u}} \in [L^2(\Omega)]^d$ *can be decomposed as* $\tilde{\mathbf{u}} = \mathbf{u} + \nabla\phi$*, with* $\phi \in H^1(\Omega)$ *and* $\mathbf{u} \in H = \{\mathbf{v} \in [L^2(\Omega)]^d : \nabla \cdot \mathbf{v}\}$

Therefore, the total cost per time step is that of solving one vector-valued advection-diffusion equation for $\tilde{\mathbf{u}}$, and one scalar-valued Poisson equation with homogeneous Neumann boudary condition for ϕ, in order to project $\tilde{\mathbf{u}}$ into the space of solenoidal velocities.
Since the nonlinear term in the Navier–Stokes equations does not affect the convergence rate of the splitting error, we shall mainly be concerned with the time-dependent Stokes equations, while the discussion of the non-linear term will be carried out later. On a finite time interval [0,T] and in a cubic domain, $\Omega \subset \Re^3$, the equations are:

$$\begin{cases} \frac{\partial \mathbf{u}}{\partial t} + \nabla p - \nu \nabla^2 \mathbf{u} = \mathbf{f}\,, & in\ \Omega \times (0,T] \\ \nabla \cdot \mathbf{u} = 0\,, & in\ \Omega \times (0,T] \\ \mathbf{u}|_{t=0} = \mathbf{u_0}\,, & in\ \Omega \\ \mathbf{u}|_{\partial\Omega} = 0\,, & in\ \partial\Omega \times (0,T] \end{cases} \tag{2.1}$$

where $\mathbf{f}$ is a smooth source term and $\mathbf{u}_0 \in H$ is an initial solenoidal velocity field with zero normal trace at the boundary Ω.

2.1.1 Chorin-Themam method

The simplest pressure-correction scheme has originally been proposed by Chorin and Temam, in 1960 [10], [11]. Choosing Implicit Euler for the time-discretization, the algorithm is as follow: let $\Delta t > 0$ be a time-step and set $t^n = n\Delta t$ for $0 \le n \le \frac{T}{\Delta t}$, compute $(\tilde{\mathbf{u}}^{n+1}, \mathbf{u}^{n+1}, p^{n+1})$

$$\begin{cases} \frac{1}{\Delta t}(\tilde{\mathbf{u}}^{n+1} - \mathbf{u}^n) - \nu \nabla^2 \tilde{\mathbf{u}}^{n+1} = \mathbf{f}^{n+1} \\ \tilde{\mathbf{u}}^{n+1}|_{\partial\Omega} = 0 \end{cases} \tag{2.2}$$

$$\begin{cases} \frac{1}{\Delta t}(\mathbf{u}^{n+1} - \tilde{\mathbf{u}}^{n+1}) + \nabla p^{n+1} = 0 \\ \nabla \cdot \mathbf{u}^{n+1} = 0, \qquad \mathbf{u}^{n+1} \cdot \mathbf{n}|_{\partial\Omega} = 0 \end{cases} \tag{2.3}$$

At the generic time-step t^{n+1}, the first sub-step accounts for viscous effects, while the second sub-step can be written as a Poisson problem and accounts for incompressibility constraint. Taking the divergence of (2.2), and using $\nabla \cdot \mathbf{u}^{n+1} = 0$ leads to:

$$-\nabla^2 p^{n+1} = \nabla \cdot \left(\frac{1}{\Delta t}(\mathbf{u}^{n+1} - \tilde{\mathbf{u}}^{n+1})\right) = -\frac{1}{\Delta t} \nabla \cdot \tilde{\mathbf{u}}^{n+1} \tag{2.4}$$

Therefore we can rewrite the second sub-step as follows:

$$\begin{cases} -\nabla^2 p^{n+1} = -\frac{1}{\Delta t} \nabla \cdot \tilde{\mathbf{u}}^{n+1} \\ \frac{\partial p}{\partial \mathbf{n}} = 0 \end{cases} \tag{2.5}$$

$$\mathbf{u}^{n+1} = \tilde{\mathbf{u}}^{n+1} - \Delta t \nabla p^{n+1} \tag{2.6}$$

The second sub-step is usually referred to as the projection step, because it is a realization of the identity $\mathbf{u}^{n+1} = P_H \tilde{\mathbf{u}}^{n+1}$, with $H = \{\mathbf{v} \in [L^2(\Omega)]^3 : \nabla \cdot \mathbf{v}\}$ and $P_H : [H^1(\Omega)]^3 \to H$.

2.1.2 Standard Incremental Pressure-Correction

In Chorin-Temam method, a term concerning the pressure gradient is missing. Goda [12] was the first to observe an increase in accuracy adding an old value of the pressure gradient in (2.2), say ∇p^n, and modifying the correction sub-step accordingly. The idea was popularized by Van Kan who proposed a second-order incremental pressure-correction scheme in [13]. Using the Backward Difference Formula of second-order (BDF2) to approximate time-derivative, leads to the following algorithm:

$$\begin{cases} \frac{1}{2\Delta t}(3\tilde{\mathbf{u}}^{n+1} - 4\mathbf{u}^n + \mathbf{u}^{n-1}) - \nu\nabla^2\tilde{\mathbf{u}}^{n+1} + \nabla p^n = \mathbf{f}^{n+1} \\ \tilde{\mathbf{u}}^{n+1}|_{\partial\Omega} = 0 \end{cases} \tag{2.7}$$

$$\begin{cases} \frac{3}{2\Delta t}(\mathbf{u}^{n+1} - \tilde{\mathbf{u}}^{n+1}) + \nabla(p^{n+1} - p^n) = 0 \\ \nabla \cdot \mathbf{u}^{n+1} = 0 \\ \tilde{\mathbf{u}}^{n+1} \cdot \mathbf{n}|_{\partial\Omega} = 0 \end{cases} \tag{2.8}$$

Through the same procedure performed for the Chorin-Temam method

$$\begin{cases} -\nabla^2(p^{n+1} - p^n) = -\frac{3}{2\Delta t}\nabla \cdot \tilde{\mathbf{u}}^{n+1} \\ \frac{\partial(p^{n+1} - p^n)}{\partial \mathbf{n}} = 0 \end{cases} \tag{2.9}$$

the second sub-step is still a projection step since it is equivalent to $\mathbf{u}^{n+1} = P_H\tilde{\mathbf{u}}^{n+1}$. From (2.9) it is possible to observe that $\nabla(p^{n+1} - p^n) \cdot \mathbf{n}|_{\partial\Omega}$, which implies $\nabla p^{n+1} \cdot \mathbf{n}|_{\partial\Omega} = \nabla p^n \cdot \mathbf{n}|_{\partial\Omega} = \ldots = \nabla p^0 \cdot \mathbf{n}|_{\partial\Omega}$, is an artificial Neumann boundary condition that introduce a numerical boundary layer limiting the accuracy of the scheme.

2.1.3 Pressure-Correction, Rotational Form

The problem pointed out previously was overcome by the algorithm proposed by Timmermans, Minev and Van De Vosse in [14]. To describe the method we consider the Stokes problem discretized in time using the implicit BDF2:

$$\begin{cases} \frac{3\mathbf{u}^{n+1} - 4\mathbf{u}^n + \mathbf{u}^{n-1}}{2\Delta t} + \nabla p^{n+1} - \nu\nabla^2\mathbf{u}^{n+1} = \mathbf{f}^{n+1} \\ \nabla \cdot \mathbf{u}^{n+1} = 0 \\ \mathbf{u}^{n+1}|_{\partial\Omega} = 0 \end{cases} \tag{2.10}$$

While retaining the viscous step unchanged

$$\begin{cases} \frac{3\tilde{\mathbf{u}}^{n+1} - 4\mathbf{u}^n + \mathbf{u}^{n-1}}{2\Delta t} + \nabla p^n - \nu\nabla^2\tilde{\mathbf{u}}^{n+1} = \mathbf{f}^{n+1} \\ \tilde{\mathbf{u}}^{n+1}|_{\partial\Omega} = 0 \end{cases} \tag{2.11}$$

the second substep was replaced by the difference between 2.10 and 2.11:

$$\frac{3}{2\Delta t}(\mathbf{u}^{n+1} - \tilde{\mathbf{u}}^{n+1}) + \nabla p^{n+1} - \nabla p^n - \nu\nabla^2\mathbf{u}^{n+1} + \nu\nabla^2\tilde{\mathbf{u}}^{n+1} = 0 \tag{2.12}$$

Making use of the following vector identities:

$$\nabla^2\mathbf{v} = \nabla(\nabla \cdot \mathbf{v}) - \nabla \times (\nabla \times \mathbf{v})$$

$$\nabla \cdot (\nabla \times \mathbf{v}) = 0$$

the second sub-step can be replaced with:

$$\begin{cases} \frac{1}{\Delta t}(\mathbf{u}^{n+1} - \tilde{\mathbf{u}}^{n+1}) + \nabla\phi^{n+1} = 0 \\ \nabla \cdot \mathbf{u}^{n+1} = 0, \qquad \mathbf{u}^{n+1} \cdot \mathbf{n}|_{\partial\Omega} = 0 \end{cases} \tag{2.13}$$

pointing $\phi^{n+1} = p^{n+1} - p^n + \nu\nabla\cdot\tilde{\mathbf{u}}^{n+1}$. The second sub-problem can be transformed into the usual Poisson problem, this time in ϕ^{n+1}. To understand why the modified scheme performs better we had to observe from (2.10) that:

$$\begin{aligned}&\left(\nabla p^{n+1} - \mathbf{f}^{n+1} + \nu\nabla\times\nabla\times\mathbf{u}^{n+1}\right)\cdot\mathbf{n}|_{\partial\Omega} = \\ &= \left(\nabla p^{n+1} - \nabla p^n + \nabla\left(\nu\nabla\cdot\tilde{\mathbf{u}}^{n+1}\right)\cdot\mathbf{n}|_{\partial\Omega} = \left.\frac{\partial\phi^{n+1}}{\partial\mathbf{n}}\right|_{\partial\Omega} = 0\right.\end{aligned}$$

is a consistent pressure boundary condition. The splitting error now manifests itself only in the form of an inexact tangential boundary condition on the velocity.
The algorithm generalize easily to a large class of time-marching algorithm. For instance, it is possible to use the q-th order backward difference formula (BFDq) to approximate $\partial_t\mathbf{u}$.
Likewise, we can denote $p^{*,n+1} = \sum_{j=0}^{r}\gamma_j p^{n-j}$ the r-th order extrapolation for p^{n+1}.
In particular:

$$p^{*,n+1} = \begin{cases} 0 & if\ r = 0 \\ p^n & if\ r = 1 \\ 2p^n - p^{n-1} & if\ r = 2 \end{cases} \tag{2.14}$$

2.1.4 Direction-splitting, fractional step method

The different strategies illustrated above require the solution of a Poisson problem equipped with Neumann boundary conditions, which turns out to have a high cost for large size problems and large Reynolds numbers. Quite recently, Guermond and Minev [2][15] proposed a new method to reduce the computational complexity, that departs from the projection paradigm. The new key idea consists of replacing the standard Poisson problem for the pressure correction by a succession of one-dimensional second-order boundary value problems in each spatial direction. Moreover, the direction-splitting technique was also applied to the momentum equation, further reducing the overall computational complexity of the method.
Let's still consider the same Stokes problem. It has been proved that the incremental pressure-correction algorithms are discrete realizations of the following $\mathcal{O}(\epsilon^2)$ perturbation of (2.1):

$$\begin{cases} \frac{\partial\mathbf{u}_\epsilon}{\partial t} + \nabla p_\epsilon - \frac{1}{Re}\nabla^2\mathbf{u}_\epsilon = \mathbf{f}\,, & \mathbf{u}_\epsilon|_{\partial\Omega\times(0,T)} = 0,\ \mathbf{u}_\epsilon|_{t=0} = \mathbf{u}_0, \\ -\Delta t\nabla^2\phi_\epsilon + \nabla\cdot\mathbf{u}_\epsilon = 0\,, & \left.\frac{\partial\phi_\epsilon}{\partial\mathbf{n}}\right|_{\partial\Omega\times(0,T)} = 0, \\ \Delta t\frac{\partial p_\epsilon}{\partial t} = \phi_\epsilon - \frac{\chi}{Re}\nabla\cdot\mathbf{u}_\epsilon\,, & p_\epsilon|_{t=0} = p_0, \end{cases} \tag{2.15}$$

where $\epsilon := \Delta t$ is the perturbation parameter and $\chi \in [0,1]$ is a user-dependent parameter (actually $\chi = 0$ correspond to the standard form of the projection method, while $\chi = 1$ correspond to the rotational form).
The analysis of (2.15) reveals that the same convergence properties for velocity and pressure can be obtained generalizing the Laplacian operator with an operator A, as follows:

$$\begin{cases} \frac{\partial\mathbf{u}_\epsilon}{\partial t} + \nabla p_\epsilon - \frac{1}{Re}\nabla^2\mathbf{u}_\epsilon = \mathbf{f}\,, & \mathbf{u}_\epsilon|_{\partial\Omega\times(0,T)} = 0,\ \mathbf{u}_\epsilon|_{t=0} = \mathbf{u}_0, \\ A\phi_\epsilon = -\frac{1}{\Delta t}\nabla\cdot\mathbf{u}^{n+1}\,, & \phi_\epsilon(t) \in D(A), \\ \Delta t\frac{\partial p_\epsilon}{\partial t} = \phi_\epsilon - \frac{\chi}{Re}\nabla\cdot\mathbf{u}_\epsilon\,, & p_\epsilon|_{t=0} = p_0, \end{cases} \tag{2.16}$$

To ensure convergence, it is necessary that the domain $D(A)$ and the bilinear form induced by A, $a(p,q) \coloneqq \int_\Omega qAp \, \mathrm{d}\mathbf{x}$, satisfy the following assumptions:

$$a \ symmetric\,, \quad and \quad \|\nabla q\|^2_{\mathbf{L}^2} \le a(q,q)\,, \quad \forall q \in D(A) \tag{2.17}$$

Many choices are possible for the operator A (using $A = -\nabla^2$ recovers the previous projection scheme). The new method developed by Guermond and Minev uses $A \coloneqq (1-\partial_{xx})(1-\partial_{yy})(1-\partial_{zz})$. It is interesting to note that operator A breaks the $\mathbf{L}^2$-orthogonality, therefore we are no longer, actually, in the projection paradigm, but $\mathbf{u}_\epsilon$ still converges to the exact solution of (2.15). The advantage of using A is that solving $A\phi = f\,, \ \forall f \in L^2(\Omega)$ becomes easier.

2.2 Algorithm

The program was developed to solve the time dependent, dimensionless incompressible Navier-Stokes equations, on a finite time interval $[0,T]$ and in a cubic domain $\Omega = (0,L_x)\times(0,L_y)\times(0,L_z)$:

$$\begin{cases} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{1}{Re}\nabla^2\mathbf{u} + \nabla p = \mathbf{f}\,, & in\ \Omega\times(0,T], \\ \nabla\cdot\mathbf{u} = 0\,, & in\ \Omega\times(0,T], \\ \mathbf{u}|_{t=0} = \mathbf{u_0}\,, & in\ \Omega \\ \mathbf{u}|_{\partial\Omega} = \mathbf{g}\,, & in\ \partial\Omega\times(0,T] \end{cases} \tag{2.18}$$

where $\mathbf{g}$ contains the boundary data.
Let the Crank–Nicolson time scheme be used for advancing the solution in time, with the leap-frog strategy for the pressure. As a result, the semi-discrete expression of the generalized pressure-correction scheme reads:

$$\begin{cases} \frac{\mathbf{u}^{n+1}-\mathbf{u}^n}{\Delta t} - \frac{1}{2Re}\nabla^2(\mathbf{u}^{n+1}+\mathbf{u}^n) = \mathbf{f}^{n+1/2} - \nabla p^{*,n+1/2} - \mathbf{nl}(\mathbf{u}^{*,n+1/2}) \\ A\phi^{n+1/2} = -\frac{1}{\Delta t}\nabla\cdot\mathbf{u}^{n+1} \\ p^{n+1/2} = p^{n-1/2} + \phi^{n+1/2} - \frac{\chi}{2Re}\nabla\cdot(\mathbf{u}^{n+1}+\mathbf{u}^n) \end{cases} \tag{2.19}$$

the superscripts n and $n+1$ indicate two successive time-steps for velocity, $n+1/2$ the intermediate time-step that is used for pressure and for the right-hand side. **nl** denote the nonlinear term, covered in the next section. Another formulation of the algorithm was also used, using a more efficient time-advancement scheme instead of Crank-Nicolson, characterized by a larger CFL stability margin. Specifically, the RK-RAI3 scheme has been implemented; it is a partially implicit method, where the viscous terms are treated with a second order Crank-Nicolson, while a third order Runge-Kutta scheme is employed for the convective terms [1].
To advance in time (2.19) the following algorithm was used:

1. **Pressure predictor**
   The first step of the algorithm consists in computing the pressure predictor $p^{*,n+1/2}$, as reported in (2.14):

   $$p^{*,n+1/2} = p^{*,n-1/2} + \phi^{n-1/2}\,, \qquad n > 0 \tag{2.20}$$

   The algorithm is initialized by setting $p^{*,1/2} = p^{*,-1/2} = p_0$.

2. **Velocity update**
   In the second step, the velocity field is updated solving the momentum equation. As mentioned above, the proposed algorithm uses a direction splitting approach, following [10], also for the momentum equation. This leads to the solution of multiple one-dimensional linear systems:

$$\begin{cases} \frac{\xi^{n+1}-\mathbf{u}^n}{\Delta t} = \frac{1}{Re}\nabla^2\mathbf{u}^n + \mathbf{f}^{n+1/2} - \nabla p^{*,n+1/2} - \mathbf{nl}(\mathbf{u}^{*,n+1/2}) \\ \frac{\boldsymbol{\eta}^{n+1}-\xi^{n+1}}{\Delta t} - \frac{1}{2Re}\partial_{xx}(\boldsymbol{\eta}^{n+1} - \mathbf{u}^n) = 0 \\ \frac{\boldsymbol{\zeta}^{n+1}-\boldsymbol{\eta}^{n+1}}{\Delta t} - \frac{1}{2Re}\partial_{yy}(\boldsymbol{\zeta}^{n+1} - \mathbf{u}^n) = 0 \\ \frac{\mathbf{u}^{n+1}-\boldsymbol{\zeta}^{n+1}}{\Delta t} - \frac{1}{2Re}\partial_{zz}(\mathbf{u}^{n+1} - \mathbf{u}^n) = 0 \end{cases} \tag{2.21}$$

   This formulation can be manipulated to make the computation of $\mathbf{u}^{n+1}$ easier:

$$\begin{cases} (\boldsymbol{\eta}^{n+1} - \mathbf{u}^n) - \frac{\Delta t}{2Re}\partial_{xx}(\boldsymbol{\eta}^{n+1} - \mathbf{u}^n) = \Delta t\mathbf{f}^{n+1/2} - \\ \qquad\qquad\qquad\qquad \Delta t\nabla p^{*,n+1/2} - \Delta t\mathbf{nl}(\mathbf{u}^{*,n+1/2}) \\ (\boldsymbol{\zeta}^{n+1} - \mathbf{u}^n) - \frac{1}{2Re}\partial_{yy}(\boldsymbol{\zeta}^{n+1} - \mathbf{u}^n) = (\boldsymbol{\eta}^{n+1} - \mathbf{u}^n) \\ (\mathbf{u}^{n+1} - \mathbf{u}^n) - \frac{1}{2Re}\partial_{zz}(\mathbf{u}^{n+1} - \mathbf{u}^n) = (\boldsymbol{\zeta}^{n+1} - \mathbf{u}^n) \end{cases} \tag{2.22}$$

3. **Penalty step**
   Step characterized by the calculation of ϕ^{n+1}, exploiting the definition of splitting operator $A := (1-\partial_{xx})(1-\partial_{yy})(1-\partial_{zz})$. The operator A make easier to compute the equation $A\phi^{n+1/2} = -\frac{1}{\Delta t}\nabla\cdot\mathbf{u}^{n+1}$, because it leads to the solution of three one-dimensional problems: find ψ, φ, and ϕ such that:

$$\begin{cases} \psi - \partial_{xx}\psi = -\frac{1}{\Delta t}\nabla\cdot\mathbf{u}^{n+1}, & \partial_x\psi|_{x=0,1} = 0 \\ \varphi - \partial_{yy}\varphi = \psi, & \partial_y\varphi|_{y=0,1} = 0 \\ \phi - \partial_{zz}\phi = \varphi, & \partial_z\phi|_{z=0,1} = 0 \end{cases} \tag{2.23}$$

4. **Pressure update**
   The last sub-step of the algorithm consists of updating the pressure as follows:

$$p^{n+1/2} = p^{n-1/2} + \phi^{n+1/2} - \chi\nu\nabla\cdot(\frac{1}{2}(\tilde{\mathbf{u}}^{n+1} + \mathbf{u}^n)) \tag{2.24}$$

The present algorithm has been implemented using central finite differences for the first and second order derivatives.

2.2.1 Non-linear Term

Referring to the equation (2.19), the term $\mathbf{nl}$ synthesize $(\mathbf{u}\cdot\nabla)\mathbf{u}$ while $\mathbf{u}^{*,n+1/2} = (3\mathbf{u}^n - \mathbf{u}^{n-1})/2$ is a second-order extrapolation of velocity. Overall, the term we need to discretize is:

$$\frac{3}{2}(\mathbf{u}^n\cdot\nabla)\mathbf{u}^n - \frac{1}{2}(\mathbf{u}^{n-1}\cdot\nabla)\mathbf{u}^{n-1}$$

therefore it is necessary to keep the non-linear term calculated at the previous time-step. The formulation of the non-linear term has been chosen to ensure energy conservation.

Not respecting energy conservation can lead to explosive instabilities, especially at high Re. It is possible to describe different forms of the non-linear term that coincide in the continuum but are different in the discrete. These formulations are essentially different due to a term multiplied by the divergence of velocity. In our case velocity is only an approximation of the divergence-free speed (2.16), therefore the different forms of the non-linear term are no longer equivalent and in general do not respect the conservation of energy. Following what is reported in [16] the conservative formulation of the non-linear term turns out to be:

$$
\begin{aligned}
&(\mathbf{u}\cdot\nabla)u_l|_{x_i,y_j,z_k} \approx \\
&\frac{1}{4\Delta x}[u_l(x_{i+1},y_j,z_k)(u(x_{i+1},y_j,z_k)+u(x_i,y_j,z_k))-u_l(x_{i-1},y_j,z_k)(u(x_{i-1},y_j,z_k)+u(x_i,y_j,z_k))]+ \\
&\frac{1}{4\Delta y}[u_l(x_i,y_{j+1},z_k)(v(x_i,y_{j+1},z_k)+v(x_i,y_j,z_k))-u_l(x_i,y_{j-1},z_k)(v(x_i,y_{j-1},z_k)+v(x_i,y_j,z_k))]+ \\
&\frac{1}{4\Delta z}[u_l(x_i,y_j,z_{k+1})(w(x_i,y_j,z_{k+1})+w(x_i,y_j,z_k))-u_l(x_i,y_j,z_{k-1})(w(x_i,y_j,z_{k-1})+w(x_i,y_j,z_k))]
\end{aligned}
\tag{2.25}
$$

2.3 Schur Complement Method

The Schur Complement (or Dual Schur Decomposition) is a direct parallel method, belonging to the category of Non-overlapping Subdomain methods. It is particularly suitable for parallel computing because each processor has actually to solve twice its subdomain plus an interface problem, keeping global communication to a minimum. A general approach has been maintained in this paragraph; actual software implementation will be treated in the next chapter.
The linear equation system to be solved is denoted as: $\mathbf{Au} = \mathbf{f}$.
After a proper domain decomposition the unknowns in vector $\mathbf{u}$ are partitioned into subsets, one subset per processor corresponding to the inner domain, plus one containing all the *interface* unknowns, labeled s:

$$\mathbf{u} = [\mathbf{u}_0, \mathbf{u}_1, ..., \mathbf{u}_{P-1}, \mathbf{u}_s]^t$$

Treating the interface separately removes any coupling between variables belonging to different subsets:

$$\text{if } i \in \mathbf{u}_{p_1} \text{ and } j \in \mathbf{u}_{p_0} \text{ then } \mathbf{A}_{i,j} = 0$$

Let $\hat{N}_p$ be the number of *inner* unknowns of subset p and N_s the number of unknowns of the interface s, while N_p refers to the total number of unknowns. According to the new ordination, the system can be expressed in terms of block matrices as:

$$
\begin{bmatrix}
\mathbf{A}_{0,0} & 0 & ... & & \mathbf{A}_{0,s} \\
0 & \mathbf{A}_{1,1} & ... & & \mathbf{A}_{1,s} \\
\vdots & & & & \vdots \\
0 & ... & & \mathbf{A}_{P-1,P-1} & \mathbf{A}_{0,s} \\
\mathbf{A}_{s,0} & \mathbf{A}_{s,1} & ... & & \mathbf{A}_{s,s}
\end{bmatrix}
\begin{bmatrix}
\mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{P-1} \\ \mathbf{u}_s
\end{bmatrix}
=
\begin{bmatrix}
\mathbf{f}_0 \\ \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_{P-1} \\ \mathbf{f}_s
\end{bmatrix}
\tag{2.26}
$$

1. $\mathbf{A}_{0,0}, ..., \mathbf{A}_{P-1,P-1}$ are sparse $\hat{N}_p \times \hat{N}_p$ matrices (for example tridiagonal matrices) expressing inner unknowns coupling;

2. $\mathbf{A}_{0,s}, ..., \mathbf{A}_{P-1,s}$ are sparse $\hat{N}_p \times N_s$ matrices expressing the coupling of inner unknowns of processor p with the interface unknowns;
3. $\mathbf{A}_{s,0}, ..., \mathbf{A}_{s,P-1}$ are sparse $N_s \times \hat{N}_p$ matrices expressing the coupling of interface unknowns with inner unknowns of processor;
4. $\mathbf{A}_{s,s}$ is an $N_s \times N_s$ matrix expressing interface unknowns coupling;

Block Gaussian elimination is used to transform the last block equation of system (2.26) from

$$[\mathbf{A}_{s,0}, \mathbf{A}_{s,1}, ..., \mathbf{A}_{s,s}] \cdot [\mathbf{u}_0, \mathbf{u}_1, ..., \mathbf{u}_{P-1}, \mathbf{u}_s]^t = \mathbf{f}_s$$

to

$$[\mathbf{0}, \mathbf{0}, ..., \mathbf{0}, \tilde{\mathbf{A}}_{s,s}] \cdot [\mathbf{u}_0, \mathbf{u}_1, ..., \mathbf{u}_{P-1}, \mathbf{u}_s]^t = \tilde{\mathbf{f}}_s$$

to obtain:

$$\tilde{\mathbf{A}}_{s,s}\mathbf{u}_s = \tilde{\mathbf{f}}_s \tag{2.27}$$

where

$$\begin{cases} \tilde{\mathbf{A}}_{s,s} = \mathbf{A}_{s,s} - \sum_{p=0}^{P-1} \mathbf{A}_{s,p}\mathbf{A}_{p,p}^{-1}\mathbf{A}_{p,s} \\ \tilde{\mathbf{f}}_s = \mathbf{f}_s - \sum_{p=0}^{P-1} \mathbf{A}_{s,p}\mathbf{A}_{p,p}^{-1}\mathbf{f}_p \end{cases} \tag{2.28}$$

with $\tilde{\mathbf{A}}_{s,s}$ known as *Schur Complement* of the block matrix (2.26).
Once $\mathbf{u}_s$ is computed from (2.27), each of the $\mathbf{u}_p$ could be determined solving:

$$\mathbf{A}_{p,p}\mathbf{u}_p + \mathbf{A}_{p,s}\mathbf{u}_s = \mathbf{f}_s \tag{2.29}$$

equation obtained for the generic processor p from (2.26). The first problem in solving (2.27) concerns the term $\mathbf{A}_{p,p}^{-1}$, required by both $\tilde{\mathbf{f}}_s$ and $\tilde{\mathbf{A}}_{s,s}$. It is not possible to obtain the inverse explicitly and efficiently except for very small problems. Furthermore, each processor p has only the matrices related to its subdomain; communication at this point is required.
Let $\tilde{\mathbf{A}}_{s,s}^p = \mathbf{A}_{s,p}\mathbf{A}_{p,p}^{-1}\mathbf{A}_{p,s}$ the contribution from processor p to $\tilde{\mathbf{A}}_{s,s}$. This term can be evaluated by any processor without explicitly compute the inverse, through column by column approach. For column c from 1 to N_s, solve the auxiliary equations:

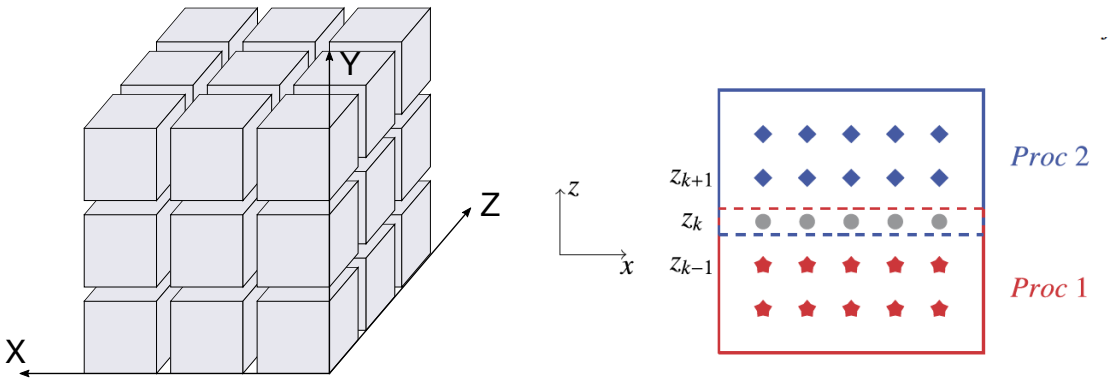
$$\mathbf{A}_{p,p}\,\mathbf{t} = [\mathbf{A}_{p,s}]_c \quad ; \qquad [\mathbf{A}_{s,s}^p]_c = \mathbf{A}_{s,p}\mathbf{t} \tag{2.30}$$

Note that a lot of effort can be saved as many of the $[\mathbf{A}_{p,s}]_c$ are null.
When all the contributions have been calculated, communication is required so that each processor can determine $\tilde{\mathbf{A}}_{s,s}$. Since the procedure depends only on A, is necessary to perform it only once at the beginning of the algorithm, like an LU decomposition procedure.
The same approach is used for $\tilde{\mathbf{f}}_s$. In this case only one system needs to be solved; there are no more N_s systems (as many as the columns of $\mathbf{A}_{s,p}$). Given $\tilde{\mathbf{f}}_s^p = \mathbf{A}_{s,p}\mathbf{A}_{p,p}^{-1}\mathbf{f}_p$ each processor evaluates its contribution as:

$$\mathbf{A}_{p,p}\,\mathbf{t} = \mathbf{f}_p \quad ; \qquad \tilde{\mathbf{f}}_s^p = \mathbf{A}_{s,p}\mathbf{t} \tag{2.31}$$

2.3.1 Parallel Implementation

The parallel implementation is based on a Cartesian block decomposition of the domain Fig.(2.1a), and use the Message Passing Interface (MPI) library for communication between blocks. The procedure is divided into 2 stages: *Preprocessing*, related to



(a) (b)

Figure 2.1. (a) Domain partitioning for parallel implementation (b) 2 blocks decomposition in z direction ($x-z$ plane view); blue diamonds and red star denote grid points; grey circle denote shared interface grid points.

the assembly of the Schur Complement, and *Solution*, related to the evaluation of $\mathbf{u}$. Once spatial discretization is applied, each one-dimensional linear problem reduces to a tridiagonal linear system. Since matrices involved in the calculation of the Schur complement don't vary in time, they can be factored once at the beginning of the algorithm and used to compute $\tilde{\mathbf{A}}_{s,s} = \mathbf{A}_{s,s} - \sum_{p=0}^{P-1} \mathbf{A}_{s,p}\mathbf{A}_{p,p}^{-1}\mathbf{A}_{p,s}$ on each processor (*Preprocessing* stage). The contribution from processor p to $\tilde{\mathbf{A}}_{s,s}$ is computed locally through (2.30) and then distributed with MPI.
During the *Solution* stage, at each iteration we had to compute the contribution to the right-hand sides of $\tilde{\mathbf{A}}_{s,s}\mathbf{u}_s = \tilde{\mathbf{f}}_s$. As pointed out in (2.31), this step requires the solution of a linear system equivalent to that for the internal unknowns. Then all the contributions are assembled via *MPI_Allreduce*. At this point each processor can solve the linear system (2.27) for the interface unknowns, feasible approach due to the fact that $N_s \ll \hat{N}_p$, and then (2.29) for the internal unknowns. In our case, the tridiagonal systems (2.27) and (2.29) are solved by Thomas' algorithm which is equivalent to a banded LU decomposition without pivoting when the coefficients are saved in the forward step.
Overall, the algorithm requires the internal linear system to be solved twice, first to calculate the right hand side contribution and second to solve for $\mathbf{u}_p$. This strategy allows us to minimize the communication between processes within the time advancement, however introducing an overwork for each processor that it is useful to highlight.
For simplicity, we consider a domain decomposed into 2 blocks in the z direction, as indicated in Fig.(2.1b). For each time step the following linear systems, both for the

momentum and for the pressure step, have to be solved:

$$\begin{cases} A_x\psi = f\,, & b.c. \\ A_y\varphi = \psi\,, & b.c. \\ A_z\phi = \varphi\,, & b.c. \end{cases} \tag{2.32}$$

The right-hand side f cannot be fully computed at the interface in either of the two processes. The stencil needed for f requires data located between the 2 processors. As long as we want to keep communication to a minimum, i.e. only when assembling the forcing contribution of the Schur Complement equation, we exploit the linearity of the equation and split the right-hand side at the interface points between the 2 processors. In detail, it is written:

$$f_i = f_{i,1} + f_{i,2} \tag{2.33}$$

where $f_{i,1}$ and $f_{i,2}$ denote the contribution to f_i evaluated with the portion of the stencil available on the two processors, respectively. Only thing to be careful is not to consider the contribution of the shared interface 2 times. Therefore, at the interface points the 2 processors compute:

$$\begin{cases} A_x\psi_{i,1} = f_{i,1}\,, & b.c.|_1\,; & A_x\psi_{i,2} = f_{i,2}\,, & b.c.|_2\,; \\ A_y\varphi_{i,1} = \psi_{i,1}\,, & b.c.|_1\,; & A_y\varphi_{i,2} = \psi_{i,2}\,, & b.c.|_2\,; \end{cases} \tag{2.34}$$

Finally, when the z direction is considered, the Schur complement is assembled from different processors and then used. Therefore, the shared interface unknown results in:

$$\phi_i = A_z^{-1}(\varphi_{i,1} + \varphi_{i,2}) = A_z^{-1}A_y^{-1}A_x^{-1}(f_{i,1} + f_{i,2}) = A_z^{-1}A_y^{-1}A_x^{-1}f_i \tag{2.35}$$

Overall the tridiagonal systems are solved twice at the interface points and the communication is limited to the assembly of the Schur's equation right-hand side.

Chapter 3

Software Code

The development of the code was done using *CUDA Fortran*, a small set of extensions to Fortran programming language that allows us to take advantage of the NVIDIA CUDA computing architecture. It includes a Fortran 2003 compiler (*nvfortran*) and toolchain for programming NVIDIA GPUs using Fortran. It is contained in the NVIDIA HPC Software Development Kit (SDK), a comprehensive suite of tools for the development of HPC applications. HPC-SDK includes compilers with GPU acceleration support in C, C ++ and Fortran languages (*nvcc*, *nvfortran*, ...), performance profiling and debugging tools (*Nsight Systems*, *Nsight Compute*, *cuda-gdb*, *nvrpof*, ...), a suite of GPU-accelerated math libraries (*cuBLAS*, *cuSOLVER*, *cuFFT*, ...) and highly optimized multi-GPU and multi-node communication libraries (*NCCL*, ...).

The structure of a program written in CUDA Fortran reflects the coexistence of a host (CPU) and one or more devices (GPUs). In general any program which uses GPUs for computation performs the following key steps [17]:

1. Initialize and select the GPU to run on;
2. Allocate memory spaces on the GPU and move data from host to device;
3. Invoke kernels to execute computations on data stored on GPU memory;
4. Gather data back from GPU and deallocate memory spaces (generally implicitly performed when host program exits);

3.1 Program Overview

The work was done, initially, following the Open Source solver AFiD-GPU as a reference [18]. In particular, these are the key concepts at the root of the code:

- It has been tried to stay as faithful as possible to the CPU version from which we started. To achieve this, the GPU implementation made extensive use of the preprocessor and all GPU specific directives were placed within the USE_CUDA macro. So with the same .f90 source file it is possible to create a CPU object file but also a GPU object file by adding the flags -cuda -Mpreprocess -DUSE_CUDA;

- All the data needed for the computation are in *double precision* and resides in the GPU, i.e. each GPU has the data in global memory to carry out the entire time-loop. There is no explicitly data transfer between GPU and CPU within the time loop. Obviously, data transfer between CPU and GPU is possible when communication between the various nodes of the cluster has to be done, even if the NVIDIA NCCL library for inter-GPU communication was used;
- The possibility to choose between different boundary conditions was preserved, despite having considerably complicated some parts of the code. This certainly weighed on performance but allows us to obtain a more usable code;
- Regarding the multi-GPU implementation, simply one device was assigned to each MPI process. An attempt was made to take advantage of the topology-aware NCCL library to develop a scalable application;

3.1.1 Program Organization

The files of the entire solver are grouped into the following categories:

- Main file
  main.f90: contains the declaration and initialization of the host and device physical variables of the problem. Contains the time-advancing scheme and the post processing activities.
- Definition and initialization of data structures
  types.f90: module containing declaration of parameters and data structures shared between various program units; **init.f90**: module initializing data structures containing information about domain decomposition and solvers, starting from input files;
- Computation of the RHS
  buildRHS.f90: module containing the CPU subroutines relating to the discretization of the various right hand side terms present in (2.22) and their assembly routines (for both CPU and GPU);
  RHSkernels.f90:module containing kernels relating to the computation of the right hand side terms;
- Solvers
  solver.f90: module containing tridiagonal system solvers for different boundary conditions (CPU version). It also contains routines that preform the direction-splitting procedure, for both CPU and GPU;
  SchurSolver.f90: module containing the CPU subroutines responsible for the execution of Schur Complement method;
  SolverKernels.f90 and **SchurKernels.f90**: these are two modules containing kernels relating respectively to tridiagonal solvers and Schur Complement algorithms;
- Assign device
  mpiDeviceUtil.f90: module used to ensure that each MPI process is mapped to a unique device;

3.1.2 CUF kernel

CUDA Fortran allows automatic kernel generation and invocation from a region of host code containing one or more tightly nested loops. Such kernel code is referred to as a *CUF kernel*, or kernel loop directives. The general form of the kernel directive is:

!$ cuf kernel do [(n)] <<< grid, block, [optional stream] >>>

where grid, block and stream have the same meaning reported in (1.3.1). The compiler maps the launch configuration specified onto the outermost n loops, starting from loop n and working backwards. Using <<< ∗, ∗ >>> leaves to the compiler the choice on how to launch the grid and the block. The launch of CUF kernels is asynchronous or non-blocking as well as that of kernels, i.e. once they are launched the control immediately returns to the host. The CUDA function *cudaDeviceSynchronize()* can be used to block the host until the previously issued operations on device have been completed.
Extensive use has been made of CUF kernel directives to parallelize simple operations, such as setting boundary conditions for velocity and pressure, and to execute expressions. The computationally intensive parts, such as the calculation of the RHS or the solution of tridiagonal systems, have instead been coded manually.
One of the area where CUF kernels are very beneficial is in performing reductions. Writing efficient reductions in CUDA is not an easy task. The use of CUF kernels allows us to do this automatically as we can see from the following portion of the CFL calculation code (from main.f90):

```
SUBROUTINE ComputeCFLmax_GPU(N,x1,x2,x3,u,cfl)

  IMPLICIT NONE
  integer, dimension(:), device, intent(in) :: N
  REAL(KIND=8), DIMENSION(0:,0:,0:,:), device, INTENT(IN) :: u
  REAL(KIND=8), DIMENSION(-overlap:), device, INTENT(IN) :: x1,x2,x3

  REAL(KIND=8)                                        :: cfl, qcf

  REAL(KIND=8) :: dx,dy,dz
  INTEGER      :: i,j,k,ierr

  cfl=0.0

    !$cuf kernel do(3) <<<*,*>>>
    DO k = 0, N(3) + 1
      DO j = 0, N(2) + 1
        DO i = 0, N(1) + 1
          dz=x3(k+1)-x3(k)
          dy=x2(j+1)-x2(j)
          dx=x1(i+1)-x1(i)

          qcf = ABS(u(i,j,k,1))/dx + &
                    ABS(u(i,j,k,2))/dy + &
                    ABS(u(i,j,k,3))/dz

          cfl=MAX(cfl, qcf)

        END DO
      END DO
    END DO

  CALL MPI_ALLREDUCE(MPI_IN_PLACE,cfl,1,MPI_REAL8,MPI_MAX,MPI_COMM_WORLD,ierr)
  cfl=cfl*grids%DeltaT

```

```
36 END SUBROUTINE ComputeCFLmax_GPU
```

Listing 3.1. CFL reduction computation performed exploiting CUF kernel directive

Once the compiler realizes that the 3 nested do loops have to be parallelized, it automatically determines that *cflm* requires a reduction. The value of *cflm* is calculated by each thread and must be compared with that of all the others to determine the maximum in the entire domain. The operation is performed behind the scenes by the compiler resulting in a much lower implementation effort.

3.1.3 Data Transposition

The data transposition kernel is widely used to improve the performance of various algorithms within the time loop. It is a simple example of memory-bound operation therefore it is suitable for presenting some basic memories utilization that are generally done when programming for CUDA architectures.
First thing to consider is how thread indices are mapped to array elements for this kernel. An abstract representation of the array elements in space was used, as a reference, as shown in Fig.(3.1). The coordinate system (x, y, z), for which origin is in the lower right corner, maps to the x, y and z components of our predefined variables threadIdx, blockIdx and blockDim. Therefore blockIdx%z will be used to change face along z, while each block of threads will transpose a $2D$ data tile on the face.

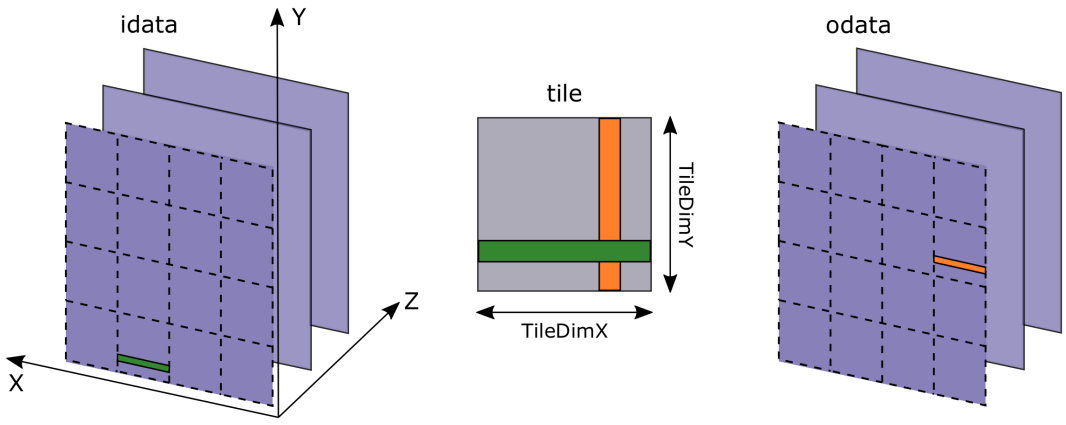


Figure 3.1. Matrix Transposition. A shared memory tile (TileDimX,TileDimY) is used to achieve full coalescing of global memory reads and writes. A warp of threads reads contiguous data from a portion of *idata* and loads it into a shared memory row. The same warp reads a column from shared memory *tile* and writes it to a partial row of *odata*

The threads are launched in blocks of $(32 \times 8 \times 1)$, while the tile dimensions are set to (32×32), through parameters $TileDimX$ and $TileDimY$. Using a thread block with fewer threads than elements in a tile means that each thread are going to transpose several matrix elements, four in our case.
The kernel code corresponding to Fig.(3.1) is:

```
1  attributes(global) subroutine transposeKernel(N1,N2,idata,odata)
2          implicit none
3          integer, value :: N1,N2
4          real(kind=8), dimension(0:N1+1,0:N2+1,0:Nz_c+1), device, intent(in) :: idata
5          real(kind=8), dimension(0:N2+1,0:N1+1,0:Nz_c+1), device, intent(out) ::
       odata
6          real(kind=8), shared, dimension(tileDimX+1,tileDimY) :: tile
7
8          integer :: i,j,k, is,js, str
9
10         i = (threadIdx%x + (blockIdx%x-1) * tileDimX)-1
11         j = (threadIdx%y + (blockIdx%y-1) * tileDimY)-1
12         k = blockIdx%z-1
13         is = threadIdx%x
14         js = threadIdx%y
15
16         do str = 0,tileDimY-1,blockDim%y
17             if(i<=N1+1 .and. j+str<=N2+1) then
18                 tile(is,js + str) = idata(i,j + str,k)
19             end if
20         end do
21
22         call syncthreads()
23
24         i = (threadIdx%x + (blockIdx%y-1) * tileDimY)-1
25         j = (threadIdx%y + (blockIdx%x-1) * tileDimX)-1
26
27         do str = 0,tileDimY-1,blockDim%y
28             if(i<=N2+1 .and. j+str<=N1+1) then
29                 odata(i ,j + str,k) = tile(js+str,is)
30             end if
31         end do
32
33     end subroutine transposeKernel3D
```

Listing 3.2. Matrix Transposition kernel

One of the most important factor of CUDA kernel performance is the accessing pattern in global memory by threads. As pointed out in Chapter 1, threads within a block are grouped into warps of 32 threads. In the architecture developed by NVIDIA, Single Instruction Multiple Threads (SIMT), each instruction on the device is issued to a warp of threads, and execution of instructions is performed by each thread in a warp in lockstep. The grouping of threads in warps is not only relevant for computation but also for accesses in global memory. When all threads in a warp execute a load/store instruction, the hardware detects whether they access consecutive global memory locations. In this case, the hardware combines, or coalesces, all these accesses into a consolidated access to consecutive DRAM locations. Such coalesced access allows the DRAMs to deliver data as a burst. Since in Fortran the first index in multidimensional variable varies the quickest, contiguous elements in memory are along the x direction, as shown in Fig.(3.1). On line 18 a warp reads contiguous data from the input variable *idata* and loads it into rows of the shared memory tile. Then, after recalculating the array indices a column on the shared memory tile is written to contiguous addresses in *odata*. Shared memory is used because there is no access pattern restrictions, therefore reading a "column" of data does not affect performance. The only performance issue with shared memory is *bank conlflicts*. Shared memory is divided into equally sized memory modules (banks) to achieve higher bandwidth. Each memory load/store spanning n distinct memory banks can be serviced simultaneously. The Volta GPU has 32 banks, each 4 bytes wide. Bank conflicts arise when multiple threads within a warp access different 4-bytes in the same bank. The hardware splits this requests into

as many separate serialized conflict-free requests, decreasing the effective bandwidth. The only exception is when threads need to access the same 4-bytes word, in that case the access is treated like a broadcast. For a shared memory tile of (32×32), all elements in a data column belong to the same memory bank, resulting in the worst case scenario: 32 serialized accesses. To avoid this problem, in line 6, the shared memory variable is declared as a tile of size $(32 + 1 \times 32)$, this technique is known as *padding*. It simply adds a dummy column to the tile so that threads can access different banks.
Syncthreads() is a block-local barrier necessary to avoid race conditions. It is used to block execution until loading is completed by all threads. In this way, we avoid the illegal use of data that has not yet been loaded into shared memory (note that each thread also uses data loaded by others). Therefore, after recalculating the array indices on line 24 and 25, a warp reads a column on the shared memory tile and loads to contiguous addresses in *odata*.

3.1.4 Assign Device

There are various ways to use MPI in conjunction with CUDA Fortran in terms of how the devices are mapped to MPI ranks. In this work a simple approach was used in which each MPI rank is associated with a single GPU. Using this configuration we can still exploit multiple GPUs per node, simply using multiple MPI ranks per node through the --ntasks-per-node instruction. To ensure that each MPI rank has a single device, the *mpiDeviceUtil* module, obtained from [3]; in particular the *AssignDevice* subroutine:

```
1  subroutine assignDevice(dev)
2      use mpi
3      use cudafor
4      implicit none
5      integer :: dev
6      character (len=MPI_MAX_PROCESSOR_NAME), allocatable :: hosts(:)
7      character (len=MPI_MAX_PROCESSOR_NAME) :: hostname
8      integer :: namelength, color, i, j, ierr
9      integer :: nProcs, myrank, newComm, newRank
10
11     call MPI_COMM_SIZE(MPI_COMM_WORLD, nProcs, ierr)
12     call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
13
14     ! allocate array of hostnames
15     allocate(hosts(0:nProcs-1))
16
17     ! Every process collects the hostname of all the nodes (compreso MPI_BCAST)
18     call MPI_GET_PROCESSOR_NAME(hostname, namelength, ierr)
19     hosts(myrank)=hostname(1:namelength)
20
21     do i=0,nProcs-1
22        call MPI_BCAST(hosts(i),MPI_MAX_PROCESSOR_NAME,MPI_CHARACTER,i, &
23             MPI_COMM_WORLD,ierr)
24     end do
25
26     ! sort the list of names
27     call quicksort(hosts,nProcs,MPI_MAX_PROCESSOR_NAME,strcmp)
28
29     ! assign the same color to the same node
30     color=0
31     do i=0,nProcs-1
32        if (i > 0) then
33           if ( lne(hosts(i-1),hosts(i)) ) color=color+1
34        end if
```

```
35          if ( leq(hostname,hosts(i)) ) exit
36        end do
37
38        call MPI_COMM_SPLIT(MPI_COMM_WORLD,color,0,newComm,ierr)
39        call MPI_COMM_RANK(newComm, newRank, ierr)
40
41        dev = newRank
42        ierr = cudaSetDevice(dev)
43        deallocate(hosts)
44
45      end subroutine assignDevice
```

Listing 3.3. AssignDevice subroutine

Several MPI routines are exploited with the aim of compiling a list of host-names used by each ranks to select a unique device. Specifically, MPI_GET_PROCESSOR_NAME routine allow us to get the name of the host in which the function was called. Each node has a name, therefore processes that are started on the same node own the same name. Once MPI_BCAST is concluded each rank will have a complete list of all hostnames. Through the C function *quicksort()*, called using the comparator *strcmp*, all the names in the list are sorted. The iso_c_binding module enables Fortran programs containing properly written interfaces to call directly into the C library functions. This module provides a standard way for dealing with inter-language data types, named constants and procedures. In some cases, NVIDIA has written small wrappers around the existing C library function, to make the Fortran call more user-friendly. Finally a color is associated to each node and through it a new communicator with MPI_COMM_SPLIT. Each new communicator will contain only MPI ranks of the associated node and with MPI_COMM_RANK we can get the new rank. The new rank will be used as an argument to *cudaSetDevice()* to select the device to associate with this host process.

3.2 Kernels

3.2.1 Tridiagonal Solvers

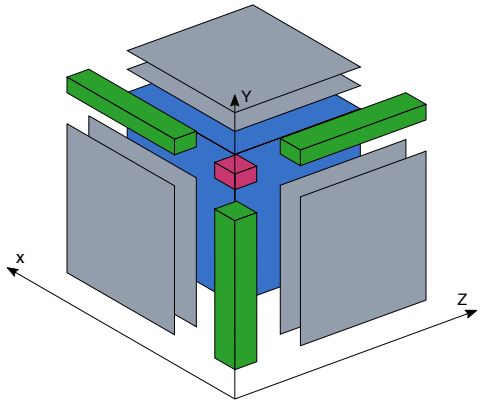
The developing of the Tridiagonal Solvers was the starting point of the following thesis work. The grid of threads is mapped on a face of the domain block. Each face of threads has the task of solving the systems resulting from the banded LU decomposition without pivoting forward and backward. While for the y and z directions, a warp of threads accesses data in global memory that is contiguous, in the x direction a warp accesses data that is strided in memory. When solving in the x direction, a warp of threads, respect to our reference system, is arranged along the y axis and therefore reads/writes data in discontinuous memory locations. This results in a loss of performance due to the fact that accesses are no longer performed in a coalesced way. Therefore, before running the kernel we perform a transposition of the velocity vector using kernel of subsection (3.1.3).

3.2.2 Laplacian Kernel

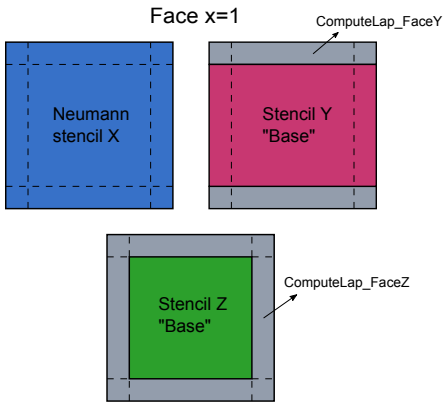
Most of the implementations tested for the Laplacian have been useful in the development of the other kernels and for this reason they will be analyzed deeply in this

subsection.

As shown in Fig.(3.2a), the domain was ideally divided into various regions charac-



(a) Rear faces removed for clarity



(b)

Figure 3.2. (a) Abstract subdivision of the domain into *"internal"* points, *"external"* faces and intersection boundary points. (b) Laplacian computation at $x = 1$. *Blu* square represent points that compute X derivative with Neumann condition; *Violet*: "base" Y derivative; *Green*: "base" Z derivative; *Grey*: stencil computed later in *ComputeLap_FaceY* or *ComputeLap_FaceZ*.

terized in general by a different computational stencil. The points that belong to the domain marked with color blue can be defined as *"internal"*; they are all characterized by the same stencil (*"base"*) which derives from discretization of the second derivative in the 3 spatial directions according to centered finite differences. In the *"external"* points, which are located in one or more faces at the boundary (up to 3 for the edges), the stencil is suitably modified to take into account the different BCs.
Keeping in mind the subdivision of the points the computation has been divided into the following kernels:

```
1  attributes(global) subroutine ComputeLap_Base(D2x, D2y, D2z, var, u, Lap)
2      implicit none
3      real(kind=8), dimension(0:Nx_c+1,dim), device, intent(in) :: D2x
4      real(kind=8), dimension(0:Ny_c+1,dim), device, intent(in) :: D2y
5      real(kind=8), dimension(0:Nz_c+1,dim), device, intent(in) :: D2z
6      integer, value :: var
7      real(Kind=8), dimension(0:Nx_c+1,0:Ny_c+1,0:Nz_c+1,dim), device, intent(in) :: u
8      real(kind=8), dimension(0:Nx_c+1,0:Ny_c+1,0:Nz_c+1,dim), device, intent(inout)
       :: Lap
9      real(kind=8), shared :: tile(TileDimX, 0:TileDimY+1, -1:1)
10     ...
11
12 attributes(global) subroutine ComputeLap_FaceXT(D2x, D2ox, D2y, D2z, var, u, Lap)
13         ...
14
15 attributes(global) subroutine ComputeLap_FaceY(D2x, D2y, D2oy, D2z, var, u, Lap)
16         ...
17
18 attributes(global) subroutine ComputeLap_FaceZ(D2x, D2y, D2z, D2oz, var, u, Lap)
19         ...
```

Listing 3.4. Kernels Laplacian computation

ComputeLap_Base compute the Laplacian at the internal points. The first 3 arguments $D2x$, $D2y$, $D2z$ are the arrays containing the coefficients that discretize the second

derivative respectively in the 3 directions. *var* indicates the component of the velocity vector u, while *Lap* is the variable in which the computation result for each point will be stored. The variable declaration was reported to highlight that was preferred an explicit declaration respect to the assumed shape declaration of the arrays. Using the assumed shape involves a *cudaMemCopy* of the array descriptor from host to device which causes an overhead in the kernel launch and increases register pressure.
An order-k in space stencil refers to a stencil that requires k input elements in each dimension, not counting the element at the intersection. Alternatively, one could refer to the $3D$ order-k stencil as a $(3k+1)$-point stencil. For this kernel each GPU thread is mapped to a point (i, j, k) in the computational domain and is assigned to the computation of a $3D$ order-2 stencil:

$$\begin{aligned} Lap_{i,j,k} \approx D_{i-1}\, u_{i-1,j,k} + D_i\, u_{i,j,k} + D_{i+1}\, u_{i+1,j,k} + \\ D_{j-1}\, u_{i,j-1,k} + D_j\, u_{i,j,k} + D_{j+1}\, u_{i,j+1,k} + \\ D_{k-1}\, u_{i,j,k-1} + D_k\, u_{i,j,k} + D_{k+1}\, u_{i,j,k+1} \end{aligned} \tag{3.1}$$

Although there is the possibility for each warp to access contiguous data in memory, each data is accessed multiple times by different threads. Reading directly from global memory results in an algorithm penalized by global memory bandwidth due to data access redundancy [19]. Specifically, *redundancy* is the ratio between the number of elements accessed and the number of elements processed. A naive approach where only global memory is used requires a refetch of $(3k+1)$-input elements to compute each output value, leading in our case to a read redundancy of 7. Since shared memory latency is lower than that of global memory it is convenient to perform a single reading from global memory at the beginning of the kernel and store data in shared memory. The size of shared memory tile is related to the block configuration execution parameter. We need also to add overlap elements to be able to calculate the stencil completely at the tile boundary. The overlap elements are loaded 2 times, once by the block that calculates the Laplacian at those points, and once by the neighbouring block. The choice of block execution parameters is a trade-off between the size of the shared memory tile, which affects occupancy, and the number of overlap elements, which affects read redundancy.
After some experimentation the best execution configuration was found in $(32, 4, 1)$ for the following shared memory utilization:

```
1  if(i<Nx_c .and. j<Ny_c) then
2
3      tile(is,js,0) = u(i,j,k,var)
4      tile(is,js,-1) = u(i,j,k-1,var)
5      tile(is,js,1) = u(i,j,k+1,var)
6
7      if(js==1) tile(is,js-1,0) = u(i,j-1,k,var)
8
9      if(js==blockDim%y .or. j==Ny_c-1) tile(is,js+1,0) = u(i,j+1,k,var)
10
11     ...
12
13 end if
```

Listing 3.5. *ComputeLap_Base*: shared memory load

First we check that the thread corresponds to an internal point, threads that are outside the indicated portion are inactive. Then each thread loads the corresponding

point data into shared memory. Points that are on the tile boundary, on the other hand, must also load the data immediately outside because it is necessary for the stencil. This procedure is carried out in y and z direction, while in x is convenient to exploit the coalesced global memory accesses. Complicating control flow to load overlap elements in the x direction was not beneficial for such a small stencil.
This strategy is compared in Table (3.1) with a kernel that completely loads data into shared memory. The results were measured with the NVIDIA Nsight-Compute tool.

	Compute_Lap_Base	Complete input array + halos in shared memory
Exec. Time	374.75 μs	476.58 μs
L1/Tex Throughput	11724.654 Gb/s	7828.61 Gb/s
Data requested L1	1,468 Gb	0.996 Gb

Table 3.1. Data collected with Nsight-Compute for different Laplacian strategy tested on V100

We can see that implementation that takes more advantage of the L1 cache achieves better results. The reason is that Volta architecture has significantly narrowed the differences in performance between shared memory utilization and implicit cache utilization for small stencils, i.e. where data reuse is low. It is even slightly inconvenient to use shared memory in these situations, not just for execution time but also for implementation effort. The mixed configuration chosen, however, performed better than the simple use of implicit cache, and for this reason it was preferred.

ComputeLap_FaceXT, *ComputeLap_FaceY*, *ComputeLap_FaceZ* are the kernels responsible for computing Laplacian in the external points located at the boundaries. To simplify the explanation, let's take the x direction as a reference and the first velocity component u; the same approach has to be extended to other directions. Threads are ideally mapped on a 2D boundary face; in particular each thread has to compute output values for different faces:

- $x = 0$, is characterized by a reduced one-dimensional stencil, with coefficients different from zero only for periodic boundary conditions:

$$\frac{\partial^2 u}{\partial x^2}|_{x=0} \approx D_0\, u_{0,j,k} + D_1\, u_{1,j,k}$$

- $x = 1$, is characterized by an extended stencil, in x direction, that takes into account the Neumann boundary conditions possibility:

$$\frac{\partial^2 u}{\partial x^2}|_{x=1} \approx D_0\, u_{0,j,k} + D_1\, u_{1,j,k} + D_2\, u_{2,j,k} + Do_x\, u_{3,j,k}$$

  Points that are at the intersections with other faces on the boundary in direction y and z need to be treated carefully. Following Fig.(3.2b) and Listing(3.6), accurately extended for y and z directions, allows to maintain the same order of operations performed on the CPU version. In this way the results of the two algorithms, if compared with each other, will be identical. Otherwise, due to the

floating-point arithmetic rounding errors, we would have obtained still correct but slightly different results.

- at faces $x = N_x$ and $x = N_x + 1$ we follow, respectively, the strategies in $x = 1$ and $x = 0$, with the addition of a complete 3D stencil in $x = N_x + 1$.

Mapping threads on domain faces results in strided global memory load/write respect to the x direction, like reported in Section (3.2.1). Therefore the transposition of the velocity vector is needed. Moreover it was utilized a shared memory tile to write in a coalesced manner on a temporary variable and then transpose data back on *Lap*. The code snippet for face $x = 1$ is shown as a reference (face $x = 1$ is responsible also for $x = 0$):

```
1  if(j<=Ny_c+1 .and. k<=Nz_c+1) then
2
3      if(i==1) then
4          tile(js,1,ks) = D2x(1,1) * u(j,0,k,var) + D2x(1,2) * u(j,1,k,var) +
                           D2x(1,3) * u(j,2,k,var) + D2ox(1) * u(j,3,k,var)
5
6          tile(js,0,ks) = D2x(0,2) * u(j,0,k,var) + D2x(0,3)* u(j,1,k,var)
7
8          if(j.ne.1 .and. j<Ny_c) then
9              tile(js,1,ks) = tile(js,1,ks) + D2y(j,1) * u(j-1,1,k,var) +
                               D2y(j,2) * u(j,1,k,var) + D2y(j,3)* u(j+1,1,k,var)
10
11             if(k.ne.1 .and. k<Nz_c) tile(js,1,ks) = tile(js,1,ks) + D2z(k,1) *
                   u(j,1,k-1,var) + D2z(k,2) * u(j,1,k,var) + D2z(k,3) * u(j,1,k+1,var)
12             end if
13         end if
14
15         ...
```

Listing 3.6. *ComputeLap_FaceXT*: Laplacian computation at faces x=1 and x=0

The high cost of velocity 3D-transposition is amortized by the fact that it is also used by other kernels, like NLT computation. For this reason it is calculated once and for all at the beginning of the subroutine that computes right-hand side. The same strategy presented has been adapted and used for computation of the NLT and Divergence, while a different procedure will be illustrated for the Gradient.

Performance overview

Table (3.2) shows a comparison between *Compute_Lap_Base* and a single kernel doing the entire computation. The data are obtained using Nsight-Compute for a grid of 256^3. Using a single kernel to fully execute the algorithm resulted to an increase in branch diverging and in the use of registers. From the results we can see that register pressure considerably reduces occupancy. This generally translates into a

	Exec. Time[μs]	Registers/Thread	Achieved Occ.[%]
Compute_Lap_Base	0.374 *ms*	40	69.09
"All-in" kernel	0.803 *ms*	64	47.24

Table 3.2. Data collected with Nsight-Compute for different Laplacian strategy tested on V100

reduced ability of the GPU to hide latency of stalled warps and therefore in a loss of performance. For this reason we decided to separate the internal part from the faces, to better use the resources made available by the GPU in the computationally more intensive part. This strategy is also possible because the overhead associated with launching kernels is very low. Overhead that can be further decreased by the fact that host can queue up multiple device kernel launches so that they execute back to back. A strategy similar to that explained in Section (3.2.1) was also tested and the results are represented overall in Fig.(3.3) as the grid size varies. It consists of a much

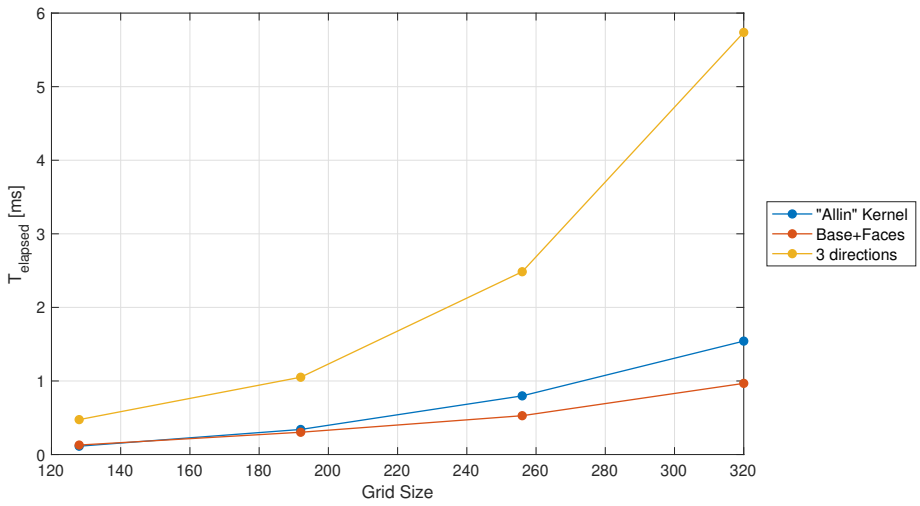


Figure 3.3. Execution time for different Laplacian implementation strategies performed on V100 as the grid increases

simpler approach than the proposed solution as the threads are mapped into one of the 3 possible faces of the domain and calculate Laplacian along the direction associated with the face. Much of the loss in performance is due to the fact that the kernel responsible for computation along the x direction requires transposition of data twice to make coalescing possible, once for reading u and once for writing Lap. Measurements for this work were taken using CUDA Events. The CUDA Events API provides calls that create and destroy events, record events (via a GPU timestamp), and convert timestamp differences into a floating-point value in units of milliseconds. The *cudaEventRecord()* function is used to place 2 events at the start and end of the GPU computation we want to measure. The device will record a timestamp every time it reaches an event in its stream. Finally, using *cudaEventElapsedTime()* function we can obtain the time elapsed between the two events.

3.2.3 Gradient Kernel

In the following section the differences between Gradient and previous computation will be explained. It should be emphasized that the algorithm uses as input a scalar variable (p) utilized to compute first derivative, discretized by centered finite differences, in the 3 spatial directions ($Grad$). The treatment of the boundary conditions for each component of $Grad$ is represented in Fig.(3.4). Gray parts represent points where first

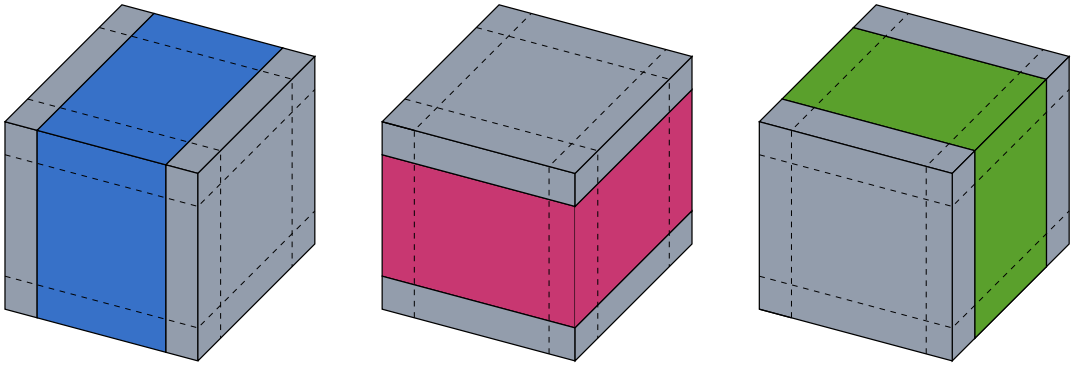


Figure 3.4. Abstract domain decomposition for the computation of *Grad* components. Gray parts represent points that have a modified stencil. Dotted lines indicate boundary faces characterized by a mixed stencil. There are no clear subdivision between internal points and boundary faces.

derivative stencil needs to be modified, to take into account Neumann or Periodic conditions. Compared to the Laplacian computation, it is evident that there is no longer a clear subdivision into internal and external points. Calculating the 3 components of *Grad* with a "Compute_Base-like" kernel requires threads mapped on the entire computational domain and then a more elaborate control flow to treat boundary faces (3 *Grad* components are computed in one kernel). So it goes against the idea of using a strategy like the one presented in the previous paragraph. Furthermore, there are no other algorithms that require the transpose of p and therefore it is not convenient to compute it specifically for the FaceX kernel. Another way is to use a kernel for each of the 3 spatial derivatives (referred in this paragraph with "*3 derivatives kernels*"). This strategy is penalized by the fact that we cannot effectively

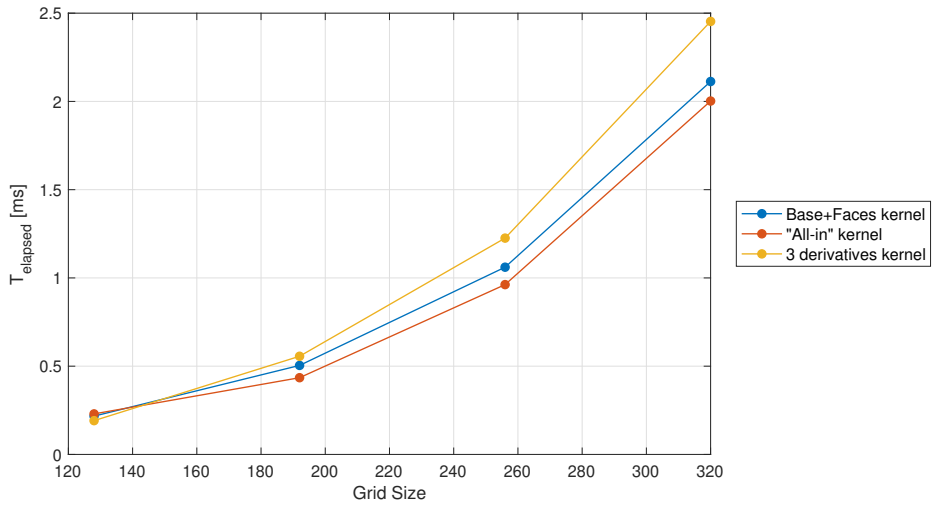


Figure 3.5. Execution time for different Gradient implementation strategies performed on V100 as the grid increases

use shared memory, as we would load the same data 3 times. Utilizing only global memory is not a problem for the Volta $V100$ for such a small stencil but it could be

for other clusters based on not so modern architectures. These 2 strategies have been compared together with the computation of the entire algorithm in a single kernel. The results are shown in Fig.(3.5). Overall, the *"All-in"* solution performed better and therefore was preferred respect to the others.
Despite this, the results obtained are quite close to each other. The reason is that, for all solutions, the warps are mainly stalled waiting for a global memory operation. This condition is reported in Nsight-Compute with the metrics *Stall Long Scoreboard*, indicating cycles spent on average by each kernel waiting for a scoreboard dependency on a L1Tex operation. Although the *"All-in"* kernel presents the best result for this

	Compute_Base_Grad	*"All-in"* kernel	3 deriv. kernels
Stall Long Scoreboard	28.09	14.50	43.61

Table 3.3. Stall Long Scoreboard metrics collected by Nsight-Compute for different Gradient implementation tested on V100

parameter, it can not achieve a better performance because the occupancy for this solution is lower, reducing the ability to hide latencies. The kernels that make use of the cache implicitly (indicated with "*3 deriv. kernels*" in the table) reach an occupancy that is close to 100% while for the solution adopted the Achieved Occupancy is barely more than 50%.

3.3 Multi-GPU communication

Communication between GPUs was carried out using the NVIDIA Collective Communications Library (NCCL) [20]. Within NCCL we can find a series of optimized multi-GPU communication primitives that allow us to exploit technologies such as NVLink high-speed interconnects within a node and NVIDIA Mellanox Network between different nodes. The advantage of using NCCL is that communications that generally require a combination of multiple operations (for example the *AllReduce* requires to perform a CUDA memory copy and the actual reduction), are implemented in a single kernel that handles both, improving performances and reducing synchronization time. The need to optimize the application for other clusters is also minimized, because the communication functions are topology-aware and support a variety of interconnect technologies.
CUDA library NCCL is accessible in Fortran through the pre-built interface module *nccl* provided by NVIDIA, which uses the *iso_c_binding* to call directly into the C library function. It is necessary to provide the following compiler options to use NCCL [21]: `-cudalib=nccl` which adds the library to the link line, and `--gpu=cc70` which compiles for compute capability 7.0.
Resuming Section (2.3.1), assembling each processor's contribution to the right-hand side of the Schur Complement equation requires communication. Since the computation is maintained entirely on the GPUs, it is necessary to create sub-communicators between the CUDA devices as indicated in the Fig.(3.6). This process splitting was done using MPI for the CPU version of the solver. Without going into implementation details, for each block, a color was assigned for each direction (color means an integer

identifier) and MPI_COMM_SPLIT was used to create the sub-communicators between processes of the same color. To create the sub-communicators between devices

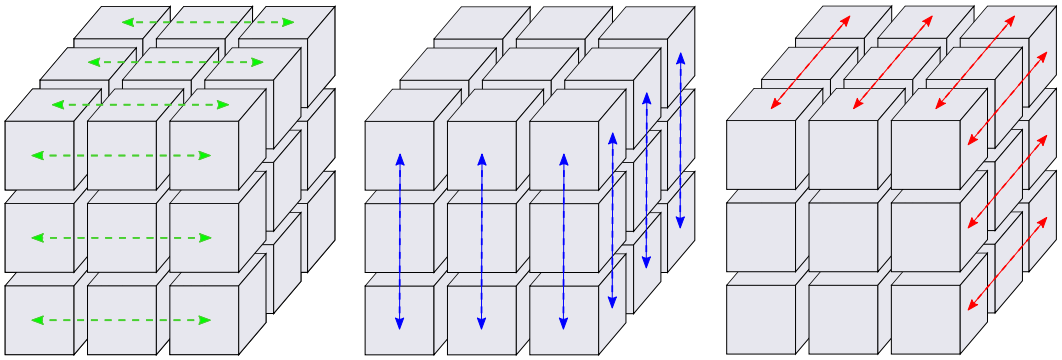


Figure 3.6. Example of sub-communicators between CUDA devices on a domain decomposed by 3 blocks in each dimension

starting from the MPI ones, the *ncclGetUniqueId* and *ncclCommInitRank* functions were used as indicated in the following fragment:

```
1  ! Generate the NCCL GPU 1D communicators
2    DO i = 1, dim
3
4      if(bloc%my1Drank(i)==0) then
5        res = ncclGetUniqueId(bloc%id(i))
6        if(res .ne. ncclSuccess) write(*,*) ncclGetErrorString(res)
7      end if
8
9      sizeID = sizeof(bloc%id(i))
10     CALL MPI_BCAST(bloc%id(i),sizeID,MPI_BYTE,0,bloc%my1Dcomm(i),ierr)
11
12     res = ncclCommInitRank(bloc%my1DcommGPU(i),bloc%my1Dsize(i),bloc%id(i),bloc%
       my1Drank(i))
13     if(res .ne. ncclSuccess) write(*,*) ncclGetErrorString(res)
14
15     res = ncclCommUserRank(bloc%my1DcommGPU(i),bloc%my1DrankGPU(i))
16     if(res .ne. ncclSuccess) write(*,*) ncclGetErrorString(res)
17
18     res = ncclCommCount(bloc%my1DcommGPU(i),bloc%my1DsizeGPU(i))
19     if(res .ne. ncclSuccess) write(*,*) ncclGetErrorString(res)
20
21   END DO
```

Listing 3.7. NCCL, creating GPU 1D sub-communicators

The data structure *bloc* contains object, rank and size of the MPI sub-communicators in the entries bloc%my1Dcomm(dim), bloc%my1Drank(dim) and bloc%my1Dsize(dim) respectively. With *ncclGetUniqueId* we can create an identification ID that we are going to distribute with MPI_BCAST to the devices we want to include in a new communicator (each process is linked to a single device). The ID will be used by *ncclCommInitRank* to create the new nccl communicator object, equivalent to the MPI sub-communicator in size and component rank. This procedure is done once and for all at the beginning of the time loop. Once the sub-communicators have been created correctly, we can use the *AllReduce* operation within the Schur solvers by:

```
res = ncclAllReduce(xint, xint, nproc*Ny*Nz, ncclFloat64, ncclSum, my1Dcomm, stream)
if(res .ne. ncclSuccess) write(*,*) ncclGetErrorString(res)

istat = cudaStreamSynchronize(stream)
```

Listing 3.8. NCCL, Utilization of the collective communication primitive AllReduce

Entering *xint* for both Sendbuff and Recvbuff arguments allows to perform an in-place operation in which the result of the *ncclSum* is saved in the variable passed as input. *cudaStreamSynchronize(stream)* blocks the execution of the host until all previous operations on the device associated with the given stream have been completed. Synchronization needed because the NCCL call returns when the operation has been effectively enqueued to the given stream, or returns an error. The collective operation is then executed asynchronously on the CUDA device. Synchronization ensures that at least for the process under consideration the reduction has been completed.

Chapter 4

Results and Performances

The purpose of this section is to show the porting results and performances compared to the CPU version of the code. To demonstrate the validity of the results, the well known cavity problem is used, compared with the one previously validated in [1]. Obviously this is not the only validation work that has been done, each kernel has been tested to produce the same results as the original CPU code. This approach is generally applicable and ensures that the two versions of the code give the same results, at least up to machine accuracy. Regarding performances, a comparison between scalability properties (strong and weak) of the two versions is presented. The CPU scaling results were made available by Dr. Alessandro Chiarini, previously measured on the Galileo supercomputer at CINECA and presented in [1].

4.1 3D Cavity Flow

This test problem consist in solving the 3D lid-driven cavity problem, following [1], in the domain $\Omega = (0,1) \times (0,1) \times (-1,1)$ at $Re = \frac{V_w h}{\nu}$, where V_w is the driven lid velocity, h is the length of the cavity edge in x and y directions, ν the cinematic viscosity. Face at $x = 1$ slides upward with imposed velocity $\mathbf{u} = (0,1,0)$, while Homogeneous Dirichlet boundary conditions are imposed on the other walls. A grid consisting of $(200 \times 200 \times 400)$ intervals, in the 3 directions respectively, has been used. Simulation has been performed up to $t = 8$, with a time-step $\Delta t = 3 \cdot 10^{-4}$, utilizing the Crank-Nicolson time scheme. The resulting CFL was slightly less than 0.4, satisfying the stability margin of this scheme. Let N_x the number of grid points in the x direction, then the coordinates of the grid points are defined as:

$$
\begin{aligned}
C_x &= 1 + (\frac{2}{L_x})^{\frac{1}{2}} \\
X_i &= \frac{i-1}{N_x - 1} L_x, \qquad i = 1, ..., N_x \\
x_i &= \begin{cases} C_x X_i^{\frac{3}{2}} (1 + X_i^{\frac{1}{2}})^{-}1 & if\ X_i \leq \frac{L_x}{2} \\ L_x - C_x (L_x - X_i)^{\frac{3}{2}} (1 + (L_x - X_i)^{\frac{1}{2}})^{-}1 & otherwise \end{cases}
\end{aligned}
\tag{4.1}
$$

to properly resolve boundary layers without excessively stretching grid cells.

Fig.(4.1) is a colour plot of the magnitude of the velocity vector in the $z = 0$ plane at

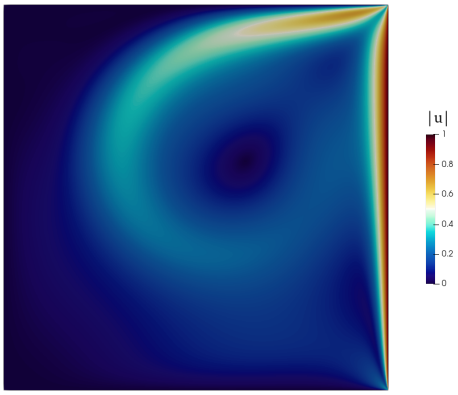


Figure 4.1. 3D driven cavity computed with Tesla V100 GPUs: velocity magnitude at $z = 0$, time $t = 8$, $Re = 1000$.

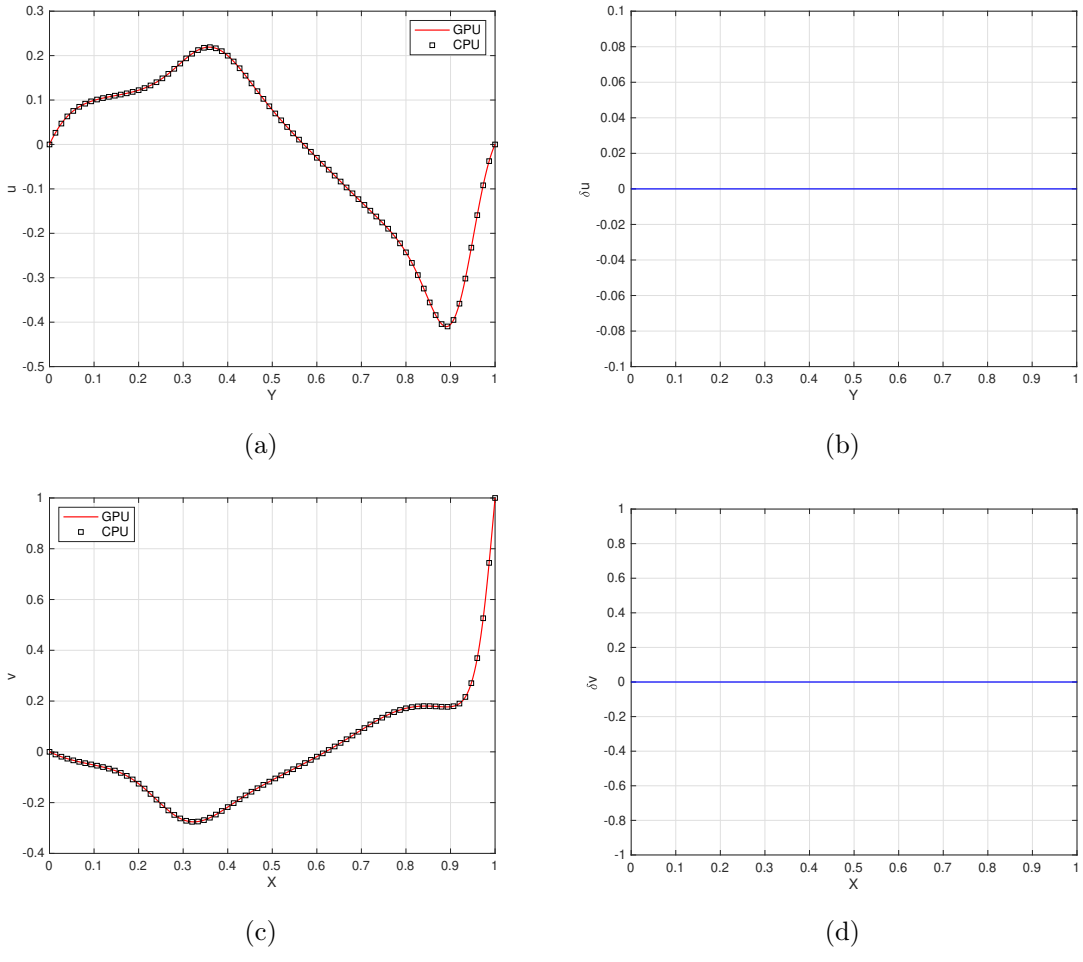


Figure 4.2. (a) First component of the velocity vector along the segment $\{x = 0.5, y \in [0, 1]\}$ in the $z = 0$ plane. (b)(d) Velocity errors between GPU and CPU computations along the same segments. (c) Second component of the velocity vector along the segment $\{y = 0.5, x \in [0, 1]\}$ in the $z = 0$ plane.

$t = 8$. The figure portraits qualitatively the same solution reported in [1].
Fig.(4.2) shows the u and v velocity components along the vertical and horizontal lines in the plane $z = 0$ passing through the point $(0.5, 0.5, 0)$ at time $t = 8$. Fig.(4.2b) and Fig.(4.2d) shows that velocity errors between GPU and CPU versions of the solver are zero. This is due to the fact that implementation effort has been spent on preserving the order of operations. Such a result is also possible because no math library functions were needed for this problem (e.g. sin, cos, etc.). For many of the simpler math functions there are known methods to achieve correctly rounded results for all possible arguments, however this comes at significant cost to execution speed. Since math libraries have to serve the needs of a large and diverse set of cases and applications, targeting correctly rounded results regardless of performance is rarely the chosen choice. Therefore, there are different implementation between GPU and regular CPU math library functions, leading both to correct results within the double precision accuracy.

4.2 Performance comparison

Scalability, or scaling, is the ability of hardware and software to produce greater computational power when the amount of available resources increases. Applications can generally be divided into strong scaling and weak scaling applications. In the case of strong scaling, problem size is kept fixed and the number of cores/GPUs to execute the program is increased. An application scales ideally when it present linear behaviour between speedup and increasing number of cores/GPUs. Clearly, as resources increase, the distributed workload decreases, but the cost of communication and synchronization between different processes increases.
When we perform a weak scaling study, on the other hand, we are carrying out a complementary work to that done for strong scaling. The size of the problem is not fixed and is increased relative to the increase of resources. In this case we ideally want the runtime to remain constant, as the work per core (or per GPU) has remained the same.

4.2.1 Problem Definition

The following manufactured solution of the Navier-Stokes equation was used to perform the scalability tests:

$$\begin{aligned} u &= \sin(x)\cos(t+y)\sin, \quad v = \cos(x)\sin(t+y)\sin(z), \\ w &= 2\cos(x)\cos(t+y)\cos(z), \quad p = \frac{3}{Re}\cos(x)\cos(t+y)\cos(z) \end{aligned} \tag{4.2}$$

The body force $\mathbf{f}$ that closes the momentum equation for this manufactured solution reads:

$$
\begin{aligned}
f_x &= \frac{\sin(x)}{Re}\Big[\frac{1}{2}(Re\cos(x)(1+2\cos(2(t+y))+\cos(2z))) \\
&\qquad -3\cos(t+y)(\cos(z)-\sin(z))-Re\sin(t+y)\sin(z)\Big], \\
f_y &= \frac{1}{4Re}[Re\cos^2(x)(3+\cos(2z))\sin(2(t+y))-2Re\sin^2(x)\sin(2(t+y))\sin^2(z)+ \\
&\quad 4\cos(x)(-3\cos(z)\sin(t+y)+(Re\cos(t+y)+3\sin(t+y))\sin(z))], \\
f_z &= \frac{1}{Re}[\cos(x)(-2Re\cos(z)\sin(t+y)+\cos(t+y)(6\cos(z)-3\sin(z))) \\
&\quad -\frac{1}{2}(Re\cos^2(x)(3+\cos(2(t+y)))\sin(2z))-Re\cos^2(t+y)\sin^2(x)\sin(2z)],
\end{aligned}
\tag{4.3}
$$

The problem is solved in a cubic domain for $Re = 1$. We impose Dirichlet boundary conditions for the velocity, calculated using the exact boundary solution. The initial condition is also set by the exact solution to $t = 0$

4.2.2 Scalability Results

The CPU version have been run on the GALILEO supercomputer at CINECA. It consists of 1022 36-core computing nodes connected through an Intel OmniPath (100Gb/s) high-performance network. Each node contains two 18-cores Intel Xeon E5–2697 v4 (Broadwell) at 2.30 GHz. The GPU version were performed on MARCONI100 supercomputer at CINECA. It consists of 980 nodes connected with a Mellanox Infiniband EDR network arranged into an architecture called DragonFly++. Each

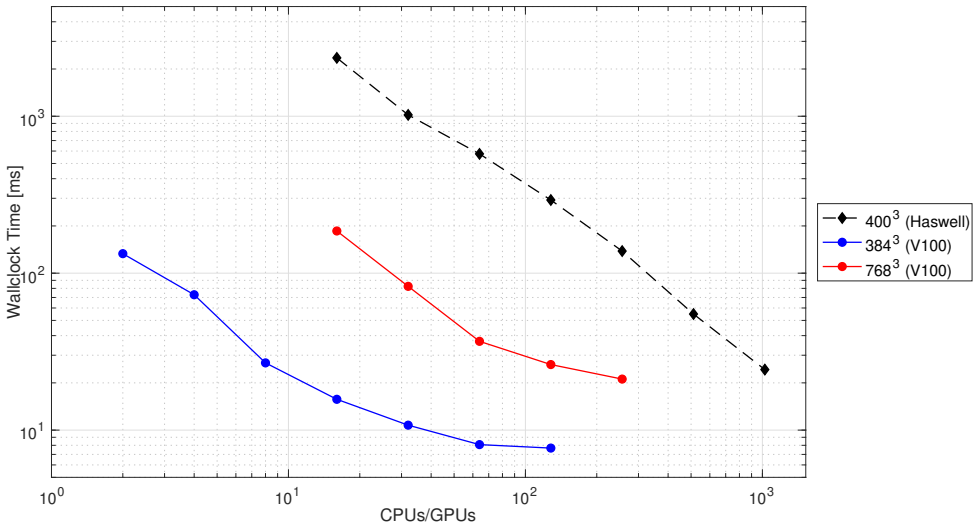


Figure 4.3. Wallclock Time per time-step for different fixed grids, measured for both CPU and GPU versions of the code.

node consists in 2 IMB POWER9 AC922 sockets, each of them with 16 cores and 2 16Gb NVIDIA Volta V100 GPUs (for a total of 32 cores and 4 GPUs per node).

GPUs are connected with the NVLink (NV3) within a socket. M100 is currently number 9 in the Top500 List of the fastest supercomputer in the world, reaching a Theoretical Peak Performance per node of 32 TFlops. Fig.(4.3) shows the wall-clock time per time-step for both CPU and GPU versions of the code on different fixed grids. The measurements were carried out using CUDA Events. Only the time loop was measured, without initial procedure and post-processing part of the code. Usually the influence of these parts on the total execution time is very low; time loop is usually performed thousands of times in such application. Compared to the CPU code, the strong scalability no longer has an ideal behavior. In particular, it has been noticed that increasing the number of devices when grids point per block are about 100^3, slightly reduces the time spent in computation (already very low) but introduces an increasing overhead in communication (probably due to GPU-CPU data transfers and synchronizations). In order to test the simulation at an higher number of GPUs was therefore used a grid of 768^3 points.
If we focus on Table (4.1), already a simulation performed on 8 GPUs reached an execution time comparable to that of 1024 CPUs, a result that highlights the acceleration obtainable by exploiting the GPUs.
Fig.(4.4) shows the weak scaling test. The number of points per GPUs was set at 256^3

Cores/GPUs	Galileo Haswell 400^3	Marconi100 V100 384^3	Marconi100 V100 768^3
2	-	133.14 ms	-
4	-	73.86 ms	-
8	-	26.81 ms	-
16	2354.4 ms	15.71 ms	185.71 ms
32	1019.9 ms	10.75 ms	82.43 ms
64	576.6 ms	8.08 ms	36.79 ms
128	292.5 ms	7.68 ms	26.16 ms
256	183.3 ms	-	21.14 ms
512	54.9 ms	-	-
1024	24.3 ms	-	-

Table 4.1. Wall-clock time per time step on various grid and hardware.

because it was considered more balanced between computation and communication respect to the 100^3 used for the CPUs version. Figure shows the execution time per grid points multiplied by 10^6. Even the GPU version of the code has good weak scalability, at least up to 256 GPUs then Fig.(4.4) hints the beginning of performances degradation. However, this degradation is mitigated due to the fact that we need less MPI processes to deliver excellent performance on large-scale grid respect to CPU-version.

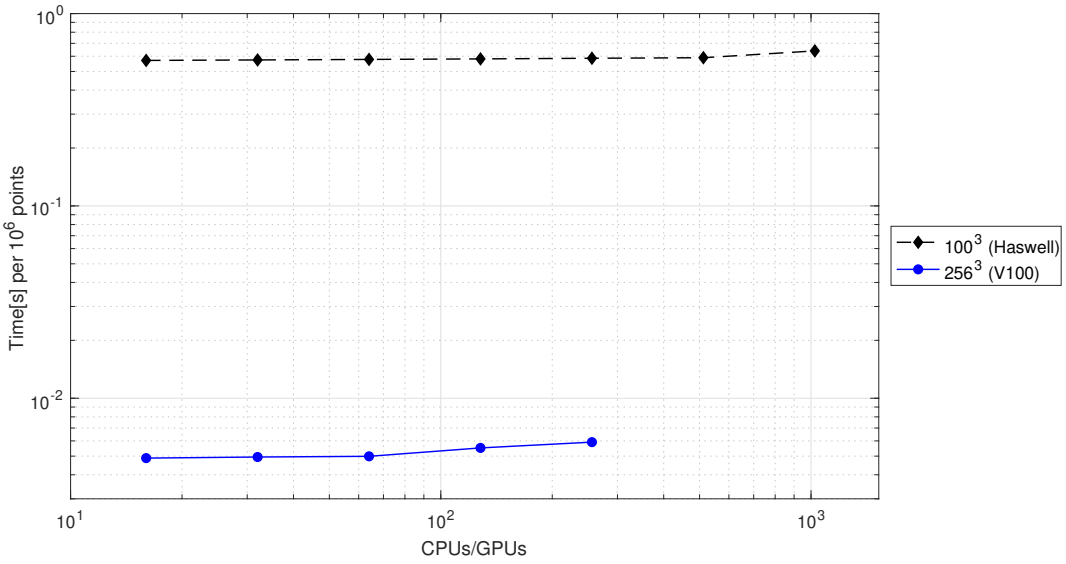


Figure 4.4. Weak Scalability: time spent for computing 10^6 points, plotted against the number of Cores/GPUs used.

Conclusions and Future Developments

In this work a finite difference solver for the incompressible Navier-Stokes equations has been developed, based on the code made available by Chiarini-Quadrio-Auteri, that exploits the computational power of modern GPU architectures. The work was motivated by the growing trend in recent years of using GPUs and accelerators in high performance computing. Although the starting solver was already proved to be efficient, it has been shown that an excellent performance, for a grid of similar size, can be obtained with a smaller number of GPUs than the CPU version. And it is possible to move to larger-scale grids reaching performance that would hardly be possible on the CPU, except with a very large number of cores. Currently, the CPU version is ready to be used in combination with the immersed boundary technique, for example to study moving boundaries or complex fluid-structure interaction problems. The GPU version requires passing to the host to perform interpolation of the immersed boundary as a GPU porting has not been yet developed. The next step will be to efficiently integrate the immersed boundary technique into the GPU solver and test it on a medium-scale preliminary problem. The goal will be to simulate the BARC test case by for the first time provides the body with finite aspect ratio. The simulation will be carried out in the unstable laminar regime, as a starting point, to then use the code on future larger applications. It should be emphasized that the code presented at the moment is mostly GPU-centric, the GPU has all the variables in memory to perform the time advancement and the CPU cores are mainly idle. We are not fully exploiting the potential provided by heterogeneous computing. The CPU cores can provide a boost both from the performance point of view but above all from the memory point of view (exploiting asynchronous data transfer).

Bibliography

[1] A. Chiarini, M. Quadrio, and F. Auteri. "A direction-splitting Navier–Stokes solver on co-located grids". In: *Journal of Computational Phisics* (2021).

[2] J. Guermond and P. Minev. "A new class of massively parallel direction spliting for the incompressible Navier–Stokes equations". In: *Comput. Methods Appl. Mech. Eng.* 200 (2011), pp. 2083–2093.

[3] Gregory Ruetsch and Massimiliano Fatica. *CUDA Fortran for Scientists and Engineers, Best Practices for Efficient CUDA Fortran Programming*. San Francisco, CA: Morgan Kaufmann, 2013.

[4] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 3rd ed. San Francisco, CA: Morgan Kaufmann, 2016.

[5] Peter S. Pacheco. *An introduction to Parallel Programming*. San Francisco, CA: Morgan Kaufmann, 2011.

[6] Jaegeun Han and Bharatkumar Sharma. *Learn CUDA Programming*. Birmingham, UK: Packt Publishing Ltd., September 2019.

[7] NVIDIA Corporation. *NVIDIA Tesla V100 GPU architecture. The world's most advanced data center GPU*. August, 2017.

[8] Z. Jia et al. "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking". In: (April 18th, 2018).

[9] J.L. Guermond, P.D. Minev, and Jie Shen. "An overview of projection methods for incompressible flows". In: *Comput. Methods Appl. Mech. Engrg.* 195 (2006), pp. 6011–6054.

[10] A.J. Chorin. "Numerical solution of the Navier–Stokes equations". In: *Math. Comput.* 22.2 (1968), pp. 745–762.

[11] R. Temam. "Sur l'approximation de la solution des equations de navier-stokes par la method des pas fractionnaires". In: *Arch. Ration. Mech. Anal.* 33 (1969), pp. 377–385.

[12] K. Goda. "A multistep technique with implicit difference schemes for calculating two- or three-dimensional cavity flows". In: *J. Comput. Phys.* 30 (1979), pp. 76–95.

[13] J. van Kan. "A second-order accurate pressure-correction scheme for viscous incompressible flow". In: *SIAM J. Sci. Stat. Comput.* 7.3 (1986), pp. 870–891.

[14] L.J.P. Timmermans, Minev. P.D., and F.N. Van De Vosse. "An approximate projection scheme for incompressible flow using spectral elements". In: *Int. J. Numer. Methods Fluids* 22 (1996), pp. 673–688.

[15] J. Guermond and P. Minev. “A new class of fractional step techniques for the incompressible Navier–Stokes equations using direction splitting”. In: *Compt. Rend. Acad. Sci., Mathematique* 348 (2010), pp. 581–585.

[16] S. Faure. “Stability of a colocated finite volume scheme for the navier–stokes equations”. In: *Numer. Methods Partial Differential Eq.* 21 (2005), pp. 242–271.

[17] *NVIDIA CUDA Fortran Programming Guide*. URL: `https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/index.html`.

[18] X. Zhu et al. “AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters”. In: *Comp. Phys. Comm.* 299 (2018), pp. 199–210.

[19] P. Micikevicius. *3D Finite Difference Computation on GPUs using CUDA*. NVIDIA San Tomas Expressway, Santa Clara, CA 95050.

[20] *NVIDIA Collective Communication Library (NCCL) Documentation*. URL: `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html`.

[21] *NVIDIA CUDA Fortran Interfaces*. URL: `https://docs.nvidia.com/hpc-sdk/compilers/fortran-cuda-interfaces/index.html`.