UPC **CTTC**

Heterogeneous parallel algorithms for Computational Fluid Dynamics on unstructured meshes

Centre Tecnològic de Transferència de Calor
Departament de Màquines i Motors Tèrmics
Universitat Politècnica de Catalunya

Guillermo Andrés Oyarzún Altamirano
Doctoral Thesis

Heterogeneous parallel algorithms for Computational Fluid Dynamics on unstructured meshes

Guillermo Andrés Oyarzún Altamirano

TESI DOCTORAL

presentada al

Departament de Màquines i Motors Tèrmics
E.T.S.E.I.A.T.
Universitat Politècnica de Catalunya

per a l'obtenció del grau de

Doctor per la Universitat Politècnica de Catalunya

Terrassa, September 2015

Heterogeneous parallel algorithms for Computational Fluid Dynamics on unstructured meshes

Guillermo Andrés Oyarzún Altamirano

Directors de la Tesi

Dr. Ricard Borrell Pol
Dr. Assensi Oliva Llena

This thesis is dedicated to

my parents, Alicia and Juan,

and my beloved, Anthi.

....Keep Ithaka always in your mind.
Arriving there is what you are destined for.
But do not hurry the journey at all.
Better if it lasts for years,
so you are old by the time you reach the island,
wealthy with all you have gained on the way,
not expecting Ithaka to make you rich.
Ithaka gave you the marvelous journey.
Without her you would not have set out.
She has nothing left to give you now.
And if you find her poor, Ithaka won't have fooled you.
Wise as you will have become, so full of experience,
you will have understood by then what these Ithakas mean.

section of C. P. Cavafy's poem Ithaka

Esta tesis esta dedicada a

mis padres, Alicia y Juan,

y a mi amada, Anthi.

....Ten siempre en tu mente a Ítaca.
La llegada allí es tu destino.
Pero no apresures tu viaje en absoluto.
Mejor que dure muchos años,
y ya anciano recales en la isla,
rico con cuanto ganaste en el camino,
sin esperar que te dé riquezas Ítaca.
Ítaca te dio el bello viaje.
Sin ella no habrías emprendido el camino.
Pero no tiene más que darte.
Y si pobre la encuentras, Ítaca no te engañó.
Así sabio como te hiciste, con tanta experiencia,
comprenderás ya qué significan las Ítacas.

extracto del poema Ítaca de C.P. Cavafy

Acknowledgements

This thesis is the finish line of a marathonian race that began several years ago. The road has not been exempted of hurdles and challenges, only overcome with the help of different people along in my life, I dedicate this work to all of them.

First of all, I would like to thank my grandparents. In particular my grandpa, for teaching me the importance of looking for knowledge, that finally ended driving me into the scientific world.

I would like to thank the teachers Wagner and Stuckrath, for their didactic, yet precise and motivational math classes that inspired me in pursuing an engineer degree.

From UTFSM, Prof. Salinas motivate me to studying topics related with scientific computing and helped me to continue my studies in Europe.

This thesis is possible thanks to Prof. Assensi Oliva, head of the *Heat and Mass Transfer Technological Center* (CTTC), for giving me the chance to pursue this PhD, his constant help along these years in all kind of matters and for allowing me to contribute in this non-conventional subjects.

I am very grateful of Ricard's guidance. He was always helpful and understanding during the development of this thesis. Definitely, this work would not be possible without him.

I was also lucky to have the help of the *brutal Guru*, Andrey. His wisdom and hard criticism always shed some light in our darkest moments.

To all Termosecta F.C. historical members. I treasure the moments I spent with the team in the field as player, and later as the coach.

To my friends that become a family in Spain, specially: Xiaofei Hou, Lluís Jofre, Aleix Báez, Santiago Torras, Federico Favre, Eugenio Schillaci and Pedro Galione.

All the rest of the CTTC members for their direct or indirect collaboration in the writing of this thesis.

I would like to thank my parents, because of their unconditional love and support in every decision I have made in my life. Without them I would not be here and nothing of this would be possible. Finally, I would like to thank the love of my life, Anthi. Her company, positive energy and patience has motivated and given me strength in the most difficult moments.

Agradecimientos

Esta tesis es la meta de una carrera maratoniana que empezó muchos años atrás. El camino no ha estado exento de obstáculos y desafíos, que han sido superados sólo gracias a la ayuda de diferentes personas a lo largo de mi vida, este trabajo está dedicado a todos ellos.

Primero que todo, me gustaría agraceder a mis abuelos. En particular al Tata, por enseñarme desde pequeño la importancia de la búsqueda del conocimiento, con sus píldoritas del saber o revistas científicas, que finalmente me encaminaron hacia el mundo de la ciencia.

Me gustaría agradecer a mis profesores Wagner y Stuckrath, por sus clases de matemáticas didácticas, precisas y motivacionales que me inspiraron en seguir un camino en la ingeniería.

En la UTFSM, agradecer al profesor Salinas, que me motivó a estudiar temáticas relacionadas con la computación científica y me ayudó a continuar mis estudios en Europa.

Esta tesis es posible gracias al Prof. Assensi Oliva, director del *Centro de Transferencia de Calor y Masa* (CTTC), por darme la oportunidad de seguir este doctorado, por su constante ayuda durante estos años en los distintos asuntos y problemas que surgieron, y por permitirme contribuir en estas temáticas no convencionales.

Estoy muy agradecido de mi tutor Ricard. Él fue siempre muy comprensivo y estuvo siempre dispuesto a tender una mano en el desarrollo de esta tesis. Definitivamente, este trabajo no hubiera sido posible sin él.

Yo fui muy afortunado de tener la ayuda de *el Gurú brutal*, Andrey. Su conocimiento y duro criticismo siempre nos proporcionó un rayo de luz en nuestros momentos más oscuros.

A todos los miembros históricos de Termosecta F.C.. Atesoro todos los momentos que compartimos en el campo de fútbol sala, primero como jugador y en mis últimos meses como director técnico.

A todos mis amigos que se convirtieron en mi familia en España, especialmente: Xiaofei Hou, Lluís Jofre, Aleix Báez, Santiago Torras, Federico Favre, Eugenio Schillaci y Pedro Galione.

A todo el conjunto de miembros del CTTC que colaboraron de forma directa o indirecta en el desarrollo de esta tesis.

Me gustaría agradecer a mis padres, por su amor incondicional y apoyo en cada decisión que he tomado en mi vida. Sin ellos yo no estaría aquí y nada de esto sería posible. Finalmente, me gustaría agradecer al amor de mi vida, Anthi. Su compañía, energía positiva y paciencia me ha motivado y dado la fuerza necesaria en los momentos más difíciles.

Abstract

The frontiers of computational fluid dynamics (CFD) are constantly expanding and eagerly demanding more computational resources. During several years the CFD codes relied its performance on the steady clock speed improvements of the CPU. In the early 2000's, the physical limitations of this approach impulsed the development of multi-core CPUs for continuing Moore's law trend. Currently, we are experiencing the next step in the evolution of high performance computing systems driven by power consumption constraints. Hybridization of the computing nodes is a consolidated reality within the leading edge supercomputers. The HPC nodes have incorporated accelerators that are used as math co-processors for increasing the throughput and the FLOP per watt ratio. On the other hand, multi-core CPUs have turned into energy efficient system-on-chip architectures. By doing so, the main components of the node are fused and integrated into a single chip reducing the energy costs.

Nowadays, several institutions and governments are investing in the research and development of different aspects of HPC that could lead to the next generations of supercomputers. This initiatives have entitled the problem as the *exascale challenge*. The idea consist in developing a sustainable exascale supercomputer ($1 \times 10^{18} FLOP/s$), with a power consumption of around 20MW. This represents a two order of magnitude improvement of the FLOP per watt ratio of the current machines. Which can only be achieved by incorporating major changes in computer architecture, memory design and network interfaces. As a result, a diversity of frameworks and architectures are competing with each other to become the prominent technology in the next generation of supercomputers.

The CFD community faces an important challenge: keep the pace at the rapid changes in the HPC resources. The codes and formulations need to be re-design in other to exploit the different levels of parallelism and complex memory hierarchies of the new heterogeneous systems. The major characteristics demanded to the new CFD software are: memory awareness, extreme concurrency, modularity and portability.

This thesis is devoted to the study of a CFD algorithm re-factoring for the adoption of new technologies. Our application context is the solution of incompressible flows (DNS or LES) on unstructured meshes. Hence, rather than focusing on the study of the physics associated to these flows, most of the work is focused on the implementation in the modern and leading edge architectures.

In particular, the contents of the four main chapters have been submitted or published in international journals and conferences. Therefore, they are written to be self-contained and only minor changes have been introduced with respect to the original sources. Consequently, some theoretical and numerical contents are repeated along them. Moreover, at the end there are two appendices including material that may be useful in order to understand some parts of this work, but that has been placed apart so that the normal reading of the thesis is not disturbed.

Our working plan started with the analysis of the key issues of the hybrid supercomputers based on GPU co-processors. The main difficulty is the required programming shift towards the stream processing model, so called Single Instruction Multiple Threads (SIMT) in the NVIDIA GPUs context. This new programming model requires major code modifications, based in creating new data structures that can be efficiently managed by the massively parallel devices. In the SIMT model the efficient execution heavy relies in the deep knowledge of the computing architecture. This makes difficult for a common user to design efficient CFD codes in those architectures. Then, our challenge was to integrate the SIMT execution model into our CFD code, so called TermoFluids, in the less invasive form.

The first approach was using GPUs for accelerating the Poisson solver, that is the most computational intensive part of our application. In particular, we targeted clusters with multiple GPUs distributed in different computing nodes and interconnected using the MPI standard. Different storage formats where studied for the sparse matrices derived from our discretizations and a two-level partitioning approach was introduced to perform the communications between host and devices. In terms of parallelism, the bottleneck of the algorithm was the communication process, therefore an overlapping of data transfer and computations was proposed in order to hide part of the MPI and the CPU-GPU communications.

The positive results obtained with the first approach motivated us to port the complete time integration phase of our application. This required major code refactoring. We decided not to focus only in GPUs, but to re-design the code so it could be easily be ported to other architectures as well. The code restructuring can be considered a big software engineering challenge, in which portability and modularity are the central objectives to be accomplished, but without losing the performance and the user friendly approach of TermoFluids. As a result, the second chapter of this thesis presents an algebraic based implementation model for CFD applications. The main idea was substituting stencil data structures and kernels by algebraic storage formats and operators. By doing so, the algorithm was restructured into a minimal set of algebraic operations. The implementation strategy consisted in the creation of a low-level algebraic layer for computations on CPUs and GPUs, and a high-level user-friendly discretization layer for CPUs that is fully localized at the preprocessing stage where performance does not play an important role. Therefore, at the time-integration phase the code relies only on three algebraic kernels: sparse-matrix-vector product (SpMV), linear combination of two vectors (AXPY) and dot product (DOT). Such a simple set of basic linear algebra operations naturally provides the desired portability. Special attention was paid at the development of data structures compatibles with the stream processing model. Improvements with respect to our first implementation were introduced in several sections of the code. In particular, the deeper knowledge of the architecture allowed us to enhance the code in terms of storage formats for the SpMV, reordering techniques for maximizing bandwidth attainment and better communication schemes were developed for overlapping purposes. A detailed performance analysis was studied in both sequential and parallel execution engaging up to 128 GPUs. The main objective was to understand the challenges and set some ground rules for attaining the maximum achievable performance in single or multiple GPU execution. In addition, an accurate estimation of the CFD performance based in the analysis of the separate kernels was proposed and showed great accuracy.

The exascale computing challenge has encouraged the study of new energy efficient technologies. In this context, the European project called Mont-Blanc was conceived to design a new type of computer architecture capable of setting future global HPC standards, built from energy efficient solutions used in embedded and mobile devices. The Mont-Blanc consortium is formed by 13 partners from 5 different countries joining industrial technology providers and research supercomputing centers. We were invited to test TermoFluids code in the Mont-Blanc prototypes. Therefore, our effort was focused in presenting one of the first CFD algorithms running in this novel architecture. The application of the portable implementation model proposed in chapter two was straightforward. However, additional tuning was required to take into account the specific characteristics of the mobile-based architecture. The Mont-Blanc nodes are composed by one fused CPU/GPU where memory is physically shared between both devices. This main characteristic removes the PCI-express overhead in the device communication and facilitates the implementation of heterogeneous algebraic kernels.

Chapter three exposed the work dedicated to the design of concurrent heterogeneous kernels using both computing devices CPU and GPU. The load balancing between the two devices exploits a tabu search strategy that tunes the workload distribution during the pre-processing stage. An overlap of computations with MPI communications was used for hiding at least partially the data exchange costs that becomes a bottleneck when engaging multiple Mont-Blanc nodes. Moreover, a comparison of the Mont-Blanc prototypes with high-end supercomputers in terms of the achieved net performance and energy consumption provided some guidelines of the behavior of CFD applications in ARM-based architectures.

Finally, the experience gained in the code re-factoring allowed us to identify other parts of TermoFluids that could be upgraded in order to exploit the new CPU technologies. In particular, we focused in the Poisson solver for discretizations with one periodic direction, that is the most time consuming part of such simulations. The current trend indicates that CPUs are increasing the number of vector registers, promoting the use of the Single Instruction Multiple Data (SIMD) extensions. Commonly known as vectorization, the SIMD model can be seen as a low level stream processing modeling, and therefore some of the concepts, acquired in the previous works, such as memory alignment and coalescence can be applied to CPUs as well.

Chapter four demonstrates the relevance of vectorization and memory awareness for fully exploiting modern CPUs. This work was developed and tested in the BlueGene/Q Vesta supercomputer of the Argonne Leadership Computing Facility (ALCF). In particular, we present a memory aware auto-tuned Poisson solver for problems with one Fourier diagonalizable direction. This diagonalization decomposes the original 3D system into a set of independent 2D subsystems.

The proposed algorithm focuses on optimizing the memory allocations and transactions by taking into account redundancies on such 2D subsystems. Moreover, we also take advantage of the uniformity of the solver through the periodic direction for its vectorization. Additionally, our novel approach automatically optimizes the choice of the preconditioner used for the solution of each frequency system, and dynamically balances its parallel distribution. Altogether constitutes a highly efficient and robust HPC Poisson solver.

The final chapter contains the main conclusions of this thesis and the future research path in order to continue with the challenge of achieving performance on the incoming pre-exascale hybrid supercomputers.

Contents

MPI-CUDA Sparse Matrix-Vector Multiplication for the Conjugate Gradient Method with an Approximate Inverse Preconditioner

Main contents of this chapter have been published in:

G. Oyarzun, R. Borrell, A. Gorobets and A. Oliva. MPI-CUDA Sparse Matrix-Vector Multiplication for the Conjugate Gradient Method with an Approximate Inverse Preconditioner. *Computers & Fluids*, 92:244–252, 2014.

Abstract. The preconditioned conjugate gradient (PCG) is one of the most prominent iterative methods for the solution of sparse linear systems with symmetric and positive definite matrix that arise, for example, in the modeling of incompressible flows. The method relies on a set of basic linear algebra operations which determine the overall performance. To achieve improvements in the performance, implementations of these basic operations must be adapted to the changes in the architecture of parallel computing systems. In the last years, one of the strategies to increase the computing power of supercomputers has been the usage of Graphics Processing Units (GPU) as math co-processors in addition to CPUs. This paper presents a MPI-CUDA implementation of the PCG solver for such hybrid computing systems composed of multiple CPUs and GPUs. Special attention has been paid to the sparse matrix-vector multiplication (SpMV), because most of the execution time of the solver is spent on this operation. The approximate inverse preconditioner, which is used to improve the convergence of the CG solver, is also based on the SpMV operation. An *overlapping* of data transfer and computations is proposed in order to hide the MPI and the CPU-GPU communications needed to perform parallel SpMVs. This strategy has shown a considerable improvement and, as a result, the hybrid implementation of the PCG solver has demonstrated a significant speedup compared to the CPU-only implementation.

1.1 Introduction

Scientific computing is constantly advancing due to the raising of new technologies. The architectures with single-processor nodes evolved into multi- and many-core platforms and, in the last years, Graphics Processing Units (GPU) have emerged as co-processors capable to handle large amount of calculations exploiting the data parallelism. However, except for some specific problems, obtaining the desired performance from GPU devices is far from being trivial and usually requires significant changes in the algorithms and/or its implementations. Indeed, GPU computing has motivated the creation of new parallel programming models such as the Compute Unified Device Architecture (CUDA) [1] developed by Nvidia or the Open Computing Language (OpenCL) [2] developed by the Khronos group. In particular, CUDA platform provides several numerical libraries, together with a software development kit, making it easier for software developers to encode new algorithms. Good results attained using CUDA for problems that show a fine-grained parallelism [3] aroused the interest of the High Performance Computing (HPC) community prompting the research on this field. The present work explores some aspects of a hybrid MPI-CUDA implementation of the preconditioned conjugate gradient (PCG) method, to be executed in multi-GPU platforms. The PCG is one of the methods of choice for the solution of symmetric and positive definite linear systems arising in many applications (*e.g.* discretizations of partial differential equations, computer analysis of circuits, chemical engineering processes, etc). In particular, our target is to accelerate the solution of the system derived from the discretization of the Poisson equation, which arises from the mass conservation constraint in the simulation of incompressible flows. This system has to be solved once per time step dominating, in general 3D cases, the computing costs. We assume that the mesh and the physical properties of the fluid are constant, so the system matrix remains constant during all the simulation as well. As a consequence preconditioners with high preprocessing costs are suitable, since its per-iteration relative costs become negligible. For this reason, and because of its parallel-friendly solution stage, we have chosen the approximate inverse preconditioner (AIP) to enhance the CG performance.

Different implementations of the CG method for GPUs have been published in the last years. One of the first attempts to run the CG on a GPU, using textures as streams and shader functions as kernels, can be found in [4]. With the advent of CUDA studies on iterative solvers, new implementations using different preconditioners, such as least squared polynomials, incomplete Cholesky factorizations or symmetric successive over relaxation smothers, have been presented in [5, 6]. The use of multi-GPU platforms on the CG can be found in [7], while ideas for minimizing the communication overhead through overlapping techniques are explained in [8]. Most of these works focus on implementations for a single-GPU or for multiple-GPUs located in a single node. We consider here the case of multiple GPUs distributed in different computing nodes, interconnected using the MPI standard. Both the CG and the AIP have the SpMV as the most time consuming part of the algorithm so we have mainly focused on the performance of this operation. The main bottleneck for the parallelization is related with the communications between different GPUs. As a consequence, a data transfer overlapping approach is used in order to minimize the communication expenses, by executing the data transfer and the MPI communications simultaneously with computations on the GPU.

The rest of the paper is organized as follows: Section 1.2 describes the fundamentals of the PCG solver and the AIP preconditioner. Section 1.3 points out our application context, the incompressible Navier-Stokes equations; in Section 1.4 the parallelization strategies and the CUDA model are explained; multi-GPU implementations of the operations that compose the solver are considered in detail in Section 1.5; Section 1.6 presents performance results; and finally, concluding remarks are stated in Section 1.7.

1.2 Preconditioned Conjugate Gradient solver

1.2.1 Conjugate Gradient

The conjugate gradient solver is an iterative projection method for the solution of systems

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where $\mathbf{A}$ is a symmetric and positive definite matrix. In the $i'th$ iteration, it is found the best possible solution, $\mathbf{x}_i$, for the system, respect to the A-norm, within the subspace $\mathbf{x}_0 + \mathbf{D}^i$, where $\mathbf{D}^i = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, ..., \mathbf{A}^{i-1}\mathbf{r}_0\}$, $\mathbf{r}_0$ is the initial residual, and the A-norm is defined as $\| \mathbf{v} \|_A = \sqrt{\mathbf{v}^T \mathbf{A}\mathbf{v}}$.

In the solution procedure, in order to find the approximation $\mathbf{x}_{i+1}$, firstly an A-orthogonal basis $\{\mathbf{d}_0, \mathbf{d}_1, ..., \mathbf{d}_i\}$ of $\mathbf{D}^{i+1}$ is fixed by means of the conjugate Gram-Schmidt process (adding a new element, $\mathbf{d}_i$, to the previously derived basis $\mathbf{d}_0, ...\mathbf{d}_{i-1}$ for $\mathbf{D}^i$):

$$\beta_i = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{r}_{i-1}^T \mathbf{r}_{i-1}}, \quad \mathbf{d}_i = \mathbf{r}_i + \beta_i \mathbf{d}_{i-1},$$

where $\mathbf{r}_i$ refers to the $i'th$ residual and β_i is the Gram-Schmidt constant. Then $\mathbf{x}_{i+1}$ can be represented in this new basis as

$$\mathbf{x}_{i+1} = \alpha_0 \, \mathbf{d}_0 + \ldots + \alpha_{i-1} \, \mathbf{d}_{i-1} + \alpha_i \, \mathbf{d}_i, \tag{1.1}$$

where, indeed, $\alpha_0, ..., \alpha_{i-1}$ are obtained from the previous iterations. Hence, in the $i'th$ iteration it is only necessary to evaluate the component α_i associated to $\mathbf{d}_i$:

$$\alpha_i = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{d}_i^T \mathbf{A} \mathbf{d}_i}.$$

Finally, the new approximation is:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{d}_i.$$

A detailed explanation of the CG algorithm and a sample of code can be found in [9]. Note that all the steps of the algorithm consist in basic linear algebra operations like SpMVs, dot products, vector additions/subtractions and operations with scalars. In the parallel executions, only two of these operations require data exchange between different parallel processes: the dot product and the SpMV.

1.2.2 Approximate Inverse Preconditioner

Sparse approximate inverse preconditioners (AIP) are based on the assumption that the inverse of the system matrix contains many small entries that may be neglected. This is the case of our application context (see Section 1.3). In the set-up process, a sparse approximation of the inverse, $\mathbf{M} \approx \mathbf{A}^{-1}$, constrained to a fixed sparsity pattern $\mathcal{S}$ has to be found. Then the equivalent, but better conditioned, system $\mathbf{MAx} = \mathbf{Mb}$ can be solved.

The preconditioned version of the CG algorithm [9] requires a multiplication by $\mathbf{M^{-1}}$ at each time step. In the case of the AIP, since this inverse is evaluated explicitly, the solution process consists only in a SpMV operation, making the algorithm attractive to be ported to GPUs. However, for symmetric and positive definite problems, in order to preserve symmetries, $\mathbf{A}^{-1}$ is approximated by a factorization $\mathbf{G}^T\mathbf{G}$ instead of a single matrix $\mathbf{M}$ (two products are then necessary instead of one), where $\mathbf{G}$ is a sparse lower triangular matrix approximating the inverse of the Cholesky factor, $\mathbf{L}$, of $\mathbf{A}$. In order to find $\mathbf{G}$, the approach to the problem is:

$$min_{G \in \mathcal{S}} ||\mathbf{I} - \mathbf{GL}||_F^2, \tag{1.2}$$

where $||.||_F$ is the Frobenius norm and $\mathcal{S}$ is a lower triangular sparsity pattern. We have followed Chow's work [10], in which it is explained how to find $\mathbf{G}$ without explicitly evaluate $\mathbf{L}$, *i.e.*, only using the initial matrix $\mathbf{A}$. Moreover in that work the sparsity pattern, $\mathcal{S}$, is fixed a priori as the pattern of a power of $\tilde{\mathbf{A}}$, where $\tilde{\mathbf{A}}$ is obtained from $\mathbf{A}$ dropping small entries. The power used to fix the sparsity pattern is referred as the sparsity level of the preconditioner. Here, for simplicity, we have considered $\mathbf{A} = \tilde{\mathbf{A}}$ (*i.e.* no elements are dropped).

In our application context (section 1.3) the system matrix $\mathbf{A}$ does not change during all the simulation, therefore the evaluation of $\mathbf{G}$ is considered a preprocess that is executed only once at the beginning of the simulation.

Different works proving the efficiency of the AIP strategy on the solution of sparse linear systems can be found in the literature [11–14]. In particular, this preconditioner has been successfully used in the resolution of linear systems that arise in the CFD context [15–17]. One of its main strengths is its parallel-friendly solution stage (it is composed of just two SpMV). This makes it specially attractive in the GPU context, in contrast with other traditional preconditioners such as the Incomplete Cholesky (IC) factorization that has an inherent sequential process (the backward and forward substitutions) that limits its parallelization potential in GPUs [5], or the multigrid-based preconditioners based in complicated multilevel operations difficult to fit in the single instruction multiple thread (SIMT) model, thus more study is needed to be efficiently ported to the GPUs [18, 19].

1.3 Application context: solution of Poisson system in the simulation of incompressible flows

In the numerical solution of incompressible flows, the Poisson equation arises from the continuity constraint and has to be solved at least once per time step. The velocity field, $\mathbf{u}$, is governed by the incompressible Navier-Stokes (NS) equations:

$$\nabla \cdot \mathbf{u} = 0, \tag{1.3}$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \Delta \mathbf{u} \tag{1.4}$$

TermoFluids code [20] has been used as a software platform to perform our studies. The equations are discretized on an unstructured grid with the finite-volume symmetry-preserving method [21]. For the temporal discretization, a second-order explicit one-leg scheme is used in [22]. Finally, the pressure-velocity coupling is solved by means of a fractional step projection method [23]. The steps of the projection method are: i) a predictor velocity, $\mathbf{u}^p$, is explicitly evaluated without considering the contribution of the pressure gradient in Equation 1.4; ii) the Poisson pressure equation is solved in order to meet the incompressibility constraint:

$$\Delta p^{n+1} = \nabla \cdot \mathbf{u}^p, \tag{1.5}$$

where p^{n+1} is the pressure field at time-step $n+1$. Further details on the numerical algorithm, the math model and the code functionalities can be found in [20,24,25].

The discrete counterpart of Equation (1.5) is referred as

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1.6}$$

where $\mathbf{A}$ is a symmetric and positive definite sparse matrix. With our finite-volume discretization the action of $\mathbf{A}$ on $\mathbf{x}$ is given by

$$[\mathbf{A}\mathbf{x}]_k = \sum_{j \in Nb(k)} \mathcal{A}_{kj} \frac{\mathbf{x}(j) - \mathbf{x}(k)}{\delta n_{kj}}, \tag{1.7}$$

where $Nb(k)$ is the set of neighbours of the k'th node. $\mathcal{A}_{kj}$ is the area of f_{kj}, the face between the nodes k and j, and $\delta n_{kj} = |n_{kj} \cdot v_{kj}|$, where v_{kj} and n_{kj} are the vector between nodes k and j, and the normal unit vector of f_{kj}, respectively. At the boundary Neumann conditions are imposed. Further details can be found on [26].

In general 3D cases, the solution of this linear system dominates the computing costs of the simulation. Assuming that the mesh and the physical properties of the fluid are constant, the system matrix remains constant during the simulation as well. In this situation, for simulations with large number of time-steps, it is suitable the use of a preconditioner with high preprocessing costs, since its per-iteration relative cost becomes negligible.

1.4 Parallelization strategy

1.4.1 MPI parallelization via domain decomposition

The underlying MPI parallelization of the solver is based on a domain decomposition approach. Therefore, the initial mesh $\mathcal{M}$, where the Poisson equation is discretized, is divided into P non-overlapping sub-meshes $\mathcal{M}_0, ..., \mathcal{M}_{P-1}$ by means of a graph partitioning tool such as METIS library [27]. For each MPI process, the corresponding unknowns of the system can be divided into different sub-sets: (1) *Owned* unknowns are those associated to the nodes of $\mathcal{M}_i$; (2) *Inner* unknowns are the owned unknowns that are coupled only with other owned unknowns; (3) *Interface* unknowns are those owned unknowns which are coupled with non-owned (external) unknowns; and finally (4) *Halo* unknowns are those external unknowns (that belong to other processes) which are coupled with owned unknowns of the MPI process considered. In Figure 1.1 an example of a domain decomposition is shown (left) and the sub-sets defined above are depicted for one of the sub-domains (right). Henceforth, the subscripts *owned, inner, interface and halo* are used in order to refer to the respective components of a vector or rows of a matrix. Note that halo components of vectors are required to solve the system in parallel. These elements have to be updated before being used if the corresponding external elements have changed. This process is carried out by means of point-to-point communications between adjacent sub-domains.

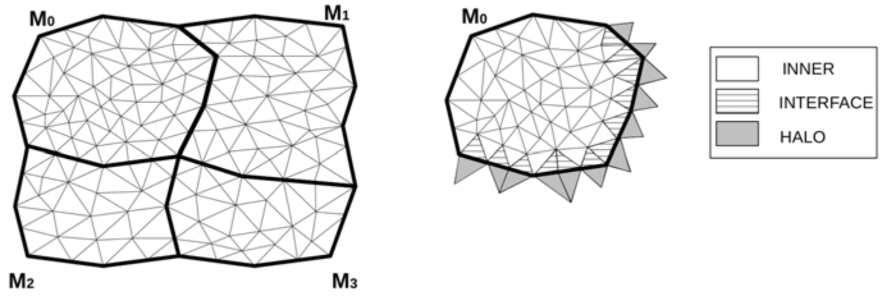


Figure 1.1: Left: Domain decomposition; Right: Distinction between the different subsets of unknowns associated to sub-domain 0.

For each sub-domain the unknowns are ordered as follows: first the inner unknowns, then the interface unknowns and, finally, the halo unknowns. Thus, the local components of any parallel vector $\mathbf{v}$ read $\mathbf{v}_{local} = [\mathbf{v}_{inner}|\mathbf{v}_{interface}|\mathbf{v}_{halo}]$). With this ordering, the part of the matrix stored by a MPI process, defined as $\mathbf{A}_{owned}$, has a block structure such as the one represented in Figure 1.2.

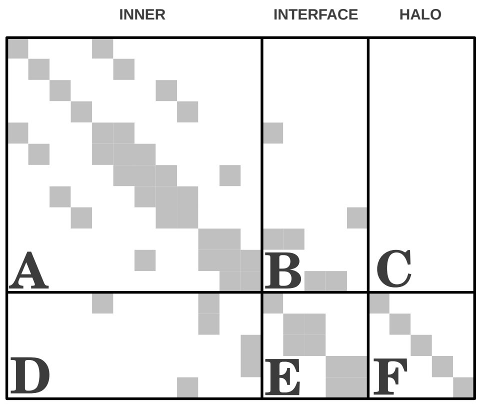


Figure 1.2: Block structure of the part of the system matrix owned by each MPI process: Block A represents couplings between inner unknowns; B between inner and interface unknowns; C is empty; $D = B^T$ between interface and inner unknowns; E between interface unknowns; and F between interface and halo unknowns.

1.4.2 The CUDA programming model for GPUs

A GPU device consist of a set of streaming multiprocessors (SMs) that employ a Single Instruction Multiple Threads (SIMT) model to create, schedule, manage and execute hundreds of threads during the execution of a program. The threads are grouped into warps, each thread of a warp runs the same instructions concurrently on a SM. A CUDA program is composed of a CPU code, so called host code, that launches tasks on a GPU device; and a GPU code, so called kernel code, that runs on the device. The kernel functions are executed on the GPU by a large set of parallel threads.

GPU devices have no common RAM memory space with CPUs nor with each other. Therefore, data transfer operations via PCI-Express are required to communicate between the devices. Nvidia devices of compute capability 1.1 and higher can overlap the copies between GPU and CPU with kernel execution. Concurrent kernels can be executed as well, the concurrency is managed through streams. A stream is defined as a sequence of commands executed in order. Then multiple streams, executed in parallel, are used to achieve the concurrency. Further details about the GPU architecture and the CUDA programming model can be found in the official documentation [1].

An in-house optimized implementation of the sparse matrix-vector multiplication, based on [28], is considered further (see Section 1.5.2). Additionally, the following kernels from the CUBLAS and CUSPARSE [29] math libraries have been used:

- sparse matrix-vector multiplication : `cusparseDcsrmv()`.
- dot product: `cublasDdot()`.
- vector operation $y = \alpha x + y$: `cublasDaxpy()`.
- vector operation $y = \alpha y$: `cublasDscal()`.

1.4.3 Two-level partitioning for hybrid CPU-GPU computing nodes

In our application context, GPUs are only used to accelerate the linear Poisson solver, which is the dominant part of our CFD algorithm (representing around 80% of the computing time). The rest of the code is based on a complex object-oriented CFD platform for unstructured meshes, which is hard to be exported with good performance to the GPU-model.

Modern hybrid computing nodes are generally composed of a couple of multicore processors (usually bringing 8∼16 CPU-cores) together with 2∼4 GPU devices. Normally CPU-cores outnumber GPU devices. In our approach, the total MPI group of processes consists of as many processes as CPU-cores in the computing nodes engaged. MPI processes are thus divided into local subgroups that share the same GPU device, which is only accessed by a master process.

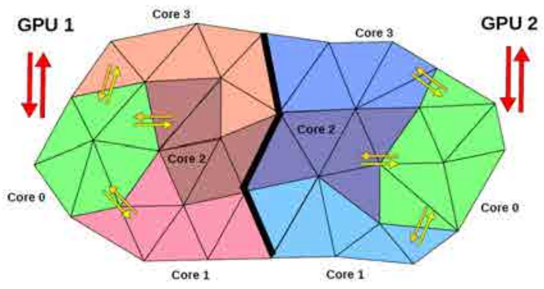


Figure 1.3: Two-level partitioning approach with CPU and GPU processing modes.

Therefore, a multilevel approach is necessary to manage the data flow between the pure-CPU and hybrid CPU-GPU mode on the resolution of the linear system. The first level of the decomposition consists of as many subdomains as GPUs engaged. The second level is obtained by dividing the first level subdomains into as many parts as MPI processes associated to the respective GPU. As a result, the second level of the decomposition has the number of sudomains equal to the number of CPU-cores engaged, and it is used for the CPU-only parts of the code; while the first level is used to distribute work among GPUs.

Figure 1.3 depicts the two level partitioning approach. In this case, the first level of decomposition divides the mesh into two subdomains linked with the GPUs. In the second level, each subdomain is decomposed into four and distributed through the CPU-cores. The subdomains linked with a master process are colored with green. Yellow arrows represent the data transfers between the workers and the master, while the red arrows symbolize the data transfers between master processes and GPUs.

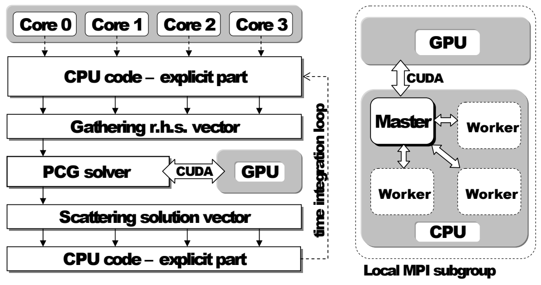


Figure 1.4: Two-level execution model with CPU and GPU processing modes.

In the execution model, explicit parts of the time step are carried out using the total MPI group (second level). Then, before starting the PCG solver execution, the worker processes of each local MPI subgroup send their parts of the right-hand-side (r.h.s.) vector to the master process (core 0), which gathers all the data, sends it to the GPU and runs the PCG on it. Finally, the other way round, the solution vector is scattered from the master to the workers, switching back to the CPU-only mode, in order to solve the next explicit parts of the time step. This gather/scatter overhead represents generally less than 1% of the solver time and can be neglected. Figure 1.4 represents the execution of the parallel CFD algorithm on a hybrid node with 4 CPU-cores per one GPU device. During the parallel execution of the PCG solver on multiple GPUs, communications are necessary for SpMV operations, in order to update halo elements of the first level partition, and for the evaluation of global values like residuals norms or dot products. This communication process, outlined in Figure 1.5, requires three steps: i) download data from the GPUs to the host memory; ii) transfer data between master MPI processes; iii) upload data back to the GPUs. The first and the third steps are limited by the bandwidth of the PCI-Express bus, while the second step is limited by the network bandwidth.

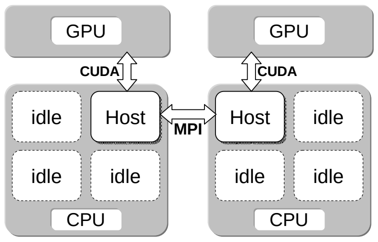


Figure 1.5: Communication Scheme.

1.5 Multi-GPU implementation of basic operations

In this section are described the parallel implementations for the two basic linear operations that require communications between GPUs, the dot product and the SpMV.

1.5.1 Dot product

The parallel dot product is performed in two stages: initially a local dot product is calculated by each active MPI process by means of the `cublasDdot()` function of the CUBLAS library, and then the global sum is collected by means of a reduction communication via the `MPI_Allreduce` function. This process is illustrated in Figure 1.6.

1.5.2 Sparse matrix-vector multiplication

The SpMV execution is dominated by the memory transactions instead of the arithmetic computations, thus it is considered a memory bounded problem. This trend is emphasized on the GPU executions, where loading data from global memory costs hundreds of clocks more than executing a multiplication-addition operation. Therefore, maximize the use of the bandwidth is one of the most important aspects of any implementation. When working with GPUs, it is also important to achieve a good load balance between the launched threads. The sparse storage format used has a major influence in both aspects. Different sparse matrix storage formats and a description of the corresponding kernels can be found in several works, for instance, in [28, 30–32].

Maximum performance achievable for the unstructured case

Discarding the reuse of data stored in cache memory, which is low for the unstructured case, the ratio between floating point operations (flop) and memory accesses

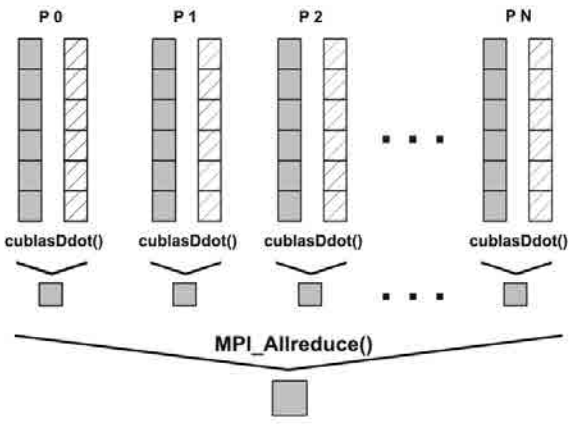


Figure 1.6: Hybrid MPI-CUDA Dot Product.

(in bytes) is lower than 1/10 for the SpMV operation: for each non-zero entry of the matrix there are two operations (one multiplication and one addition) and its necessary to read at least two 8-byte double values (the entry itself and the corresponding vector component) and one 4 byte integer (column index). Indeed, as shown further, sparse storage formats require reading some additional local indices of the data structures.

The Nvidia M2090 used in the numerical experiments of this paper, can achieve a maximum bandwidth of 165 GB/s and a peak performance of 650 GFlops. Considering the optimistic flop per byte ratio of 1/10, discarding cache effects, it is possible to estimate the maximum performance achievable for the SpMV kernel multiplying the flop per byte ratio by the bandwidth, leading it to a maximum expected performance of 16.4 GFlops. Unfortunately this represents only a 3% of the peak performance of the M2090 (measured under ideal conditions)

Formats

The present paper is devoted to the optimization of the overall parallel model but not the GPU calculations. Consequently, we are not introducing new storage formats, however, we have tested two of the existing formats in the numerical experiments. The first one is the common general-purpose CSR (Compressed Sparse Row) format. It consist of three arrays. The first two arrays store the column indices and the corresponding coefficient values for each non-zero entry of the matrix. The elements in these arrays are grouped by rows and sorted ascending the column index. The third array stores for each row the initial position of its elements in the first two arrays and, at the end, the total number of entries. Figure 1.7 (top) illustrates the CSR storage format on a sample matrix. The kernels of the CUSPARSE library, used in this paper as a reference for comparison, work with the CSR format.

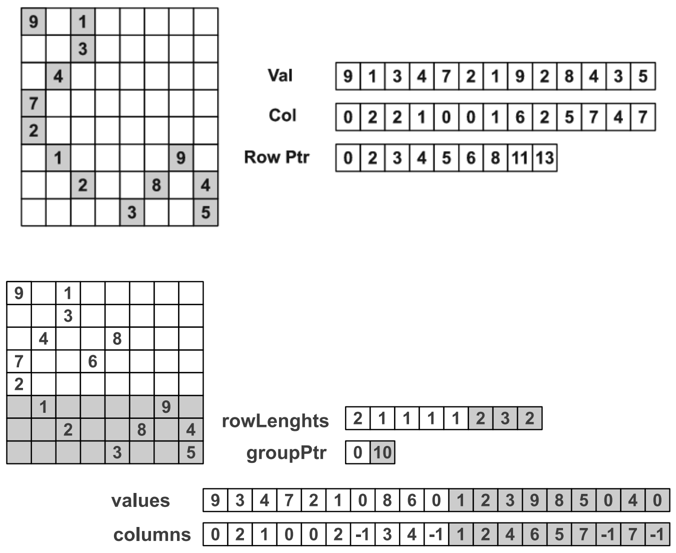

Val 9 1 3 4 7 2 1 9 2 8 4 3 5
Col 0 2 2 1 0 0 1 6 2 5 7 4 7
Row Ptr 0 2 3 4 5 6 8 11 13
rowLenghts 2 1 1 1 1 2 3 2
groupPtr 0 10
values 9 3 4 7 2 1 0 8 6 0 1 2 3 9 8 5 0 4 0
columns 0 2 1 0 0 2 -1 3 4 -1 1 2 4 6 5 7 -1 7 -1

Figure 1.7: Examples of sparse matrix storage formats CSR (top) and RGCSR (bottom).

The second format considered is a GPU-oriented Improved row-grouped CSR format (RgCSR) [28]. The main objectives of the RgCSR format are to keep the balance between threads and to ensure a coalesced data access. The algorithm to store the matrix consists in the following steps:

- Create group of rows, maintaining, as equal as possible, the number of non-zero entries per row.
- Divide each row into equally sized sub-groups of non-zero elements, called chunks. A thread is assigned to each chunk. The number of non-zero elements per chunk (chunk-size) is fixed for each group of rows.
- Zeros are padded when the number of non-zero elements per row is not divisible by the chunk-size, this will maintain the balance across threads.

- The vector of `values` and the vector of `columns` are stored following a chunk-wise order. The order is based in store the first element of each chunk of the group, after the second element of each chunk, and so on. This ordering reinforce coalesced memory reads since the executing threads will access contiguous memory adresses.
- The vector `rowLengths` stores the number of entries per each row, and the vector `groupPtr` indicates the starting position of each group.

An example of RgCSR matrix format is shown on Figure 1.7 (bottom). The matrix has been divided in two groups of 5 and 3 rows respectively. In the first group the chunk-size is 2 and 4 zeros have been padded, while in the second group the chunk-size is 3 and 2 zeros have been padded.

For more details about the format the reader is referred to [28]. A kernel to execute the SpMV using the RgCSR format was implemented following the specifications of the authors.

Non-overlapped SpMV

The sparse matrix-vector product operation, $\mathbf{w} = \mathbf{A} \cdot \mathbf{v}$ for general sparse matrices, is under study. Since this operation is performed on multiple GPUs using a domain decomposition approach (see Section 1.4.3), each GPU only evaluates the components $\mathbf{w}_{owned}$ associated to its owned elements. Recalling that halo elements are the neighboring external elements coupled with owned elements by entries of the matrix, it is necessary to update the halo components of the vector $\mathbf{v}$ before performing the SpMV operation. With the standard non-overlapped SpMV approach, referred here as NSpMV, this update is performed by means of a communication procedure previous to the calculations (see Algorithm 1). Obviously, this communication affects negatively the parallel performance.

Algorithm 1 NSpMV

```
1: Download $\mathbf{v}_{interface}$ from GPU to *host* CPU
2: Update $\mathbf{v}_{halo}$ (MPI point-to-point communication)
3: Upload $\mathbf{v}_{halo}$ from *host* CPU to GPU
4: Compute GPU kernel $\mathbf{w}_{owned} = \mathbf{A}_{owned} * \mathbf{v}_{local}$
```

Note that the ordering used (defined in Section 1.4.1), in which the elements of the sub-sets *inner*, *interface* and *halo*, are grouped, facilitates the data exchange between devices.

Overlapped SpMV

In order to perform the Overlapped SpMV (OSpMV), the matrix $\mathbf{A}_{owned}$ is split into its inner and interface parts, $\mathbf{A}_{owned} = \mathbf{A}_{inner} + \mathbf{A}_{interface}$. An example of this partition is represented in Figure 1.8. The overlapped SpMV is an optimization of Al-

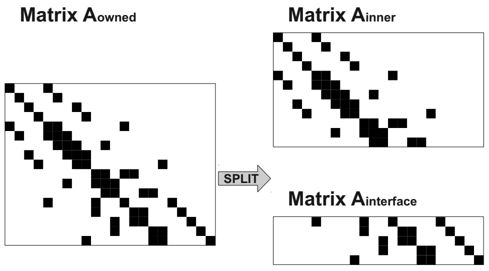


Figure 1.8: Splitting of the matrix

gorithm 1 in which the communications required in order to update v_{halo} are performed simultaneously with the evaluation of the inner components of the product, $\mathbf{w}_{inner} = \mathbf{A}_{inner} * \mathbf{v}_{owned}$. Recalling that the inner elements are those owned elements that are coupled only with other owned elements (see Section 1.4.1), they can be processed independently of the halo update operation. In order to overlap the two processes, calculation of inner elements and halo update, two simultaneous streams are launched in a concurrent execution model. These streams are then synchronized and finally the interface components of the product can be evaluated. Normally $\mathbf{A}_{interface}$ is much smaller than $\mathbf{A}_{inner}$ since it only represents the couplings of the nodes at the border of the sub-domain. The overall process is outlined in Figure 1.9. As a result of this optimization, the time spend on the halo update operation is hidden (at least partially) behind the calculation of inner elements.

1.6 Numerical Experiments

The TGCC Curie supercomputer [33], member of the pan-European computing and data management infrastructure PRACE, was used for numerical tests. Its hybrid nodes, with two Intel Xeon E5640 (Westmere) 2.66 GHz quad-core CPUs and two Nvidia M2090 GPUs each, are interconnected via an Infiniband QDR network. Version 4.0 of CUDA was used to implement the GPU kernels.

The finite-volume discretization of the Poisson equation was performed on unstructured tetrahedral meshes over a 3D spherical domain. The discretization of this equation is detailed in Section 1.3.

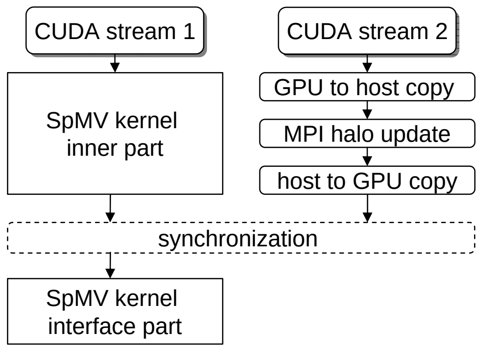


Figure 1.9: Concurrent Execution Model

The number of nodes used in the tests ranges from 1 to 4, the size of the mesh was varied in order to keep a fixed amount of CV per process. For all the meshes the matrix resulting from the discretization has in average 4.9 entries per row, being therefore tremendously sparse.

When running the MPI-CUDA PCG on multiple nodes the parameters used in setup of the AIP were: i) sparsity level of the preconditioner is fixed $\mathcal{S} = 1$, ii) and no elements of the approximated matrix are dropped.

1.6.1 Distribution of execution time

It is well known that the cost of the CG is dominated by the SpMV operation [34–36]. This is is exemplified in Figure 1.10 for the solution of the discrete Poisson equation on a mesh over a spherical domain with $400,000$ control volumes (CV) on a single CPU. The CG is preconditioned with the AIP and the distribution of time is shown for different levels of the AIP sparsity pattern. Recalling that the solution stage of the AIP consists of two SpMV operations (see Section 1.2), the total cost of the SpMV operation in the overall algorithm can be obtained summing the cost of preconditioner with the cost of the other product required in the CG (the two lower parts in each column of the figure). Both parts take at least 78% of the total solution time. As expected, the percentage grows (up to 90%) when increasing the power of $\mathbf{A}$ used to generate the sparsity pattern, because the preconditioner becomes denser. With this scenario in mind, we have focused in the optimization of the MPI-CUDA implementation of the SpMV operation.

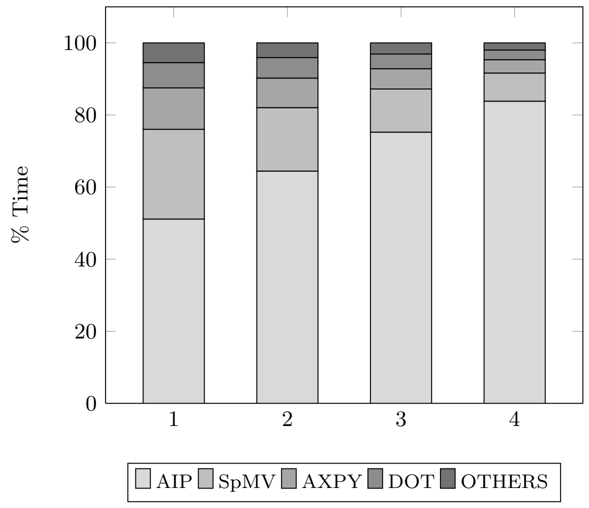


Figure 1.10: Distribution of computing time across PCG operations.

1.6.2 Single GPU vs. single CPU-core comparison

Since no MPI communications are needed during the algorithm execution on a single GPU, performance against a single CPU-core is directly compared. Two different storage formats are used for the sparse matrices: the CSR and the RgCSR.

The tests were executed on meshes ranging from 100,000 to 400,000 control volumes and the performance was analyzed for the SpMV, for the AIP with sparsity level 1 and for the whole PCG iteration. The resulting execution times and the achieved net performance are shown in the Table 1.1.

In all the cases the GPU outperforms the CPU execution by a factor ranging from 6 to 12 times. These results are consistent with those reported in the works [4, 6]. The comparison of the speedup with respect to the one CPU core achieved with both formats is represented in Figure 1.11. The kernels based in RgCSR format execute up to 5%, 40% and 8% faster in average than those based in CSR format for the SpMV, AIP and CG respectively.

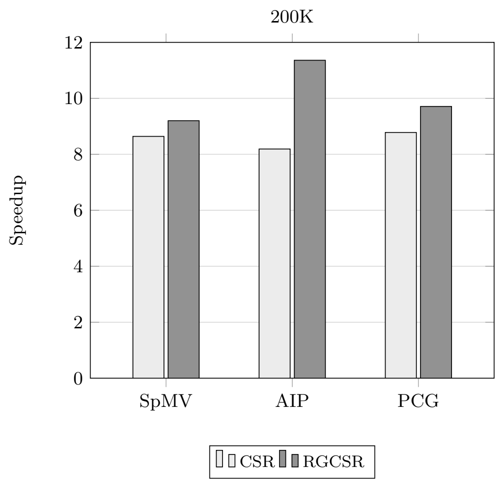
Considering that we are working with unstructured sparse matrices with an average of 4.9 entries per row and using the double-precision floating-point format, the net performance achieved is in the range of the results reported in [31,37] for similar matrices. The characteristics of our matrices make the SpMV operation take only around 1% of the peek performance of the Nvidia M2090. This is the cruel reality for such a memory-bounded operations with unstructured memory access and such a low flop per byte ratio.

Mesh		CPU core		GPU		GPU	
		CSR		CUSPARSE		RgCSR	
size	Operation	Time	Gflops	Time	Gflops	Time	Gflops
100K	SpMV	1.10	0.87	0.18	5.3	0.17	5.49
	AIP	2.31	0.5	0.37	3.1	0.24	4.83
	PCG	6.09	0.52	0.88	3.59	0.79	4.02
200K	SpMV	2.67	0.73	0.31	6.3	0.29	6.52
	AIP	5.00	0.48	0.61	3.9	0.44	5.35
	PCG	10.1	0.63	1.15	5.55	1.04	6.11
300K	SpMV	4.42	0.66	0.44	6.66	0.42	6.92
	AIP	7.66	0.46	0.83	4.23	0.62	5.65
	PCG	16.9	0.56	1.72	5.53	1.59	5.99
400K	SpMV	6.64	0.58	0.55	7.0	0.53	7.34
	AIP	10.9	0.43	1.07	4.4	0.85	5.58
	PCG	24.8	0.51	2.35	5.43	2.23	5.73

Table 1.1: Comparison of the net performance in Gflops and Time in ms between single-CPU core and single-GPU implementations based on CUSPARSE (CSR) and RgCSR.

1.6.3 Multi-GPU SpMV. Overlapped vs. non-overlapped approach

The main objective of the tests shown in this section is to analyze the efficiency of the parallel SpMV schemes explained in Section 1.5.2. Recalling that each supercomputer node contains 8 CPU-cores and 2 GPUs, each MPI subgroup corresponding to a GPU consists of 4 MPI processes. Initially, we compare the storage formats here considered for both, the non-overlapped and the overlapped approaches, on different number of GPUs, keeping the load per GPU constant at $200,000$ CV. Figure 1.12 shows the speedup obtained by the RgCSR with respect to the CSR format for the SpMV of the laplacian matrix. This acceleration is negligible for the non-overlapped approach but reaches up to 40% for the overlapped one. This difference is produced by the better performance of the RgCSR model on the multiplication of the extremely sparse interface part of the matrix for the overlapped approach, because of the regularity obtained by padding zeros. Being consequent with these results and the ones obtained in the single GPU tests, we have chosen the RgCSR kernels for further experiments. The achieved net performance and execution times for the OSpMV and NSpMV are compared in Table 1.2 for different work loads and number of GPUs. As expected, the overlapped version outperforms in each case the non-overlapped SpMV because it minimizes the communication overhead (see Section 1.5.2). This improvement ranges between 12% and 44% of the execution time.



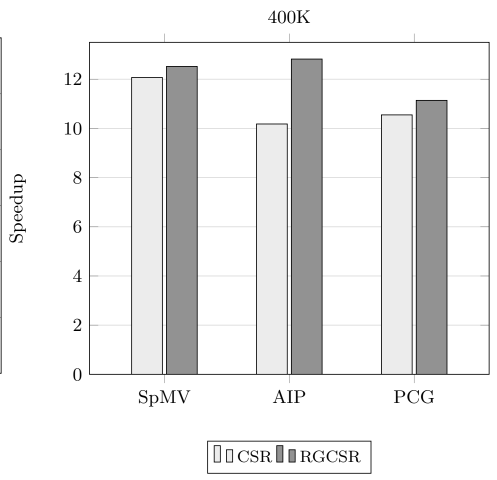


Figure 1.11: Speedup of the single GPU with respect to the single CPU execution of the SpMV, AIP and PCG iteration using the CSR (white) and RGCSR (grey) formats for mesh sizes 200, 000 (left) and 400, 000 (right).

Figure 1.13 shows exactly how much of the communication overhead can be hidden by the overlapped version. The non-overlapped SpMV time has been separated into computation and communication parts and compared with the overall execution time of the overlapped SpMV. Despite there is a certain overhead in the decomposition of SpMV into two kernels, which mainly comes from the kernel execution latency, around 50% of communication costs are hidden.

1.6.4 Parallel performance. Multi-GPU vs. CPU-only version.

The RgCSR-based multi-GPU implementations of the SpMV, the AIP and the whole PCG solver are compared with the original CPU-only versions based on the CSR format. The comparison is carried out for 32 CPU-cores versus 8 GPUs on a mesh with 400, 000 CV per GPU (*i.e.* 100, 000 per CPU). Execution times and GFlops are shown in Table 1.3. The sparsity level of the AIP is fixed to 1 and, since the convergence of the PCG is not being considered, results are shown for only one PCG iteration. As expected, the multi-GPU version outperforms in every case the CPU-only implementation. The resulting overall speedup achieved for the PCG algorithm is around 3.3.

A weak speedup study of the PCG algorithm for the both CPU and GPU implementations is displayed in Figure 1.14. The load of 400, 000 CV per GPU (100, 000

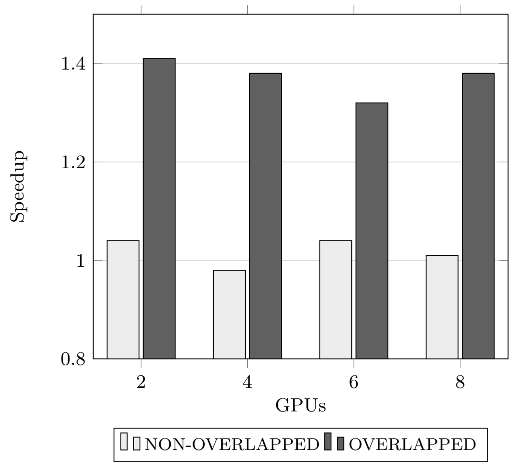


Figure 1.12: Speedup obtained by the RgCSR with respect to the CSR format, for the overlapped and non-overlapped multi-GPU SpMV on different number of GPUs.

Mesh size per GPU	SpMV	2 GPU		4 GPU		6 GPU		8 GPU	
		Time	Gflops	Time	Gflops	Time	Gflops	Time	Gflops
200K	N	0.54	7.24	0.64	12.27	0.67	17.82	0.68	22.91
200K	O	0.45	8.54	0.49	16.09	0.54	22.13	0.55	28.72
400K	N	0.94	8.40	1.08	14.63	1.22	19.46	1.17	27.01
400K	O	0.83	9.55	0.84	18.78	0.84	28.20	0.85	37.38
600K	N	1.31	9.04	1.48	16.04	1.55	22.99	1.59	29.92
600K	O	1.20	9.92	1.23	19.36	1.23	28.94	1.23	38.51

Table 1.2: Comparison of the net performance in Gflops and Time in ms, achieved with overlapped (O) and non-overlapped (N) SpMV implementations for different loads and number of GPUs.

		32 CPU-cores		8 GPU	
Mesh	Operation	Time, ms	Gflops	Time, ms	Gflops
400K	SpMV	3.15	10.10	0.85	37.38
	AIP	4.83	9.86	1.68	28.30
	PCG	11.64	10.94	3.49	36.50

Table 1.3: Comparison of multi-GPU and CPU-only implementations using 4 computing nodes (32 CPU cores vs. 8 GPUs).

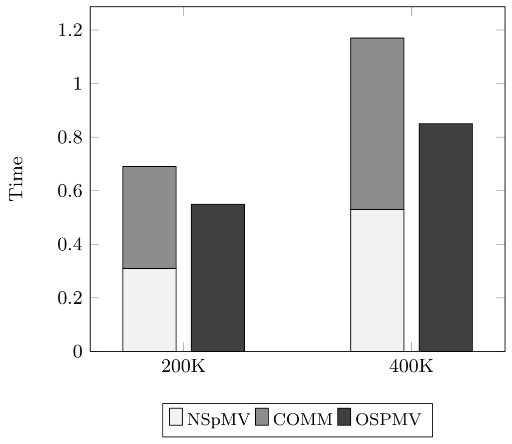


Figure 1.13: Communication overlapping diagram for the load per GPU 200,000 and 400,000 control volumes.

CV per CPU) is kept constant. The plot shows nearly ideal scaling with both implementations and the difference between them keeps constant at growing the number of GPUs/CPUs involved.

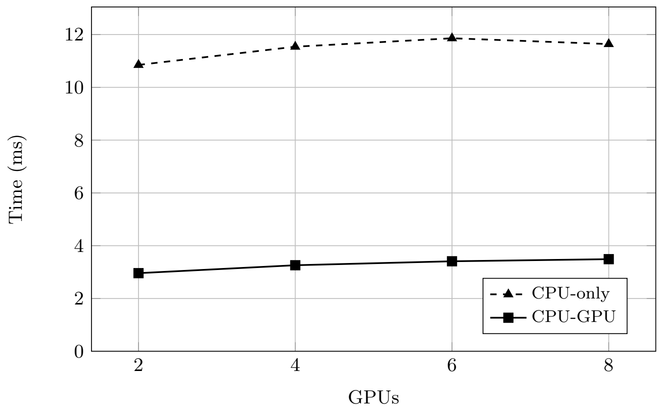


Figure 1.14: Weak speedup for one PCG iteration with 400000 CV per GPU (100000 CV per CPU).

Finally, a comparison is performed for different loads in order to see its influence on the resulting speedup. Figure 1.15 shows the speedup of the multi-GPU vs. the

CPU-only version for 200, 000, 400, 000 and 600, 000 CV per GPU (50, 000, 100, 000 and 150, 000 CV per CPU). The benefits of the multi-GPU implementation increase with the mesh size since the ratio between the GFlops obtained with both devices increases (this trend is also observed in Table 1.1 for other mesh sizes). In this case a maximum speedup of around 4.4× is reached for the largest load on 4, 6 and 8 GPUs.

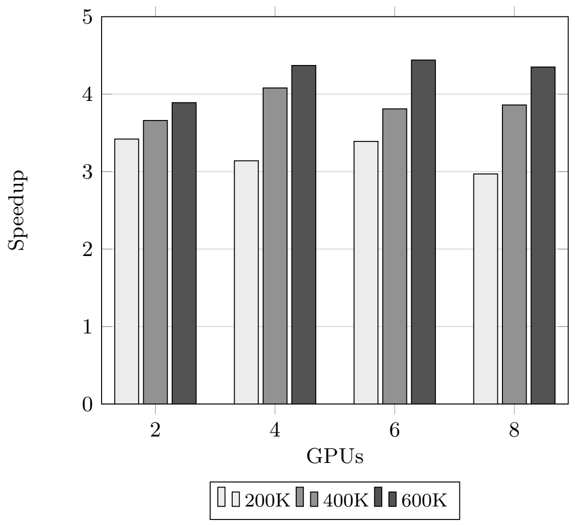


Figure 1.15: Speedup of the multi-GPU PCG compared to the CPU-only MPI version for different mesh sizes and numbers of GPUs (corresponding number of CPU cores is 4 times the number of GPUs).

Recalling that in the present approach the PCG solver is the only part of the overall CFD algorithm accelerated via GPU coprocessors, and considering that in the CPU-only version takes around 80% of the time-step cost, the overall acceleration would be up to 2.6× (and 2.4× in average over different mesh sizes).

1.7 Concluding Remarks

A hybrid MPI-CUDA implementation of the approximate inverse preconditioned conjugate gradient method, has been presented. We have mainly focused in the SpMV operation because it dominates the computing costs. The main characteristic of the hybrid SpMV developed is its overlapping strategy that allows to hide part of the data transfer overhead produced by the device-to-host and MPI communications behind calculations on GPUs.

The context of application has been the resolution of the Poisson equation within the numerical process for the resolution the Navier Stokes equations in the simulation of incompressible flows. Since only the Poisson solver is ported to the GPU coprocessors, a two level domain decomposition has been proposed in order manage the data flow between the CPU-only and CPU-GPU parallelization modes.

Performance tests, engaging different numbers of hybrid nodes and unstructured meshes of different sizes, have demonstrated speedups of the hybrid PCG solver of around 3.7× compared to the CPU-only solver. The corresponding speedup of the overall CFD algorithm replacing the linear solver by the new version would be in average 2.4×.

Moreover, it has been demonstrated that the performance of general purpose sparse libraries such as CUSPARSE can be improved by an in-house SpMV kernels implemented considering specific information of the matrix. This performance is directly linked with the sparse matrix storage format being used, therefore, as future work, new storage formats may be tested or developed in order to optimize the GPU computations. Finally, other parts of the overall CFD code may also be ported to the hybrid mode in order to better use the potential of GPU devices.

References

[1] Nvidia-Corporation. Nvidia CUDA C Programming Guide, 2007.

[2] Khronos Group. Opencl 1.2 Specification. Webpage: http://www.khronos.org/opencl/, 2009.

[3] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. In *Micro, IEEE*, volume 28, pages 13–27, 2008.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schroøder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM Trans. Graph.*, volume 22, pages 917–924, 2003.

[5] R. Li and Y. Saad. Gpu-accelerated Preconditioned Iterative Linear Solvers. *The journal of supercomputing*, 63:443–466, 2013.

[6] D. Michels. Sparse-matrix-cg-solver in cuda. In *The 15th Central European Seminar on Computer Graphics*, 2011.

[7] A. Cevahir, A. Nukada, and S. Matsuoka. Fast conjugate gradients with multiple gpus. In *Computational Science, ICCS 2009 of Lecture Notes in Computer Science*, volume 5544, pages 893–903, 2009.

[8] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. In *18th Euromicro International Conference on: Parallel, Distributed and Network-Based Processing (PDP)*, volume 5544, pages 583–592, 2010.

[9] Y. Saad. *Iteratives Methods for Sparse Linear Systems*. SIAM, 2003.

[10] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal of Scientific Computing*, 19:995–1023, 1998.

[11] M. Benzi, C.D. Meyer, and M. Tuma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal of Scientific Computing*, 17:1135–1149, 1996.

[12] T. Huckle and M. Grote. A new approach to parallel preconditioning with sparse approximate inverses. Tech Report Stanford University, 1994.

[13] E. Chow and Y. Saad. Approximate inverse preconditioners for general sparse matrices. In *Proc. of the Colorado Conference on Iteratives Methods*, 1994.

[14] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[15] M. Kremenetsky, J. Richardson, and H.D. Simon. Parallel preconditioning for cfd problems on the cm-5. In *Parallel Computational Fluid Dynamics: New Trends and Advances*, pages 401–408, North Holland, 1995.

[16] Y. Saad. Preconditioned krylov subspace methods for cfd applications. In *Proceedings of the International Workshop on Solution Techniques for Large-Scale CFD Problems*, pages 179–195. Wiley, 1995.

[17] V. Deshpande, M. Grote, P. Messmer, and W. Sawyer. Parallel implementation of a sparse approximate inverse preconditioner. In *Lecture Notes in Computer Science, Parallel Algorithms for Irregularly Structured Problems*, pages 63–74, 1996.

[18] M. Wagner, K. Rupp, and J. Weinbub. A comparison of algebraic multigrid preconditioners using graphics processing units and multi-core central processing units. In *2012 Symposium on High Performance Computing (HPC '12)*, number 12, page 8, 2012.

[19] N. Bell, S. Dalton, and L.N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. Nvidia Technical Report, 2011.

[20] O. Lehmkuhl, C.D. Perez-Segarra, R. Borrell, M. Soria, and A. Oliva. Termofluids: A new parallel unstructured cfd code for the simulation of turbulent industrial problems on low cost pc clusters. In *Parallel Computational Fluid Dynamics*, number 12, page 8. Elsevier, 2007.

[21] R. W. C. P. Verstappen and A. E. P. Veldman. Symmetry-preserving discretization of turbulent flow. *Journal of Computational Physics*, 187:343–368, 2003.

[22] F. X. Trias and O. Lehmkuhl. A self-adaptive strategy for the time-integration of navier-stokes equations. *Numerical Heat Transfer, part B*, 60(2):116–134, 2011.

[23] A. J. Chorin. Numerical solution of the navier-stokes equations. *Mathematics of Computation*, 22:745–762, 1968.

[24] O. Lehmkuhl, R. Borrell, I. Rodríguez, C.D. Pérez-Segarra, and A. Oliva. Assessment of the symmetry-preserving regularization model on complex flows using unstructured grids. *Journal of Computers and Fluids*, 60:108–116, 2012.

[25] G. Colomer, R. Borrell, F. X. Trias, and I. Rodríguez. Parallel algorithms for sn transport sweeps on unstructured meshes. *Journal of Computational Physics*, 232:118–135, 2013.

[26] R. Borrell, O. Lehmkuhl, F. X. Trias, and A. Oliva. Parallel direct poisson solver for discretisations with one fourier diagonalisable direction. *Journal of Computational Physics*, 230:4723–4741, 2011.

[27] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20:359–392, 1998.

[28] M. Heller and T. Oberhuber. Improved row-grouped csr format for storing of sparse matrices on gpu. In *Proceedings of Algoritmy 2012*, pages 282–290. Handlovicová A., Minarechová Z. and Sevcovic D. (ed.), 2012.

[29] Nvidia-Corporation. Nvidia cusparse and cublas libraries, 2007.

[30] T. Oberhuber, A. Suzuki, and J. Vacata. New row-grouped csr format for storing the sparse matrices on gpu with implementation in cuda. In *Acta Technica*, volume 56, pages 447–446, 2011.

[31] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Nvidia Technical Report NVR-2008-004, 2008.

[32] A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on nvidia gpus. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5657, pages 289–297. K. Bertels, N. Dimopoulos, C. Silvano, S. Wong (Eds.), 2009.

[33] Tgcc curie supercomputer. Webpage: http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm, 2011.

[34] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., 1994.

[35] F.N. Sibai and H.K. Kidwai. Implementation and performance analysis of parallel conjugate gradient on the cell broadband engine. *IBM Journal of Research and Development*, 54(10):1–12, 2010.

[36] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, 1991.

[37] K. Matam and K. Kothapalli. Accelerating sparse matrix vector multiplication in iterative methods using gpu. In *International Conference on Parallel Processing (ICPP)*, pages 612–621, 2011.

Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers

Main contents of this chapter have been submitted in:

G.Oyarzun, R. Borrell, A. Gorobets and A. Oliva. Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers. *SIAM Journal on Scientific Computing* 2015. In review.

Abstract.
Nowadays supercomputers are intensively adopting new technologies aiming to overcome power wall constraints. This brings new computing architectures, programming models and, eventually, the requirement of rewriting codes and rethinking algorithms and mathematical models. It is a disruptive moment with a variety of novel architectures and frameworks, without any clarity of which one is going to prevail. In this context, the portability of codes across different architectures is of major importance. This paper presents a portable implementation model for DNS and LES simulation of incompressible turbulent flows, based on an algebraic operational approach. A strategy to decompose the non-linear operators into two SpMV is proposed, without loosing any precision in the process. As a result, at the time-integration phase the code relies only on three algebraic kernels: sparse-matrix-vector product (SpMV), linear combination of two vectors (AXPY) and dot product (DOT), providing high modularity and, consequently, the desired portability. Additionally, a careful analysis of the implementation of this strategy for hybrid CPU/GPU supercomputers has been carried out. Special attention has been paid to rely on data structures that fit the stream processing model. Moreover, a detailed performance analysis is provided by tests engaging up to 128 GPUs. The objective consists in understanding the challenges of implementing CFD codes on GPUs and forming some fundamental rules to achieve the maximal possible performance. Finally, we propose an accurate and useful method to estimate the performance of the overall time-integration phase based only on separate measurements of the three main algebraic kernels.

2.1 Introduction

The pursuit of exascale has driven the adoption of new parallel models and architectures into HPC systems in order to bypass the power limitations of the multi-core CPUs. The last prominent trend has been the introduction of massively-parallel accelerators used as math co-processors to increase the throughput and the FLOPS per watt ratio of the HPC systems. Such devices exploit a stream processing paradigm that is closely related to single instruction multiple data (SIMD) parallelism. A multilevel parallelization that combines different kinds of parallelism is required for hybrid systems with accelerators. In any case the top level needs multiple instruction multiple data (MIMD) distributed memory parallelization in order to couple nodes of a supercomputer. At this level there is a sufficient clarity with the programming frameworks and algorithms: the message passing interface (MPI) standard is most commonly used for distributed memory parallelization based on a geometric domain decomposition. At the bottom level a SIMD-based parallelism is needed: either for SIMD extensions via vector registers (like in CPUs) or for streaming multiprocessors of GPUs.

The problem that the CFD community has to face is the variety of competing architectures and frameworks without any clarity of which one is going to prevail. The key point is that if an algorithm naturally fits stream processing, the most restrictive paradigm at the bottom level, then it will work well on upper levels (shared and distributed memory MIMD parallelization) and, it will work well on GPUs, MICs, and CPUs. This leads to a conclusion that a fully-portable algorithm must be composed only of operations that are SIMD-compatible.

Taking into account this diversity of frameworks and architectures, and the increasing complexity of programming models, the idea proposed in this paper is that the algorithm must: 1) only consist of operations compatible with stream processing (portability); 2) rely as much as possible on a minimal set of common linear algebra operations with standard interfaces (modularity). This maximizes the independence of the implementation from a particular computing framework.

The algorithm for modeling of incompressible turbulent flows on unstructured meshes presented in this paper is based on only three linear operators: 1) the sparse matrix-vector product (SpMV); 2) the dot product; 3) the linear combination of vectors $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$ (referred as AXPY in the BLAS standard nomenclature). This way, the problem of porting the code is reduced just to switching between existing implementations of these operations. The non-linear operators are decomposed into two SpMV operations, avoiding the introduction of new operators. As a result, the SpMV takes more than 70% of the total execution time, so it is the dominant kernel.

Then, our portable approach has been implemented for both hybrid supercomputers with GPU co-processors and standard multi-core systems. This has allowed us to carry out a detailed comparative performance analysis. This study aims at understanding the challenge of running CFD simulations on hybrid CPU/GPU supercomputers, and forming some fundamental rules to attain the maximal achievable performance. The Minotauro supercomputer of the Barcelona Supercomputing Center (Intel Westmere E5649 CPUs and NVIDIA Tesla M2090 GPUs) has been used for performance tests. Consequently, the CUDA platform was chosen for implementing the main algebraic kernels. The cuSPARSE library for CUDA provides the necessary linear algebra operations. However, an in-house SpMV implementation optimized for our specific sparsity patterns has been developed. It incorporates asynchronous communications with overlapping of communications and computations on the accelerators. Scalability tests engaging up to 128 GPUs were performed and the results have been directly compared with executions on the same number of 6-cores CPUs. The effects on the performance of the cache usage efficiency on CPUs and the multiprocessor occupancy on GPUs are analyzed in detail. Finally, it is demonstrated that the overall performance of our CFD algorithm can be precisely estimated by analyzing only the parallel performance of the three basic kernels individually.

Regarding the related work, an early attempt of using GPU for CFD simulations is described in [1], where a CUDA implementation of a 3D finite-difference algorithm based on high-order schemes for structures meshes was proposed. Other examples of simulations on GPU using structured meshes can be found in [2, 3]. However, these works focused only on a single-GPU implementation and rely on stencil data structures for structured meshes. OpenCL implementations of such a class of algorithms for single GPU can be found for instance in [4], where a novel finite element implementation based on edge stencils is presented, and in [5], where a set of basic CFD operators based on high order schemes is studied.

Within the class of multi-GPU CFD implementations, a successful example of a high-order finite difference and level set model for simulating two-phase flows can be found in [6]. In addition, a Hybrid MPI-OpenMP-CUDA fully 3D solver is presented in [7]. Both implementations are restricted to structured grids. Some efforts for porting unstructured CFD codes to multi-GPU were conceived by porting only the most computational intensive parts of the algorithm (Poisson equation), as explained in [8, 9]. Although, this methodology fails to attain the maximum of GPU potential because of Amdahl's law limitations. Finally, DNS simulation using unstructured meshes and multi-GPU platforms were shown in [10, 11]. The strategy adopted there was based on a vertex-centered finite volume approach including a mixed precision algorithm. Nevertheless, in all the mentioned examples the overall implementation seem to be tightly coupled with the framework it relies upon . Therefore, portability of those codes requires a complex procedure and large programming efforts. In contrast, the present paper focuses on fully-portable implementation of a CFD algorithm on unstructured meshes that can run on multiple accelerators without restrictions to a particular architecture.

The rest of the paper is organized as follows: Section 2.2 describes the math background for the numerical simulation of incompressible turbulent flows; in Section 2.3 the portable implementation model based on algebraic operations is described; Section 2.4 focuses on its implementation on hybrid systems engaging both CPU and GPU devices; numerical experiments on the Minotauro supercomputer of the Barcelona Supercomputing Center are shown in Section 2.5; and, finally, concluding remarks are stated in Section 2.6.

2.2 Math model and numerical method

The simulation of a turbulent flow of an incompressible Newtonian fluid is considered. The flow field is governed by the incompressible Navier-Stokes equations written as:

$$\nabla \cdot \mathbf{u} = 0, \tag{2.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \Delta \mathbf{u} \tag{2.2}$$

where $\mathbf{u}$ is the three-dimensional velocity vector, p is the kinematic pressure scalar field and ν is the kinematic viscosity of the fluid.

In an operator-based formulation, the finite-volume spatial discretization of these equations reads

$$\mathbf{M}\mathbf{u}_c = \mathbf{0}_c, \tag{2.3}$$

$$\boldsymbol{\Omega}\frac{d\mathbf{u}_c}{dt} + \mathbf{C}(u_s)\mathbf{u}_c + \mathbf{D}\mathbf{u}_c + \boldsymbol{\Omega}\mathbf{G}p_c = \mathbf{0}_c, \tag{2.4}$$

where $\mathbf{u}_c$ and p_c are the cell-centered velocity and pressure fields, u_s is the velocity field projected to the faces' normals, $\boldsymbol{\Omega}$ is a diagonal matrix with the sizes of control volumes on the diagonal, $\mathbf{C}(u_s)$ and $\mathbf{D}$ are the convection and diffusion operators, and finally, $\mathbf{M}$ and $\mathbf{G}$ are the divergence and gradient operators, respectively. In this paper, a second order symmetry-preserving and energy conserving discretization is adopted [12]: the convective operator is skew symmetric, $\mathbf{C}(\mathbf{u}_c) + \mathbf{C}(\mathbf{u}_c)^* = 0$, the diffusive operator is symmetric positive-definite and the integral of the gradient operator is minus the adjoint of the divergence operator, $\Omega\mathbf{G} = -\mathbf{M}^*$. Preserving the (skew-) symmetries of the continuous differential operators has shown to be a very suitable approach for accurate numerical simulations [13–15]. For the temporal discretization, a second order explicit Adams-Bashforth scheme is used, and the fully-discretized problem reads

$$\boldsymbol{\Omega}\frac{\mathbf{u}_c^{n+1} - \mathbf{u}_c^n}{\Delta t} = \mathbf{R}\left(\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}\right) + \mathbf{M}^* p_c^{n+1}, \tag{2.5}$$

$$\mathbf{M}\mathbf{u}_c^{n+1} = \mathbf{0}_c, \tag{2.6}$$

where $\mathbf{R}(\mathbf{u}_c) = -\mathbf{C}(u_s)\mathbf{u}_c - \mathbf{D}\mathbf{u}_c$. The pressure-velocity coupling is solved by means of a classical fractional step projection method [16]. In short, reordering Eq. 2.5 is obtained the next expression for $\mathbf{u}_c^{n+1}$

$$\mathbf{u}_c^{n+1} = \mathbf{u}_c^n + \Delta t\boldsymbol{\Omega}^{-1}\left(\mathbf{R}\left(\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}\right) + \mathbf{M}^* p_c^{n+1}\right). \tag{2.7}$$

Then, substituting this into Eq. 2.6, leads to a Poisson equation for p_c^{n+1},

$$-\mathbf{M}\boldsymbol{\Omega}^{-1}\mathbf{M}^* p_c^{n+1} = \mathbf{M}\left(\frac{\mathbf{u}_c^n}{\Delta t} + \boldsymbol{\Omega}^{-1}\mathbf{R}\left(\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}\right)\right), \tag{2.8}$$

which must be solved once per time-step. The left hand side of (2.8) is the discrete Laplace operator, $\mathbf{L} = -\mathbf{M}\boldsymbol{\Omega}^{-1}\mathbf{M}^*$, which is symmetric and negative definite.

At each time step, the non-linear convective operator is re-evaluated according to the velocity at the faces of the control volumes, $\mathbf{C}(u_s)$. In our collocated scheme the evaluation of the velocity at the faces is based on [17]. Two additional operators are required: $\mathbf{\Gamma}_{c\rightarrow s}$ to project a cell-centered vector field to the faces' normals; and $\mathbf{G}_s$, to evaluate the gradient of a face-centered scalar field. The evaluation of u_s reads:

$$u_s = \mathbf{\Gamma}_{c\rightarrow s}\mathbf{u}_c^{n+1} - \Delta t \left(\mathbf{G}_s p_c^{n+1} - \mathbf{\Gamma}_{c\rightarrow s}\mathbf{g}_c^{n+1} \right) \tag{2.9}$$

where $\mathbf{g}_c^{n+1}$ is the cell-centered pressure gradient field.

In addition, when the LES model is activated, the viscosity at the faces ν_s needs to be updated at each time-step according to the turbulence eddie viscosity at faces ν_{ts}. As a result, the diffusive term becomes a non-linear operator that also needs to be re-evaluated at each time-step as $\mathbf{D}(\nu_s)$. The computation of $\nu_{ts} = \mathbf{K}(\mathbf{u}_c)$ requires the calculation of the velocity gradients to construct the tensor operators, and, depending of the LES model, perfom certain tensor operations. Various modern LES models fit this approach, including Smagorinsky, WALE, QR, Sigma, S3PQR (for details about models see, for instance, [18] and references therein). Further details on this integration method and some options for the definition of the discrete operators can be found in [12,17]. The overall time-step algorithm is outlined in Algorithm 1.

Algorithm 1 Time integration step

```
1: Evaluate the predictor velocities: $\mathbf{u}_c^p = \mathbf{u}_c^n + \Delta t \Omega^{-1} \left( \mathbf{R}(\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}) \right)$
   ($\mathbf{R}$ evaluated according to $u_s$ and $\nu_s$)
2: Solve the Poisson equation: $\mathbf{L}p_c^{n+1} = \frac{1}{\Delta t}\left(\mathbf{M}\mathbf{u}_c^p\right)$
3: Correct the centered velocities: $\mathbf{g}_c^{n+1} = \mathbf{G}p_c^{n+1}, \quad \mathbf{u}_c^{n+1} = \mathbf{u}_c^p - \Delta t(\mathbf{g}_c^{n+1})$
4: Calculate the velocities at the faces: $u_s = \mathbf{\Gamma}_{c\rightarrow s}\mathbf{u}_c^{n+1} - \Delta t \left( \mathbf{G}_s p_c^{n+1} - \mathbf{\Gamma}_{c\rightarrow s}\mathbf{g}_c^{n+1} \right)$
5: Calculate the eddie viscosity (if LES used): ${\nu_t}_s^{n+1} = \mathbf{K}(\mathbf{u}_c^{n+1}),\ \nu_s^{n+1} = \nu_s + {\nu_t}_s^{n+1}$
6: Calculate $\Delta t$ based on the CFL condition: $\Delta t = CFL(\mathbf{u}_c^{n+1})$
```

2.3 Operational algebraic implementation approach

Leaving aside the linear solver, the most common form of implementing a CFD code is by using stencil data structures. This is how it is arranged our CFD code, TermoFluids [19], that is an object oriented code written in C++. TermoFluids includes a user-friendly API to manage the basic discrete operations of the geometric discretization. This API is used on the pre-processing stage to generate stencil raw data structures, storing in a compact form the geometric information required by the numerical methods. On the time integration phase, where most of computing time is spent, most of computations are based on sweeping through the stencil arrays and operating on scalar fields that represent physical variables. Basing the operations on raw flattened stencil data structures, rather than on our higher level object-oriented intuitive API, optimizes the memory usage and increases the arithmetic intensity of the code resulting in higher performance. In this paper we present a new implementation approach, we have replaced stencil data structures and stencil sweeps by algebraic data structures (sparse matrices) and algebraic kernels (SpMV). The high-level abstractions based on object-oriented programming are the same but the data structures at the bottom layer, used to accelerate the time integration phase, are sparse matrices stored in compressed formats instead of stencil arrays. Both implementations are equivalent in terms of the physical results and very similar in terms of performance. However, with the algebraic approach a perfect modularity is achieved since the code mainly results on a concatenation of three algebraic kernels: the sparse-matrix vector product (SpMV), the linear combination of two vectors (denoted "AXPY" in the BLAS standard) and the dot product of two vectors (DOT). In the numerical experiments section is shown how these three kernels represent more than 95% of the computing time. As a result, the portability of the code becomes also straightforward since we need to focus only on three algebraic kernels. Moreover, note that these standard kernels are found in many linear algebra libraries, some of them optimized to perform well on specific architectures [20].

The linear operators that remain constant during all the simulation can be evaluated as a sparse-matrix vector product in a natural way. For the non-linear operators, such as the convective term, the sparsity pattern remains constant during all the simulation but the matrix coefficients change. For these operators, we have followed the strategy of decomposing them into two SpMV: the first product is to update the coefficients of the operator, and the second to apply it.

In particular, the coefficients of the convective operator are updated at each time step according to u_s. If N_f is the number of mesh faces, u_s is a scalar field living in $\mathbb{R}^{N_f}$. On the other hand, in practice, the coefficients of the convective term are stored in a one-dimensional array of dimension N_e, where N_e is the number of non-zero entries in $\mathbf{C}(u_s)$. The arrangement of this array depends on the storage format chosen for the operator. Under these conditions, we define the evaluation of $\mathbf{C}(u_s)$ as a linear operator $\mathbf{E_C} : \mathbb{R}^{N_f} \mapsto \mathbb{R}^{N_e}$, such that:

$$\mathbf{C}(u_s) \equiv \mathbf{E_C} u_s. \tag{2.10}$$

Therefore, the evaluation of the non-linear term, $\mathbf{C}(u_s)\mathbf{u}_c$, results on the concatenation of two SpMV. In particular, the definition of $\mathbf{E_C}$ for the sliced ELLPACK storage format used in this paper is presented in Section 2.4.

In an analogous way, when the LES model is activated, the coefficients of the diffusive term need to be updated according to ν_s. Therefore, the evaluation of the diffusive operator is performed as a linear map $\mathbf{E_D} : \mathbb{R}^{N_f} \mapsto \mathbb{R}^{N_e}$, such that:

Step of Algorithm 1	SpMV	axpy	dot	extras
1 - predictor velocity	8	6	0	0
2.1 - Poisson equation (r.h.s)	3	1	0	0
2.2 - **Poisson equation (per iteration)**	**2**	**3**	**2**	**0**
3.1 - velocity correction	3	3	0	0
4 - velocity at faces	7	0	0	0
5 - eddie viscosity (optional)	9	0	2	1
6 - CFL condition	0	0	0	1
Total outside Poisson solver	**30**	**10**	**2**	**2**

Table 2.1: Number of times that each basic operation is performed in the numerical algorithm

$$\mathbf{D}(\nu_s) \equiv \mathbf{E_D}\nu_s. \tag{2.11}$$

This strategy allows the evaluation of the non-linear operators by calling two consecutive SpMV kernels with constant coefficients, without adding new functions to the implementation.

Table 2.1 sums up the number of times that each kernel is called at the different steps of Algorithm 1. The column "extra" corresponds to operations different from our three main algebraic kernels. In Section 2.5 is shown that these operations have a relatively negligible computing cost.

In our implementation the vector fields $\mathbf{u}_c$ and $\mathbf{g}_c$ are stored as three scalar fields, one for each cartesian component. Therefore the linear operators applied to them result in three SpMV calls. In particular, for the convective and diffusive terms, the three components are multiplied by the same operator, so this can be optimized by using a generalized SpMV (see Section 2.4). On the other hand, the vectorial operator $\mathbf{G}$ is decomposed into the matrices $\mathbf{G}_x$, $\mathbf{G}_y$,$\mathbf{G}_z$ which are operated independently.

In the first step of Algorithm 1 the $SpMV$ kernel is called eight times: one to re-evaluate the coefficients of the convective operator ($\mathbf{E_C}$), then (considering the LES model activated) another one is needed to update the diffusive operator ($\mathbf{E_D}$), and finally six calls are required to apply the convective ($\mathbf{C}$) and diffusive ($\mathbf{D}$) operators to the velocity components. Additionally six AXPY operations are performed, three to evaluate the linear combination of the velocities ($\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}$) and three more to multiply by Ω^{-1}, that is a diagonal matrix stored as a scalar field. Step 2 is separated into two sub-steps: firstly, the right hand side of the Poisson equation is calculated, here the divergence operator ($\mathbf{M}$) requires 3 SpMV; secondly, the preconditioned conjugate gradient (PCG) method is used to solve the Poisson equation. Within a PCG iteration one SpMV, three AXPY and two DOT are performed, the preconditioner is counted as an additional SpMV. In the step 3 the velocity is corrected using the pressure gradient, the gradient operator (**G**) requires three SpMV. The projection of the velocities at the faces in step 4 requires six SpMV coming from the operator $\mathbf{\Gamma}_{c\rightarrow s}$ and one instance of $\mathbf{G}_s$.

When LES model is activated, the eddie viscosity is evaluated ($\nu_{t_s} = \mathbf{K}(\mathbf{u}_c)$) at the fifth step of Algorithm 1. The most costly part of this computation is the evaluation of a tensor dot product over the gradients of the velocity fields. For portability purposes, the linear part of the LES model, that derives on nine SpMV and two DOT, is separated from "extra" operations (`pow exp ,etc`) that depend on the model selected. Moreover, if a dynamic choice of the time step size is enabled one more extra operation is performed at step 6 . This operation consists on calculating the local CFL condition and obtaining the minimum value across the mesh cells.

In summary, the explicit part of the time step requires 30 calls to the SpMV kernel, 10 AXPY, 2 DOT products and few additional operations on the evaluation of the turbulence model and the CFL condition. These "extra" operations are simple kernels compatible with stream processing and easily portable to any architecture. On the other hand, at the solution of the Poisson solver, the repetitions of each kernel depends on the number of iterations required but the ratio is 2 SpMV and 2 DOT for each 3 AXPY.

2.4 Implementation in a hybrid CPU/GPU supercomputer

Once the CFD algorithm has been reconstructed in a portable mode, based on an algebraic operational approach, our aim is to implement this strategy to port our code to hybrid architectures engaging both multi-core CPUs and GPU accelerators. The introduction of accelerators into leading edge supercomputers has been motivated by the power constraints that require to increase the FLOPS per watt ratio of HPC systems. This seems to be a consolidated trend according to the top500 list statistics [21]. Therefore, this development effort is aligned with the current HPC evolution trend. In particular, our computing tests were performed on the Minotauro supercomputer of the Barcelona Supercomputing Center (BSC). Its nodes are composed of two 6-core Intel Xeon E5649 CPUs and two NVIDIA M2090 GPUs, and are coupled via an InfiniBand QDR interconnect. The kernels were implemented in CUDA 5.0 since it is the natural platform to implement a code in NVIDIA GPUs [22]. Moreover the availability of the cuBLAS and cuSPARSE libraries provides all the necessary linear algebra operations, making the code portability straightforward. Nevertheless, for the SpMV kernel, we have focused on optimizing the implementation targeting our specific application context, *i.e.* targeting the sparsity pattern derived from our unstructured discretizations. On the other hand, note that the AXPY and DOT are algebraic kernels independent of any application context, so we have relied on the efficient implementations of the cuBLAS 5.0 library. The rest of this section is focused on the implementation of the SpMV kernel on single GPU and multi-GPU platforms.

Theoretical performance estimation

For a performance estimation of the SpMV we count the floating point operations to be performed and the bytes to be moved from the main memory to the cache, then we can estimate the cost of both processes by comparing with the presumed computing performance and memory bandwidth of the device where the kernel is executed.

We consider as a representative problem the discrete Laplacian operator over a tetrahedral mesh, a second order discretization results in 5 entries per row (diagonal + 4 neighbors of a tetrahedron, except boundary cells). Therefore, if the mesh has N cells the Laplacian operator will contain approximately $5N$ entries. The size of the matrix in memory is $60N$ bytes (with double precision 8-byte values): $5N$ double entries ($40N$ bytes), plus $5N$ integer entries ($20N$ bytes) to store the column indices of the non-zero elements (additional elements to store row indices depend on the chosen storage format). We need to add also the two vectors engaged in the SpMV which contribute with $16N$ additional bytes. Regarding the arithmetics, 5 multiplications and 4 additions are required per each row of the Laplacian matrix, so this results in a total of $9N$ floating point operations. In our performance estimation we assume an infinitely fast cache, zero-latency memory and that each component of the input vector is read only once, i.e. ideal spatial and temporal locality. For the NVIDIA M2090 accelerator with ECC mode enabled the peak memory bandwidth is 141.6 GB/s. Therefore, the total time for moving data from DRAM to cache is $76N/141.6 = 0.54Nns$. On the other hand, the peak performance of the NVIDIA M2090 for double precision operations is 665.6 GFLOPS (fused multiplication addition is considered). Therefore, the total time to perform the floating point operations of the SpMV is estimated as: $9N/665.6 = 0.014Nns$, which is $38\times$ lower than the time to move data from DRAM to cache! This is therefore a clearly memory-bounded kernel characterized by a very low FLOP per byte ratio: $9N/76N = 0.12$. Thus, the efficiency of the implementation basically depends on the memory transactions rather than on the speed of computations. Under these conditions, a tight upper bound for the achievable performance is the product of the arithmetic intensity by the device bandwidth, this gives 16.8 GFLOPS, *i.e.* 2.5% of the peak performance of the M2090 GPU! With this estimation in mind, that is based in some optimistic assumptions, special attention has been paid to the memory access optimization.

Heterogeneous implementation

The utilization of all resources of a hybrid system, such as Minotauro supercomputer, to run a particular kernel requires a heterogeneous implementation. As shown in the previous subsection, the maximum performance achievable with the SpMV kernel is proportional to the memory bandwidth of the device where the kernel is executed. The memory bandwidth of the CPU and the GPU composing the Minotauro supercomputer is 32GB/s and 142 GB/s, respectively. Thus, in idealized conditions the GPU outperforms the CPU by $4.4\times$ (numerical experiments in Section 2.5 show higher accelerations up to $8\times$). Consequently, considering the $4.4\times$ ratio, a balanced workload distribution would be 82% of the rows for the GPU and 18% for the CPU. However, this partition requires a data-transfer process between both devices, to transfer the components of the vector stored on each device that are needed by the other. This communication episode, that needs to be performed through the PCI-e, cancels most of the benefit obtained from the simultaneous execution in both devices. Moreover, if the SpMV kernel is distributed between both devices, the other two kernels need to be partitioned as well. Otherwise, the data transfer requirements to perform the SpMV increase. A heterogeneous implementation of the AXPY does not require any data transfer, but the performance of the DOT operation would be clearly degraded. Therefore, considering the low potential profit of the heterogeneous implementations in the Minotauro supercomputer, in this paper we have performed all computations on the GPU device. However in the parallel implementation (described latter in this section), CPUs are used to asynchronously manage inter-GPU communications by overlapping them with calculations on the GPUs. We expect for the future a closer level of integration between CPUs and GPUs, given by improved interconnection technologies or, as shown in different examples, by the integration of both devices on a single chip.

Unknowns reordering

A two-level reordering is used in order to adapt the SpMV to the stream processing model and to improve the efficiency of the memory accesses. Firstly, the rows are sorted according to the number of non-zero elements. For hybrid meshes this means that the rows corresponding to tetrahedrons, pyramids or prisms are grouped continuously. The boundary elements are reordered to be at the end of the system matrix. This ordering aims to maintain regularity from the SIMD point of view: threads processing rows of the same group have identical flow of instructions. Secondly, a band-reduction Cuthill-McKee reordering algorithm [23] is applied at each subgroup in order to reduce the number of cache misses when accessing the components of the multiplying vector. Note that in the SpMV kernel the random memory accesses affect only the multiplying vector, the matrix coefficients and the writes on the solution vector are sequential. Each component of the multiplying vector is accessed as many times as neighbors has the associated mesh cell, so this is the measure of the potential temporal locality or cache reuse.

Generalized SpMV

Another way to reduce the memory communications is by grouping wherever possible different SpMV into a so-called "Generalized SpMV" (GSpMV), that multiplies the same matrix to multiple vectors simultaneously. For instance, the velocity vector in any cell is described by three components (u, v, w), which in practice are stored on three independent scalar fields. Some operators are multiplied by each of the three velocity components, resulting in three calls to the SpMV kernel using the same sparse matrix. Therefore, if the components are operated independently, the same matrix coefficients and indices are fetched three times from the DRAM to the cache. With the GSpMV the matrix elements are fetched only once and the arithmetic intensity increases by $\approx 2.1\times$. However the reuse of components of the multiplying vector stored in the cache may reduce, because it is filled with elements of the three velocity vectors instead of only one. Nevertheless, in our numerical experiments we have obtained in average 30% time reduction by using the GSpMV.

Storage formats

Given a sparse matrix, its sparsity pattern determines the most appropriate storage format. The goal is to minimize the memory transmissions and increase the arithmetic intensity by: i) reducing the number of elements required to describe the sparse matrix structure, ii) minimizing the memory divergence caused by its potential irregularity. The sparse matrices used in our CFD algorithm arise from discretizations performed on unstructured meshes. Considering that we are using second order schemes for the discretization of the operators, and that the typical geometrical elements forming the mesh are tetrahedrons, pyramids, prisms and hexahedrons, generally the number of elements per row ranges between 5 and 7 for the interior elements and is 1 or 2 for the few rows associated to boundary nodes. Since we are not concerned in modifying the mesh, we restrict our attention to static formats, without considering elements insertion or deletion.

With this scenario in mind, an ELLPACK-based format is the best candidate to represent our matrices. The standard ELLPACK format is parametrized by two integers: the number of rows, N, and the maximum number of non-zero entries per row, K. All rows are zero-padded to length K forcing regularity. The matrix is stored in two arrays: one vector of doubles with the matrix coefficients (`Val`), and a vector of integers with the corresponding column indices (`Col`). The row indices do not need to be explicitly stored because it is assumed that each K components of the two previous vectors correspond to a new row. This regularity, benefits the stream processing model because each equal-sized row can be operated by a single thread without imbalance.

However, the zero-padding to the maximum initial row-length can produce an important sparse overhead, specially when there is a reduced number of polygons in the mesh with maximal number of faces. To overcome this overhead, a generalization of the previous algorithm, called sliced ELLPACK (sELL) [24], has been implemented. In the sELL format the matrix is partitioned into slices of S adjacent rows, which are stored using the ELLPACK format. The benefit of this approach is that each slice has its particular K parameter, reducing the overall sparse overhead. The reordering strategy described above, i.e. grouping rows by its length, additionally increases the regularity within each slice. In the sELL format it is added an integer-vector, called `slice_start`, holding the indices of the first element on each slice. Thus, for the i'th slice, the number of non-zero entries per row (including the zero-padded elements) can be calculated as

$$K_i = \frac{slice\_start[i+1] - slice\_start[i]}{S}.$$

If $S = N$ the sELL storage format becomes the standard ELLPACK. On the other hand, if $S = 1$ no zeros are padded and the format becomes equivalent to the Compressed Sparse Row format (CSR). For GPU implementations S is set equal to the number of threads launched per block, this way a single thread is associated to each row of the slice.

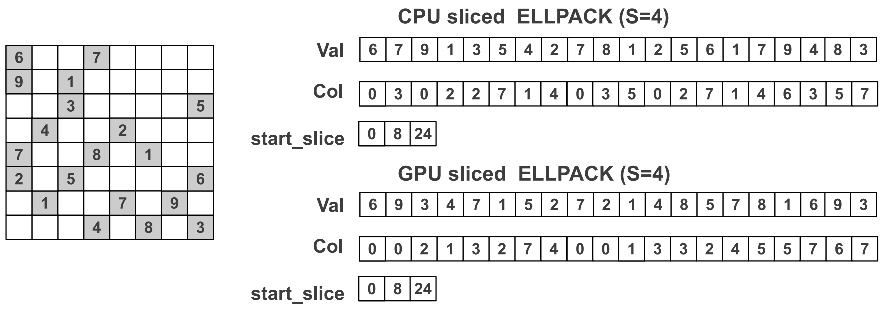

CPU sliced ELLPACK (S=4)
Val 6 7 9 1 3 5 4 2 7 8 1 2 5 6 1 7 9 4 8 3
Col 0 3 0 2 2 7 1 4 0 3 5 0 2 7 1 4 6 3 5 7
start_slice 0 8 24
GPU sliced ELLPACK (S=4)
Val 6 9 3 4 7 1 5 2 7 2 1 4 8 5 7 8 1 6 9 3
Col 0 0 2 1 3 2 7 4 0 0 1 3 3 2 4 5 5 7 6 7
start_slice 0 8 24

Figure 2.1: An example of sELL format and its memory layout for CPUs and GPUs.

Figure 2.1 shows the storage of a sparse matrix using the sELL format for both a CPU and a GPU execution. In this case S=4 and the matrix is divided in two slices with 3 and 4 elements per row, respectively. However, note that the memory layout is different in both cases. In the CPU execution the elements are stored in row-major order within each slice, because a sequential process operates one row after the other. In the GPU execution a single thread is associated to each row, consequently, the elements are stored in column-major order to achieve coalesced memory accesses to the `Val` and `Col` arrays. Further details of this aspects can be found in [25].

Non linear operators

Since the sliced ELLPACK storage format has already been described in previous subsection, now it can be explicitly defined the respective operator $\mathbf{E_C} : R^{N_f} \mapsto R^{N_e}$ that, given u_s, generates the array `Val` storing the corresponding coefficients of the convective operator in the sELL format. However, first of all the discretization of the convective operator needs to be explicitly specified:

$$[\mathbf{C}(u_s)\mathbf{u}_c]_i = \sum_{j \in nb(i)} (\mathbf{u}_{cj} + \mathbf{u}_{ci}) \frac{u_{sij} A_{ij}}{2} \tag{2.12}$$

where i is a mesh node, $nb(i)$ are its neighboring nodes, A_{ij} is the area of the face between node i and node j, and u_{sij} the corresponding component of the field u_s. For each node i, the indices of the non-zero entries of its row are: $entries(i) = nb(i) \cup i$. Being these indices sorted in ascending order, we refer to the relative position of any index $j \in entries(i)$ with the notation $ord_i(j)$. For each ordered pair (i, j) of neighboring nodes, the next two coefficients are introduced to $\mathbf{E_C}$:

$$\mathbf{E_C}(ind(i,j), ij) = \mathbf{E_C}(ind(i,i), ij) = \frac{A_{ij}}{2} \tag{2.13}$$

where ij is the index corresponding to the face between nodes i and j, and the function $ind(i, j)$, that fixes the layout of the elements of the convective operator in the array `Val`, is defined as:

$$ind(i,j) = i\%S + ord_i(j) * S + \sum_{l < \frac{i}{S}} K_l * S, \tag{2.14}$$

where S and K_l are parameters of the sELL format described in previous subsection. The first two terms of Equation 2.14 define the position of the new index within its slice: the first term represents the row and the second the column. The third term is the offset corresponding to all previous slices.

The sparse matrix $\mathbf{E_C}$ has constant coefficients and is also stored in a compressed format. However, its rows can not be reordered since the order is determined by the sELL layout of $\mathbf{C}(u_s)$. Therefore, with the purpose of avoiding a high sparse overhead produced by the zero-padding, $\mathbf{E_C}$ is stored using the standard CSR format [26].

Finally, note that the definition of $\mathbf{E_D}$, the sparse linear operator used to evaluate the coefficients of $\mathbf{D}(\nu_s)$, is performed following the same strategy than with $\mathbf{E_C}$.

Multi-GPU parallelization

At the top level the parallelization strategy to run the code on multiple GPUs is based on a standard domain decomposition: the initial mesh, $\mathcal{M}$, is divided into P non-overlapping sub-meshes $\mathcal{M}_0, ..., \mathcal{M}_{P-1}$, and an MPI process executes the code at each sub-mesh. For the $i'th$ MPI process its unknowns can be categorized into different sub-sets:

- *owned* unknowns are those associated to the nodes of $\mathcal{M}_i$;
- *external* unknowns are those associated to the nodes of other sub-meshes;
- *inner* unknowns are the owned unknowns that are coupled only with other owned unknowns;
- *interface* unknowns are the owned unknowns coupled with external unknowns;
- *halo* unknowns are the external unknowns coupled with owned unknowns.

In our MPI+CUDA implementation, the mesh is divided into as many sub-domains as GPUs engaged. The communication episodes are performed on three stages: i) copy data from GPU to CPU, ii) perform the required MPI communication, iii) copy data from CPU to GPU. Note that the inter-CPU communications are performed through the system network while the communications between the CPU and GPU through the PCI-e bus. For the SpMV kernel, we have developed a classical overlapping strategy, where the sub-matrix corresponding to the inner unknowns is multiplied at the same time that the halo unknowns are updated. Once the updated values of the halo unknowns are available at the GPU, the sub-matrix corresponding to the interface unknowns can also be multiplied. An schematic representation of this two-stream concurrent execution model is depicted in Figure 2.2. Note that a synchronization episode is necessary after both parts of the matrix are multiplied. This strategy is really effective because communications and computations are performed simultaneously on independent devices that do not disturb each-other (see examples on Section 2.5).

In order to facilitate and optimize the partition of the sparse matrices to perform the overlapped parallel SpMV, a first reordering of variables is performed forming three groups: inner, interface and halo variables. This ordering precedes and is prioritized versus the orderings described above for grouping rows with the same number of elements and for band-reduction. Therefore, for parallel executions a three-level reordering strategy is applied in order to optimize the data locality and regularity. Further details of our overlapping strategy can be found in our previous work [9].

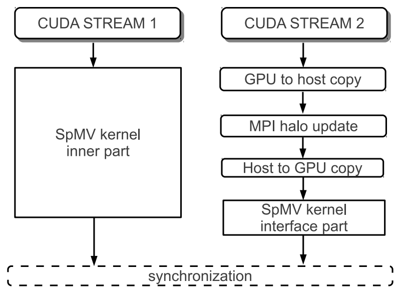


Figure 2.2: Two-stream concurrent execution model.

2.5 Computing experiments and performance analysis

The performance testing has been carried out on the Minotauro supercomputer of the Barcelona Supercomputing Center (BSC), its characteristics are summarized in Section 2.4. In the Minotauro supercomputer there is one 6-core CPU for each GPU, therefore, at comparing the multi-CPU vs the multi-GPU implementation we have respected this ratio running 6 CPU-cores for each GPU.

2.5.1 Profiling the algorithm on a turbulent flow around a complex geometry

First of all, the MPI-only and the MPI+CUDA versions of the code are profiled. The percentage of the total execution time spent in each algebraic kernel has been measured during the time integration process. The test case is the turbulent flow around the ASMO car geometry ($Re = 7 \times 10^5$), for which a detailed study was previously published in [27]. The mesh has around 5.5 millions of control volumes that include tetrahedrons, pyramids and triangular prisms that are used in the boundary layer. The solver tolerance relative to the right-hand-side norm for the Poisson equation is set to 10^{-5}. In this particular case the number of PCG solver iterations averaged across a large number of time steps is around 37 when using a Jacobi preconditioner. Additionally, the LES turbulence modeling is enabled with the WALE [28] eddie-viscosity model.

The profiling results on different number of CPUs and GPUs are shown in Figure 2.3. Results for the multi CPU implementation show that the contribution of the SpMV in the overall algorithm stays constant at around 78% with a growth of the number of 6-core CPUs from 4 to 128. This indicates that the SpMV accelerates equally as the overall time-step. On the other hand, the AXPY operation is the leader in speedup: its contribution reduces from 14% to around 6%. This was expected since there are no communications in the AXPY operation. Finally, the contribution of the DOT operation grows with number of CPUs due to the collective reduction communications, its negative contribution is counteracted by the AXPY super-linearity. Finally, the remaining 2% of time is spent in other operations (the evaluation of the eddie viscosity on step 5 of Algorithm 1, and the evaluation of the CFL condition on step 6 of Algorithm 1). Therefore, the three basic algebraic kernels sum up to 98% of the time-step execution time on CPUs. This fact emphasizes the strong natural portability of the algorithm.

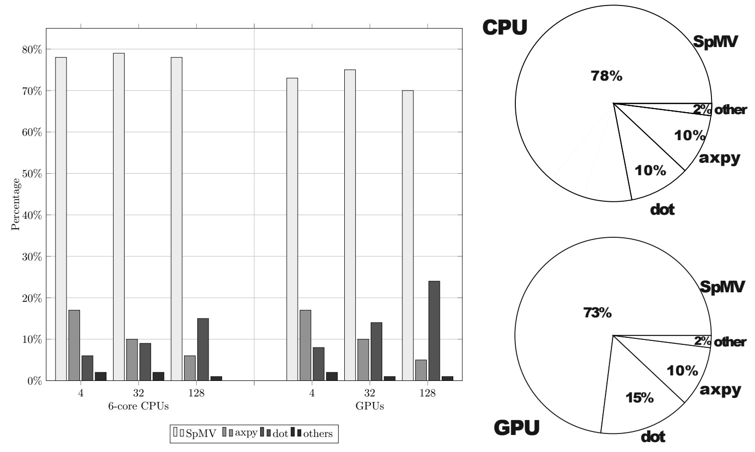


Figure 2.3: Relative weight of the main operations for different number of CPUs and GPUs (left) and diagrams of the average relative weight (right)

The GPU execution shows a lower percentage of time spent on the SpMV kernel. As demonstrated in next subsection, this is due to a better exploitation of the device bandwidth by the SpMV kernel on GPUs. It is also observed a significant increase of the DOT percentage, which is mainly penalized by the all-reduce communication and the overhead of the PCI-Express host-device transactions.

In Figure 2.3 (right) are depicted the average results for all the previous tests. Note that the distribution of the computing cost among different kernels depends on the number of iterations required by the Poisson solver, because the distribution is substantially different for the solver and for the explicit part of the time step. For instance, if PCG solver in the same test is considered separately, the average time distribution for a single CG iteration executed on the GPUs is 58% for the SpMV, 23% for the AXPY and 17% for the DOT kernel. For the explicit part of the time step, the distribution is 92%, 2% and 5%, respectively.

2.5.2 Performance of SpMV on a single GPU and a single 6-core CPU

The SpMV tests have been carried out for the discrete Laplace operator since it represents the dominant sparsity pattern. The Laplace operator has been discretized on 3D unstructured meshes. The number of non-zero entries per row ranges from 5 to 7 for tetrahedral, prismatic, pyramidal and hexahedral cells. Our implementation of the CPU and GPU SpMV with the sELL storage format is compared with general-purpose SpMV implementations of commonly used standard libraries: Intel Math kernel library (MKL) 13.3 for CPUs and cuSPARSE 5.0 for GPUs. Intel MKL only supports the CSR format, while cuSPARSE supports the CSR format and a hybrid (HYB) format, which automatically determines the regular parts of the matrix that can be represented by ELLPACK while the remaining are solved by COO format [26]. The execution on the 6-core CPUs employs OpenMP shared memory loop-based parallelization with a static scheduling. In all cases, the two-level reordering explained in Section 2.4 (grouping of rows with equal number of non-zero entries + band reduction reordering) is used in order to improve the memory performance. Unknowns reordering results in 40% time reduction in average for CPU executions.

The achieved net performance, in GFLOPS for the different storage formats under consideration and for different mesh sizes is shown in Table 2.2. In all the cases our in-house sELL format shows the best performance. In average the speedup versus the Intel MKL and NVIDIA cuSPARSE library is 38% and 11%, respectively. Consequently, the sELL format has been chosen for the rest of this paper.

Device, SpMV format	Mesh size, thousands of cells					
	50	100	200	400	800	1600
CPU CSR *MKL*	2.45	2.18	1.49	1.37	1.30	1.18
CPU sliced ELLPACK	**3.44**	**3.02**	**2.89**	**2.76**	**2.41**	**2.06**
GPU CSR *cuSPARSE*	3.64	4.10	4.40	4.58	4.79	4.70
GPU HYB *cuSPARSE*	8.74	11.25	13.36	14.93	15.62	15.94
GPU sliced ELLPACK	**10.91**	**12.79**	**14.90**	**15.92**	**16.15**	**16.37**

Table 2.2: Net performance in GFLOPS obtained with different implementations of various matrix storage formats

Figure 2.4, shows the performance evolution of the sELL executions for matrices of different sizes and for both the CPU and GPU devices. There is a solid horizontal line on the plots that indicates an estimation of the maximal theoretically achievable performance evaluated as explained in Section 2.4. Up to 90% of the peak estimation is achieved on the CPU and up to 97% on the GPU. It is important to note that the CPU shows better performance on a smaller matrices while the GPU on bigger ones. This trend can be clearly seen in Figure 2.4. The CPU performance decreases with the matrix size due to the increasing weight of cache misses. In other words, the perfect temporal locality assumed in our estimation is not met when the problem size grows. In particular, some of the multiplying vector components that need to be reused can not be held on the cache because of its limited size. On the GPU the effect is the opposite, there is a net performance growth with the matrix size due to the resulting higher occupancy of the stream multiprocessors, which allows to more efficiently hide memory latency overheads by means of hardware multi-threading. Saying it from the opposite perspective, if the amount of parallel threads (i.e. rows) is not enough the device can not be fully exploited. For the executions on the GPU the grid of threads was parametrized in accordance to the occupancy calculator provided by NVIDIA [22]. Moreover, on the NVIDIA M2090 GPU the distribution of the local memory between shared memory and L1 cache can be tuned. For the SpMV kernel the best performance is obtained when the maximum possible fraction of the local memory of the stream multiprocessors is used for cache functions (48KB).

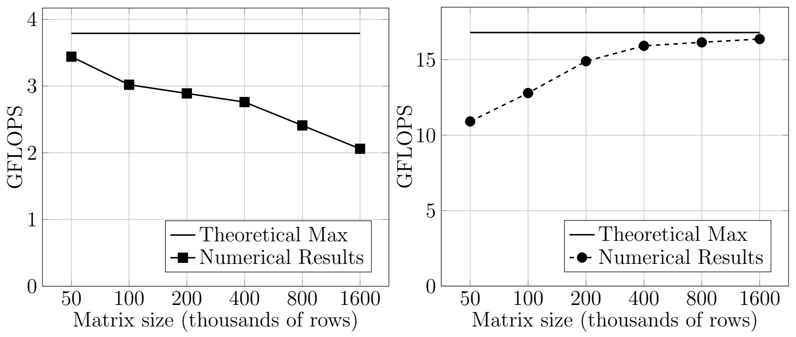


Figure 2.4: Net performance achieved on a single Intel Xeon E5649 6-core CPU (left) and a single NVIDIA M2090 GPU (right) for different mesh sizes. Solid line indicates an estimation of maximal theoretically achievable performance

Finally, Figure 2.5 depicts the speedup of the execution on a single GPU versus the execution on a single 6-core CPU. Due to the trends observed in the abovementioned tests the speedup significantly grows with the mesh size, since the CPU performance goes down with size and the GPU performance goes up. The speedup ranges in the tests from around 3× up to 8×. Being the ratio between the GPU and CPU peaks memory bandwidth 4.4 (141.6GB/s for the GPU and 32GB/s for the CPU), and recalling that the SpMV is a clearly memory bounded operation, we can conclude from the result shown in Figure 2.5 that the bandwidth is better harnessed on the GPU than on the CPU, because in most of cases the speedup achieved is greater than 4.4. For the matrix with $50K$ rows the opposite result is observed, however this case is particularly small for devices with 6 GB and a 12 GB of main memory such as the the GPU and CPU under consideration, so it does not represent realistic usage.

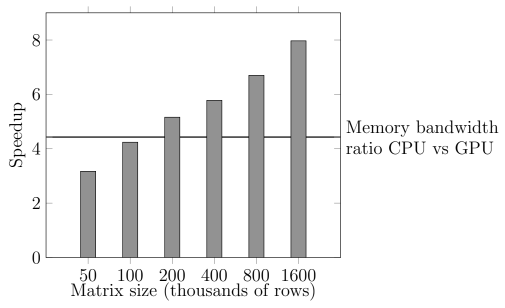


Figure 2.5: Speedup of the SpMV execution on a single GPU vs. on a 6-core CPU, for different mesh sizes.

2.5.3 Parallel performance of the SpMV on multiple GPUs

Firstly, we evaluate the benefit of overlapping computations with communications. In Figure 2.6 is compared the time needed to perform the SpMV with and without overlapping, engaging 128 GPU for different matrix sizes. The matrices used range between 50 and 200 million (M) rows, corresponding to workloads between 0.4M and 1.6M per GPU, respectively. From 64% up to 86% of the communication costs are hidden behind computations using the overlapping strategy. The benefit increases with the problem size since the weight of the computations grows and the communications can be more efficiently hidden . All the remaining parallel performance tests are carried out using the overlap mode. Further details on the implementation of the SpMV with the overlapping strategy are in Section 2.4.

Weak speedup tests for the SpMV on multiple GPUs are shown in Figure 2.7. Three different local matrix sizes are considered: 0.4M, 0.8M, and 1.6M rows. The execution time grows for the biggest load only 1.2$\times$ while the problem size is increased 128$\times$. For the smallest load considered this factor is around 2$\times$. The slowdown is caused by an increase of the communication costs with the number of GPUs engaged, and is proportional to the relative cost of communications. The maximal load of 1.6M cells per GPU was chosen according to the limitation of the mesh size that could fit in memory if we would run a complete CFD simulation.

Furthermore, the strong speedup of the SpMV kernel is shown in Figure 2.8 (left) for matrices of different sizes. Certainly, the larger is the size of the problem the better is the speedup obtained, because the relative weight of the communications is inversely proportional to the problem size. In the right part of the figure the same results are shown in terms of parallel efficiency (PE). It can be observed that in the

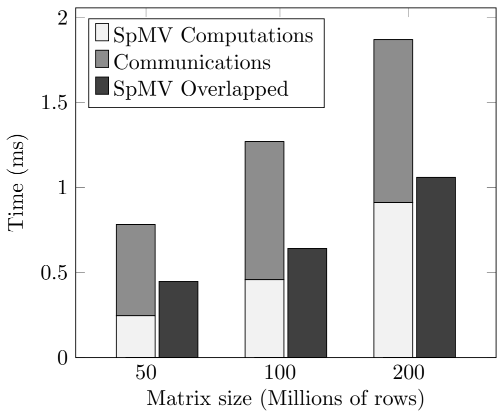


Figure 2.6: Effect of overlapping communications and computations in the SpMV for different mesh sizes on 128 GPU

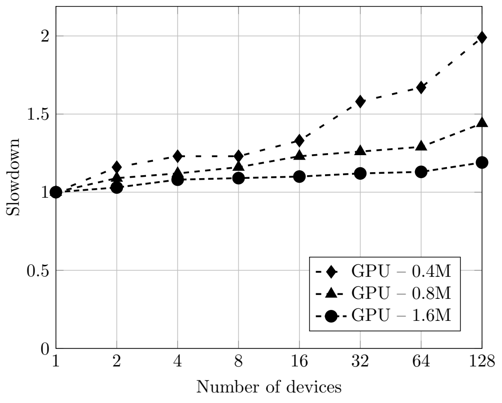


Figure 2.7: Weak speedup tests for different local matrix sizes

range of tests performed the PE achieved mainly depends on the local workload rather than on the number of devices engaged: ∼80% PE is obtained for a workload of 400K rows, 55-60% for 200K rows, 37-41% for 100K rows and ∼30% for 50K rows. Therefore, if we are seeking for high PE we should not reduce the local problem below 400K rows per GPU. This result is consistent with the decrease of the GPU performance for matrices below 400K rows shown in Figure 2.4, that is produced by a lack of occupancy of the device. Indeed, note that the load of 400K unknowns could be considered a moderate one for a GPU with 6 GB of main memory. In any case, the resources engaged in a simulation should be consistent with this evaluation in order to use efficiently the granted computing time.

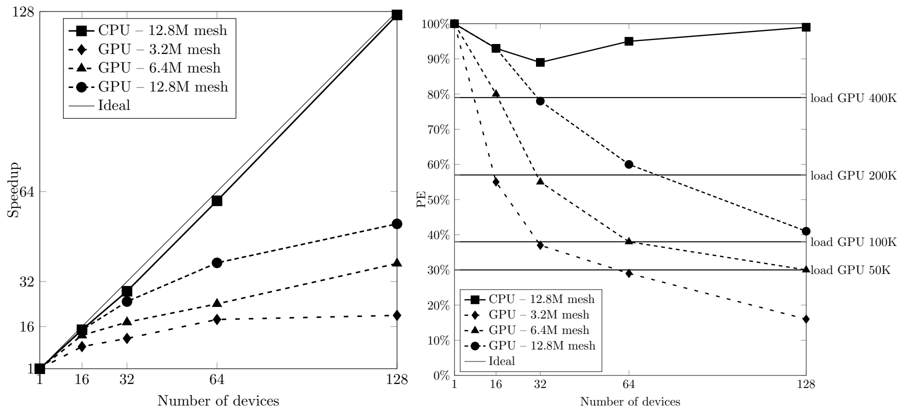


Figure 2.8: Strong speedup on multiple GPUs for different mesh sizes (left). Parallel efficiency on multiples GPU for different mesh sizes (right)

In Figure 2.8 is also included the speedup of the multi-CPU SpMV execution for the 12.8M cells mesh, which appears to be much higher than for GPUs: 127× vs. 53× on 128 devices. The PE degradation with the number of GPUs is twofold: i) an increase of the communications overhead ii) the sequential performance of the GPU on the solution of the local problem reduces when the occupancy falls. An opposite situation is observed for the CPUs. The performance of the CPU increases when the local problem reduces since the cache memories reach a larger portion of the problem. In this situation the increase of the communication is compensated by the boost on the CPU performance. Figure 2.9 (left) demonstrates this effect in detail. It shows how the net performance of one device changes with the growth of the number of computing devices (CPUs or GPUs) engaged, i.e with the reduction of the local problem size. Results shown are normalized by the performance achieved when running the whole matrix on a single device. The CPU performance increases more than twice due to the cache effect while there is a 25% degradation of the GPU performance.

Figure 2.9 (right) shows an estimation of how the speedup plot would look like if the CPU and GPU performance would remain the same as the achieved on a single device. i.e. canceling cache and occupancy effects. The CPU and GPU plots in these imaginary conditions appear very close to each other, therefore, we conclude that eventually the speedup on the CPUs is produced by the cache driven super-linear acceleration of the local computations that lacks on the GPUs. The variation between the real case and the imaginary estimation is minimal in the GPUs, the reason is that the communications dominate the computations in the overlapping scheme, so canceling the degradation produced by the occupancy fall only affects the cost of the interface sub-matrix product (see Section 2.4). This result also demonstrates the efficiency of the overlapping communications scheme versus the standard strategy, because, despite the GPU execution is 9× faster for the sequential run, when the cache effects are canceled the speedup on CPUs and GPUs is very similar. Therefore, the overhead produced by the overlapped communications is much lower than the one obtained with the standard communication process.

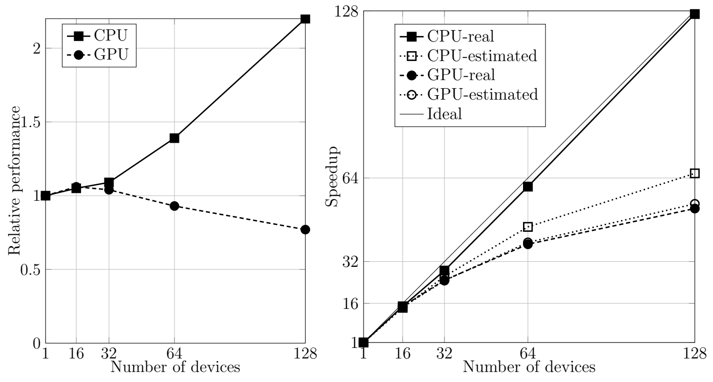


Figure 2.9: Relative net performance of CPUs and GPUs compared to the performance on a single device for the mesh of 12.8M cells (left); estimation of how speedup would look like if the CPU and GPU performance would remain the same as on a single device (right)

2.5.4 Experiments with complete CFD simulations

The overall CFD algorithm has been evaluated on two test cases: the LES of the ASMO car ($Re = 7 \times 10^5$) executed on a mesh of 5.5M cells, that has been previously used for profiling; and the LES of a driven cavity ($Re = 10^5$) with a mesh of 12.8M cells (the driven cavity was chosen in this paper just for its simple geometry that makes generation of meshes of different sizes very easy). In Figure 2.10 (left) strong speedup results are shown for the two cases. The measurements are averaged across numerous time steps during the period when the flow already reaches the statistically stationary regime. The Poisson solver tolerance was set to 10^{-5}, and the LES turbulence modeling is enabled with the WALE [28] eddie-viscosity model.

In Figure 2.10 (left) are also includes estimations of the strong speedup. These are based only on measurements for the three basic algebraic kernels evaluated separately and the number of calls of each kernel shown in Table 2.1. The simple formula reads:

$$
\begin{aligned}
T_{exp} &= 30T_{spmv} + 10T_{axpy} + 2T_{dot} \\
T_{imp} &= 2T_{spmv} + 3T_{axpy} + 2T_{dot} \\
T_{total} &= T_{exp} + it \cdot T_{imp}
\end{aligned}
$$

where the subindex of T indicates the corresponding kernel, and *it* refers to the iterations required by the Poisson solver. The high level of agreement achieved provides also great portability in terms of performance evaluation: we can predict the performance and scalability of our code in any architecture by just studying the three main kernels separately.

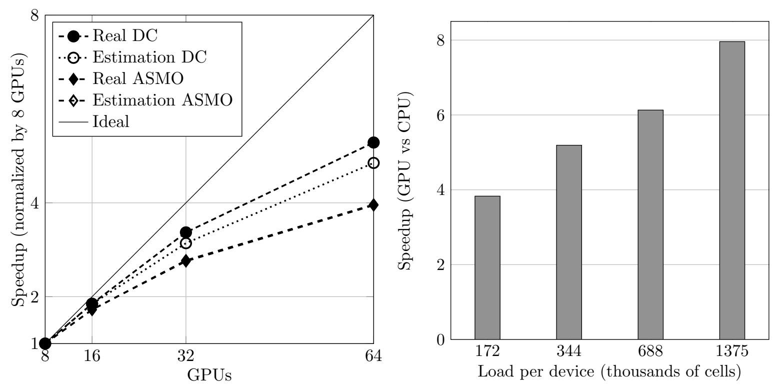


Figure 2.10: Strong speedup for the overall time-step (real, and estimation based on evaluation of performance of the basic operations) for the ASMO car case with mesh of 5.5M cells and the driven cavity (DC) case with mesh of 12.8M cells (left); speedups on GPUs vs 6-core CPUs, ASMO case (right)

Finally, the speedups achieved for the overall CFD algorithm comparing the MPI+CUDA implementation on multiple GPUs with the MPI only implementation on the equal number of 6-core CPUs is shown in Figure 2.10 (right). The different loads per device correspond to the different number of devices engaged. The GPU version outperforms the CPU one by a factor that ranges from $4\times$ to $8\times$ and increases with the local problem size.

2.6 Concluding remarks

The contribution of this paper is twofold. Firstly, we propose a portable modeling for LES of incompressible turbulent flows based on an algebraic operational approach. The main idea is substituting stencil data structures and kernels by algebraic storage formats and operators. The result it that around 98% of computations are based on the composition of three basic algebraic kernels: SpMV, AXPY and DOT, providing a high level of modularity and a natural portability to the code. Among the three algebraic kernels the SpMV is the dominant one, it substitutes the stencil iterations and the non-linear terms, such as the convective operator, are rewritten as two consecutive SpMVs. The modularity and portability requirements are motivated by the current disruptive technological moment, with a high level of heterogeneity and many computing models being under consideration without any clarity of which one is going to prevail.

The second objective has been the implementation of our model to run on heterogeneous clusters engaging both CPU and GPU co-processors. This objective is motivated by the increasing presence of accelerators on HPC systems, driven by power efficiency requirements. We have analyzed in detail the implementation of the SpMV kernel on GPUs, taking into account the characteristics of the matrices derived from our discretization. Our in-house implementation based on a sliced ELLPACK format clearly outperforms other general purpose libraries such as MKL on CPUs and cuSPARSE on GPUs. Moreover, for multi-GPU executions we have developed a communication-computations overlapping strategy. On the other hand, the AXPY and DOT kernels don't require specific optimizations because they are application-independent kernels with optimal implementations in libraries such as cuSPARSE.

Finally, several numerical experiments have been performed on the Minotauro supercomputer of the Barcelona Supercomputing Center, in order to understand in detail the performance of our code on multi-GPU platforms, and compare it with multi-core executions. First we have profiled the code for both implementations showing that certainly 98% of time is spent on the three main algebraic kernels. Then, we have focused on the SpMV kernel, we have shown its memory bounded nature, and how the throughput oriented approach of the GPU architecture better harnesses the bandwidth than the standard latency reducing strategy implemented on CPUs by means of caches and prefetching modules. The result is that although the bandwidth ratio between both devices is 4.4 the speedup of the GPU vs the CPU implementation reaches up to $8\times$ in our tests. Then the benefits of the overlapping strategy for multi-GPU executions has been tested, showing that large part of the communication (86%) can be hidden. We have also included strong and week speedup tests engaging up to 128 GPUs, in general good PE is achieved if the workload per GPU is kept reasonable. Considering the overall time-step, the multi-GPU implementation outperforms the multi-CPU one by a factor ranging between 4× and 8× depending on the local problem size. Finally, we have demonstrated that the performance of our code can be very well estimated by only analyzing the three kernels separately.

References

[1] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceeding of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009.

[2] A. Alfonsi, S. Ciliberti, M. Mancini, and L. Primavera. Performances of navier-stokes solver on a hybrid cpu/gpu computing system. *Lecture Notes in Computer Science*, 6873:404–416, 2010.

[3] Elsen E., LeGresley P., and Darve E. Large calculation of the flow over a hypersonic vehicle using a gpu. *Journal of Computational Physics*, 227:10148–10161, 2008.

[4] R. Rossi, F. Mossaiby, and S.R. Idelsohn. A portable opencl-based unstructured edge-based finite element navier-stokes solver on graphics hardware. *Journal of Computers & Fluids*, 81:134–144, 2013.

[5] S.A. Soukov, A.V. Gorobets, and P.B. Bogdanov. Opencl implementation of basic operations for a high-order finite-volume polynomial scheme on unstructured hybrid meshes. In *Procedia Engineering*, volume 61, pages 76–80, 2013.

[6] Dana A. Jacobsen and Inanc Senocak. Multi-level parallelism for incompressible flow computations on gpu clusters. *Journal of Parallel Computing*, 39:1–20, 2013.

[7] P. Zaspel and M. Griebel. Solving incompressible two-phase flows on multi-gpu clusters. *Journal of Computers & Fluids*, 80:356–364, 2013.

[8] Ali Khajeh-Saeed and J. Blair Perot. Direct numerical simulation of turbulence using gpu accelerated supercomputers. *Journal of Computational Physics*, 235:241–257, 2013.

[9] G.Oyarzun, R. Borrell, A. Gorobets, and A. Oliva. Mpi-cuda sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. *Computers & Fluids*, 92:244–252, 2014.

[10] I.C. Kampolis, X.S. Trompoukis, V.G. Asouti, and K.C. Giannakoglou. Cfd-based analysis and two-level aerodynamic optimization on graphics processing units. *Computer Methods in Applied Mechanics and Engineering*, 199:712–722, 2010.

[11] V. G. Asouti, X. S. Trompoukis, I. C. Kampolis, and K. C. Giannakoglou. Unsteady cfd computations using vertex-centered finite volumes for unstructured grids on graphics processing units. *International Journal for numerical methods in fluids*, 67:232–246, 2010.

[12] F.X. Trias, O. Lehmkuhl, A. Oliva, C.D. Pérez-Segarra, and R.W.C.P. Verstappen. Symmetry-preserving discretization of navier–stokes equations on collocated unstructured grids. *Journal of Computational Physics*, 258:246–267, 2014.

[13] R. Verstappen and A. Veldman. Symmetry-preserving discretization of turbulent flow. *Journal of Computational Physics*, 187:343–368, 2003.

[14] O. Lehmkuhl, I. Rodríguez, R. Borrell, J. Chiva, and A. Oliva. Unsteady forces on a circular cylinder at critical Reynolds numbers. *Physics of Fluids*, page 125110, 2014.

[15] I. Rodríguez, R. Borrell, O. Lehmkuhl, C. D. Pérez-Segarra, and A. Oliva. Direct Numerical Simulation of the Flow over a Sphere at $Re = 3700$. *Journal of Fluid Mechanics*, 679:263–287, 2011.

[16] Chorin A. Numerical solution of the navier-stokes equations. *J. Math. Comp*, 22:745–762, 1968.

[17] L. Jofre, O. Lehmkuhl, J. Ventosa, F.X. Trias, and A. Oliva. Conservation properties of unstructured finite–volume mesh schemes for the navier–stokes equations. *Numerical Heat Transfer, Part B: Fundamentals*, 65, 2014.

[18] F.X. Trias, D. Folch, and A. Oliva A. Gorobets. Building proper invariants for eddy-viscosity subgrid-scale models. *Physics of Fluids*, 27, 2015.

[19] O. Lehmkuhl, C.D. Perez-Segarra, R. Borrell, M. Soria, and A. Oliva. Termofluids: A new parallel unstructured cfd code for the simulation of turbulent industrial problems on low cost pc cluster. *Lecture Notes in Computational Science and Engineering. Parallel Computational Fluid Dynamics*, 67:275–282, 2007.

[20] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling. Programming cuda and opencl: A case study using modern c++ libraries. *SIAM Journal on Scientific Computing*, 35(5):C453–C472, 2013.

[21] Ranking of supercomputers according to linpack benchmark. Webpage: http://www.top500.org, 2015.

[22] Nvidia. Nvidia cusparse and cublas libraries. Webpage: http://developer.nvidia.com/cuda-toolkit, 2007.

[23] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th Nat. Conf. ACM*, pages 157–172, 1969.

[24] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. *High Performance Embedded Architectures and Compilers Lecture Notes in Computer Science*, 5952:111–125, 2010.

[25] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, 2009.

[26] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Nvidia Technical Report NVR-2008-004, 2008.

[27] D.E. Aljure, O. Lehmkuhl, I. Rodríguez, and A. Oliva. Flow and turbulent structures around simplified car models. *Computers and Fluids*, 96:122–135, 2014.

[28] F. Nicoud and F. Ducros. Subgrid-scale stress modelling based on the square of the velocity gradient tensor. *Flow, Turbulence and Combustion*, 62:183–200, 1999.

3

Hybrid auto-balanced implementation of parallel unstructured CFD kernels in MontBlanc ARM-based platforms

This work has been developed in the context of TermoFluids as End-User group in the "Mont-Blanc project: european approach towards energy efficient high performance."

Abstract. The challenge of exascale computing is fomenting the investigation of new computer architectures that satisfy the power consumption constraints. Supercomputers have drifted into hybridization of the systems with nodes composed by multicore-CPUs and accelerators to increase the FLOP per watt ratio. In this context, the Mont-Blanc project aims at creating the first mobile-based hybrid supercomputer with the potential of becoming a reference of exascale systems. This paper reports the performance of our portable implementation model for CFD simulations in the Mont-Blanc ARM-based platforms. Our CFD algorithm is composed of a minimal set of basic linear algebra operations compatible with stream processing and SIMD parallelism. Its implementation is focused in engaging all the computing resources in each ARM-node by performing a concurrent heterogeneous execution model that distributes the work between CPUs and GPUs. The load balancing algorithm exploits a tabu search strategy that tunes the optimal workload distribution at run-time. In addition, an overlap of computations with MPI communications is used for hiding part of the data exchange costs that become the bottleneck when engaging multiple Mont-Blanc nodes. Finally, a comparison of the Mont-Blanc prototypes with high-end supercomputers in terms of the net performance and energy consumption provides some guidelines of the behavior of CFD applications in ARM-based architectures.

3.1 Introduction

The HPC community is constantly exploring alternative architectures and programming models for surpassing the power limitations of the multi-core CPUs. Hybridization of the computer nodes is a trend that have consolidated its position within the current leading edge supercomputers. The hybrid nodes are composed by multi-core CPUs and massively parallel accelerators, requiring several levels of parallelism for exploiting the different execution models. Such systems benefits from the accelerators that employ the stream processing paradigm for increasing the throughput and the FLOP per watt ratio. On the other hand, the traditional multi-core CPUs systems have evolved towards energy efficient system-on-chip (SoC) architectures. By doing so, the different components of the node are fused and integrated into a single chip in order of minimizing the energy costs.

Nowadays, several institutions and governments are investing in the investigation of different aspects of HPC that could lead to the future generation of supercomputers. This set of initiatives have entitled the problem as *the exascale challenge* [1,2]. In short, the problem consist in developing a sustainable exascale computing system ($1 \times 10^{18} FLOP/s$) with a maximum power consumption of 20MW. As a consequence, a benchmark was necessary for determining the top power efficient supercomputers. The Green500 list [3] classifies the supercomputers according to the energy efficient metric (GFLOP/s/Watt) when running the LINPACK tests. By extrapolating the energy consumption of the #1 system in the current Green500 list, we conclude that would be necessary a 25× efficiency improvement for achieving exascale computing at the proposed power constraint.

In this context, the Mont-Blanc project [4] is an European initiative devoted into designing a new type of computer architecture for HPC systems. Its hybrid nodes are based in embedded mobile devices (ARM) capable of providing energy efficient solutions. The strategy aims at taking advantage of the commodity trend of the mobile devices, in the same way that CMOS trend was adopted in the 90s in despite of vector machines. The heterogeneous ARM-nodes are comprised by CPUs and GPUs fused in a SoC approach. This low power architecture has the potential of becoming the leading edge trend in the future.

The early developments of the Mont-Blanc project using the Tibidabo prototype are reported in [5], it consists of a full description of the cluster deployment and summarizes the main implementation aspects for HPC. Other results based in porting common HPC kernels in the mobile processors are found in [6, 7]. These works present a detailed performance comparison between three mobile processors and a commodity CPU. On the other hand, the interconnection between low power nodes and its limitations is explained in [8]. The new mobile GPUs, supporting OpenCL, motivated the strategy of porting the same basic HPC kernels to the device [9], this work demonstrates the efficiency of the GPU with respect to mobile CPUs. The first attempt for porting CFD applications based on different mathematical models is described in [10], their study is focused in the scalability of the application when engaging up to 96 nodes using ARM Cortex A9 processors. A case of the Nbody algorithm developed in OpenCL for mobile GPUs is studied in [11]. However, all of this early implementations are based on using only one of the computing devices integrated in the modern mobile chips.

We were invited to test TermoFluids code in the Mont-Blanc prototypes. Termofluids software is a general purpose unstructured CFD code written in C++ utilized to perform simulations in different industrial areas such as: renewable power generation, thermal equipments and refrigeration. The algorithm for modeling of incompressible turbulent flows is based in a portable algebraic based implementation model explained in detail in [12]. As a result, the code is based on only the following three linear algebra operations: 1) the sparse matrix-vector product (SpMV); 2) the dot product; 3) the operation of the form $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$, so called "axpy" operation. This strategy have made the code naturally portable to a any architecture and the problem of porting the code is reduced just to switching between existing implementations of these operations. All the algorithmically complex preprocessing stage is implemented only for traditional CPUs in a object-oriented user-friendly way. This stage includes processing of the mesh geometry, construction of control volumes, calculation of the coefficients of underlying discrete operators, etc. On the other hand, the time integration phase relies on a restricted set of operations and is free from complex object-oriented overhead. It operates with a raw flattened data represented in common storage formats compatible with linear algebra libraries.

This article is devoted to the study of a heterogeneous implementation of the algebraic kernels used in our CFD implementation in the Mont-Blanc nodes. The strategy consist in the distribution of the working load for engaging the CPUs and GPUs of the ARM-based nodes. The auto-balancing approach proposed is based in a tabu search algorithm for obtaining the optimal distribution of work. OpenCL [13] is the programming model selected for exploiting the massive parallelism execution of the GPU and its implementation also considers the use of vectorized operations (SIMD). For the CPU, OMP [14] is responsible for launching the threads in the multicore processors. Communications are managed by means of by the Message Passage Interface (MPI) [15] in a domain decomposition approach. In addition, this multiple level execution model is studied in terms of net performance and power efficiency.

The rest of the article is organized as follows: Section 3.2 describes the math model and the numerical algorithm that leads to our algebraic based implementation; the details of the Mont-Blanc prototypes and the main considerations in the implementation are explained in Section 3.3; the SpMV implementation details and the auto-balancing hybrid algorithm for our algebraic kernels are proposed in Section 3.4; the numerical results showing the efficiency of our algorithm are presented in Section 3.5; finally, Section 3.6 is devoted to the main conclusions of our work.

3.2 Governing equations and implementation model

The simulation of a turbulent flow of an incompressible Newtonian fluid is considered. The flow field is governed by the incompressible Navier-Stokes equations written as:

$$\nabla \cdot \mathbf{u} = 0, \tag{3.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \Delta \mathbf{u} \tag{3.2}$$

where $\mathbf{u}$ is the three-dimensional velocity vector, p is the kinematic pressure scalar field and ν is the kinematic viscosity of the fluid.

TermoFluids code has been adapted for facilitating the adoption of new technologies. The time integration phase is based in a portable implementation model for DNS and LES simulation of incompressible turbulent flows explained in detail in Chapter 2. The main idea is substituting stencil-based data structures and kernels by algebraic storage formats and operators. The result it is that around 98% of computations are based on the composition of three basic algebraic kernels: SpMV, AXPY and DOT, providing a high level of modularity and a natural portability to

Step of Algorithm	SpMV	axpy	dot	extras
1 - predictor velocity	8	6	0	0
2.1 - Poisson equation (r.h.s)	3	1	0	0
2.2 - **Poisson equation (per iteration)**	**2**	**3**	**2**	**0**
3.1 - velocity correction	3	3	0	0
4 - velocity at faces	7	0	0	0
5 - eddie viscosity (optional)	9	0	2	1
6 - CFL condition	0	0	0	1
Total outside Poisson solver	**30**	**10**	**2**	**2**

Table 3.1: Number of times that each basic operation is performed in the numerical algorithm

the code. Among the three algebraic kernels the SpMV is the dominant one, it substitutes the stencil iterations and the non-linear terms, such as the convective operator, are rewritten as two consecutive SpMVs. Table 3.1 sums up the number of times that each kernel is called at the different steps of the fractional step algorithm.

3.3 Mont-Blanc Prototypes

3.3.1 Architecture of a Prototype

The Mont-Blanc project is an European initiative that aims at developing the basis for an exascale supercomputer based on power efficient embedded technologies. The prototype nodes are composed by ARM system-on-chip (SoC) technology adapted from the mobile industry. The idea is to profit the impulse of the mobile market mainstream trend and redirect its efforts into the development of energy efficient supercomputers. The components of the mobile chip that are useless for high performance computing (bluetooth, wireless, etc.) are removed with the intention to reduce as much as possible the power consumption. The primary components of the Mont-Blanc prototypes are the following:

- **ARM Cortex-A15 CPU:** The cortex-A15 is a dual core CPU designed with advanced power reduction techniques. This processor is based on the Reduced Instruction Set Computing (RISC), requiring significantly fewer transistors than traditional computers, and reducing costs, power and heat. The processor provides a highly out-of-order processing engine with a 15 stage pipeline, enabling a high level instruction parallelism. In addition, the memory hierarchy is defined by 32KB L1 instruction cache and 32KB L1 data cache per core, moreover both cores share a 1MB L2 cache. In addition, the ARM Neon SIMD engine is incorporated for accelerating the single-precision computations because it can perform the same instruction on multiple sets of 128-bit wide in parallel. Unfortunately, our CFD cases demand double-precision and it is impossible to profit this engine for the moment.

- **ARM Mali T604 GPU:** This ARM GPU is composed by four shader cores with two arithmetic pipelines per core. Mali-T604 pipelines provides the standard IEEE double-precision floating-point math in hardware. This is an important characteristic for providing accuracy and correctness when running CFD simulations. Moreover, each core is capable of supporting up to 256 active threads, and as in the discrete GPUs it is necessary to keep active a sufficient number of threads in order to hide the instruction latency. Arithmetic logical units consists of 128-bit vector registers allowing the concurrent execution of 2 double-arithmetic operations. The main differences from discrete GPUs are in the memory layout. Firstly, both cache and global memories are located physically in the main memory of the device, and consequently its accesses have the same transfer costs. Secondly, the device memory is physically shared with the CPU memory, sharing the same memory bandwidth. This results in ideal conditions for heterogeneous implementations because the PCI express bottleneck is not present.

- **Network:** Commodity mobile SoC are not designed with network interfaces for working in a cluster environment. As a consequence, the interconnection between ARM nodes becomes a major issue when implementing a mobile-based supercomputer. The Mont-Blanc team developed a way of bypass this problem by creating a network interface on the USB port that allows to connect the Mont-Blanc nodes through a Gigabit Ethernet network. However, this network runs several times slower than the current standard in the supercomputers.

3.3.2 Programming Model

As mentioned in Section 3.2, our code is manly composed by three algebraic kernels: SpMV, AXPY and DOT. These memory bounded functions offer the potential to profiting different execution models. Consequently, we have considered an implementation in which the following frameworks coexists:

- **OpenMP:** The multicore ARM CPUs are utilized by means of a shared memory multiprocessing approach [14], suitable for avoiding data replication in this low memory prototypes. Enough OMP-threads are launched for fully engage all the CPU-cores.

- **OpenCL:** The Mali GPU architecture is conceived to support OpenCL 1.1 programming standard [13]. The stream processing model is used for creating, scheduling, managing and launching the thousands of OpenCL-threads. The memory latency is hidden by context switching, the idea is keeping active the maximum number of threads maximizing the memory throughput. The programmer assumes the responsibility of creating efficient execution paths and memory footprints. This framework also permits the execution of SIMD operations within the OpenCL-threads for profiting the 128-bit vector registers of the ARM-GPU. Further implementation details for running applications in mobile devices with OpenCL can be found in [16].

- **MPI:** The interconnected nodes require a distributed memory approach based in a geometrical domain decomposition. The message passing interface (MPI) standard is the most mature programming framework for engaging multiple nodes. In our case, the number of MPI tasks is equal to the number of ARM nodes engaged. Then, each task performs the successive calls to OMP and OpenCL kernels in a concurrent mode.

3.3.3 ARM-nodes main considerations

The architecture of the Mont-Blanc nodes differs in many aspects from the high-end nodes of the traditional supercomputers. The hardware characteristics requires specific considerations when implementing the main kernels of our CFD code. The main features that impact the performance are:

- **Compilation:** An important aspect for tunning the execution of our CFD kernels is to provide the proper compilation flags. The recent versions of GNU compilers `gcc, mpic++` contain the necessary extensions for ARM-based compilation. First of all, we must define the architecture we are working with by using the flag `-mcpu=cortex-a15`. Double precision is specified by the flag `-mfpu=vfpv4` and to explicitly allow its execution the flag `-mfloat-abi=hard` must be added as well.

- **Unified memory accesses:** The main advantage of the ARM SoC architecture is that allows the interaction between host and device by means of a unified memory space. This feature is already supported in the discrete GPU devices, but it is more profitable in the fused CPU+GPU architecture. In such devices the memory is physically shared between host and device, and memory bandwidth is the same for both computing units. As consequence, the overhead of memory transfers through the PCI-express is avoided. This characteristics are important in the development of the kernels because it makes attractive the use of concurrent execution between computing units.

The unified memory space is a region of memory (Buffer) that is accessible from OpenCL-threads and OMP-threads. Note that this memory can not be allocated by means of the standard `malloc` operation, because in that case OpenCL-threads could not access it. The solution consist in allocate buffers by means of the OpenCL API using the `clCreateBuffer` function. This operation requires the definition of a parameter that describes the type of usage of the buffer. The best option for concurrent execution algorithms is using the flag `CL_MEM_ALLOC_HOST_PTR`. By doing so, a zero-copy mode is established when accessing the buffer from the CPU. This means that OMP-threads do not create local copies of the memory region, but operate directly on it. The access from OpenCL kernels is straightforward by using the pointer obtained from the buffer definition. However, a memory mapping function needs to be used for enabling the access to a unified memory region from CPU. The `clEnqueueMapBuffer` and `clEnqueueUnmapMemObject` functions are used controlling the accesses from the OMP-threads.

- **Vectorization:** Both computing units, CPU and GPU, support vector registers for increasing the throughput of the algorithms. In the case of the CPU, the NEON Advanced SIMD is the engine that provides vectorization functionalities. However, it was designed to accelerate media and signal processing functions of up to 32-bit registers and therefore it can not be used to perform double precision arithmetics. Each ARM Mali GPU core contains 128-bit wide vector registers. This allows to perform SIMD operations within each OpenCL-thread. In particular, up to two double precision operations can be executed at once. Rather than using automatic vectorization, we utilize the vector types (`double2`) provided by the OpenCL API. This permits us controlling the sections of the code that exploit the vector registers.

The implementation of our algebraic CFD kernels also considers some more general aspects that are not unique from the ARM-based nodes. In particular, the implementation details explained in Chapter 2 are suitable for the development of kernels that profit the stream processing. The main aspects adopted are the following:

- **Occupancy and Thread Divergence:** As in the discrete GPUs, the ARM Mali necessitates a sufficient number of concurrent threads for hiding the execution latency of instructions. The number of active threads is limited by physical resources (registers and shared memory), and by logical implementation (threads, blocks). Therefore, our algebraic kernels are based on utilizing the minimum number of registers per threads, aiming at keeping active the maximum threads (256) per shader core.

- **Memory aware programming:** The maximum performance in the streaming processing model combined with the SIMD instructions can only be achieved by using appropriate data arrangements. The AXPY and DOT kernels are dense vector operations that do not need special data structures. On the contrary, the SpMV performance is determined by the storage format selected. In this case, we use a sliced ELLPACK format presented in [17]. Memory is accessed following coalescing and alignment rules facilitating the memory locality. This data structure has proven to be an efficient way for handling the laplacian matrices arisen from the discretization of the Navier–Stokes equations.

3.4 SpMV implementation details

The main algebraic kernels of our implementation model are the SpMV, AXPY and DOT product. The modularity and portability of our approach aims at profiting the existent libraries that provide a straightforward implementation. However, these libraries are not tuned for working with the fused CPU+GPU architecture of the Mont-Blanc nodes. We focus our attention in the development of an optimized SpMV for the ARM nodes. As commented in Section 3.3.3, the sliced ELLPACK format is the most appropriate format for the matrices arisen from the discretization on unstructured meshes. On the other hand, AXPY and DOT operations are independent of the application context, exploiting the same enhancements that are explained for the SpMV.

3.4.1 Vectorized SpMV

The single-CPU implementation is straightforward and can be found in [17]. On the other hand, the single-GPU requires extra tunning for profiting the 128-bit wide registers of the mali GPU. By doing so, the algorithm is capable to perform two double operations at the same time. The vectorization of the algorithm is implemented by means of the OpenCL vector data types: `double2` and `int2`. The load, store and arithmetic operations are overridden by the OpenCL compiler, implicitly calling SIMD instructions when working with vector data types. Note that the arithmetic intensity of the SpMV can not be improved by vectorization because there is not data reuse. However, the memory accesses of the matrix elements and the output vector can gain performance by being executed with SIMD instructions. The reason is that SIMD store/load instructions bring into memory two memory addresses at once in a single memory transaction. This results in a better harness the memory bandwidth. Therefore, the strategy for the vectorization consists in double the workload per work item. Each OpenCL-thread is responsible of computing two rows of the matrix, instead of one as in traditional GPU implementation. By doing so, matrix elements are read in pairs making suitable the use of vectorization. The final vectorized algorithm is shown in Kernel 3.1.

```
ELL_SPMV_VECT(global double2* Val, global int2* Index /*matrix*/,
  global double* b /*input vector*/, global double2* x /*output vector*/,
  int m/*half number of rows*/, int K/*number of non-zeros per row*/)
{
  int row= get_global_id(0);
  int offset= (row -get_local_id(0))*K;
  if (row >= m) return;
  double2 dot=(double2)0;
  int2 cols;
  double2 xs;
  for(int i=0;i<K;i++){
              cols=Index[get_local_id(0) + offset+ i*get_local_size(0)];
              xs.x=x[cols.x];
              xs.y=x[cols.y];
              dot+= Val[get_local_id(0) + offset+ i*get_local_size(0)]*xs;
  }
  y[row]=dot;
}
```

Kernel 3.1: sliced ELLPACK kernel implementation

In an analogous way, the same approach of increasing the thread workload is applied for creating vectorized version of the AXPY and DOT operations.

3.4.2 Auto-balance of heterogeneous implementations

Heterogeneous computing is of major importance for making competitive the infrastructures based in mobile technology. As explained in Section 3.3.3, the idea of using all the computing units of the Mont-Blanc nodes is conceived by the uniform memory model supported by a physically shared memory. The balancing algorithm consist in finding the best workload partition to be processed by each computing unit. The total time of execution can be estimated by measuring the independent execution in each computing unit as $T = max(T_{GPU}, T_{CPU})$. However, this first approximation of the execution time is not always correct because is based in an independent performance of the computing units. Note, that performance in each device is complex to be predicted without executing the kernels, because it depends on how effectively works the cache in the CPU, or the occupancy hides the instruction latency in the GPU. Moreover, the ARM nodes are configured with a frequency downscaling system for avoid overheating after intense computation periods. As a result, under stress situations CPU performance downgrades and negatively affects the work balance. As we will see in Section 3.5, the independent analysis of the CPU/GPU kernels may lead to not optimal configurations, for this reason it is better to base our calculation of T in measuring the concurrent execution of the kernels.

The complexity of finding, a priori, an optimal distribution makes necessary the use of an iterative process. Our balancing meta-heuristic is based in a tabu search strategy [18], in which the target function consist in minimizing the concurrent execution time described by $T(x)$. Let us consider the function that describes the total execution time of the distributed algorithm defined as $T(x) : [0, N] \mapsto \mathbb{R}^+$,where N is the number of rows. The work balancing consists in finding the number $m \in [0, N/2]$ such that minimizes T. Given a m, the distribution of workload is defined as: elements within the range $[0, 2m]$ are processed by the GPU, and the remaining elements $[2m + 1, N]$ are handled by the CPU. For practical purposes we have established the parameter $r = 2m/N$ that represents the ratio of elements in GPU. Our approach is described in Algorithm 1.

Algorithm 1 Auto-balancing algorithm

```
1: $m_0 \leftarrow$ initial solution
2: calculate time $T(m_0)$
3: $m_{opt} \leftarrow m_0$
4: $T_{opt} = T(m_0)$
5: tabuList[ ] $\leftarrow push(\{m_0\})$
6: candidates[ ] $\leftarrow push(\{m_0 + \Delta m, m_0 - \Delta m\})$
7: **while** candidates[ ] $\neq \phi$ **do**
8:     $m_i = pop(candidates[\,])$
9:     tabuList[ ] $\leftarrow push(\{m_i\})$
10:    calculate time $T(m_i)$
11:    **if** $T(m_i) < T_{opt}$ **then**
12:        $m_{opt} \leftarrow m_i$
13:        $T_{opt} = T(m_i)$
14:        candidates[ ] $\leftarrow push(\{m_i + \Delta m, m_i - \Delta m\} - tabuList[\,])$
15:    **end if**
16: **end while**
```

The algebraic based CFD kernels of our implementation model are memory bounded, which means its performance is dominated by the memory transactions rather than the computations. Therefore, we can estimate an initial distribution, m_0, based in the memory bandwidth (BW) of the computing unit and the total number of elements (N), such as:

$$\frac{2m}{BW_{GPU}} = \frac{(N - 2m)}{BW_{CPU}} \tag{3.3}$$

therefore,

$$m = \frac{BW_{GPU}}{2(BW_{GPU} + BW_{CPU})} N \tag{3.4}$$

In Mont-Blanc nodes, the bandwidth of both computing units is the same, consequently the initial distribution consist in distribute an equal amount of work for each computing unit or $m = N/4$. The tabu search requires the definition of a *step function* which determines the candidate movements for the next iterations. When a distribution m is selected, it is also added in the `tabulist[]`. This list represents the previous configurations that have been tested and it is used for avoiding repetition in the movements. Then, since our search is performed in a one-dimensional space $[0, N]$, our *step function* consists in $\pm\Delta m$ such as the new candidates are not in the `tabulist[]`. The function is defined as:

$$\Delta m = n \times bsize \tag{3.5}$$

where *bsize* is the number of elements processed by an OpenCL work-group, and n is a constant integer parameter that can take values in the range of $[1, \frac{N}{bsize} - 1)]$. The parameter n determines the granularity of the auto-balancing algorithm. A small size of n spans a larger search space allowing a more precise partition, but is more likely to be stuck in local minima. On the other hand, a big n avoids local minima at expenses of precision since the search space is more coarse. Note that those parameters may not be defined as constants, but for simplicity purposes we avoid introducing more complex strategies that may have a bigger overhead in the algorithm. Once the possible candidates are obtained, they are *push* in a stack called `candidates[]` that operates in a LIFO (last in, first out) arrangement. This implementation detail is necessary for being able of following positive tendencies, since it privileges the selection of candidates arisen from the last movement that improved the balancing.

The algorithm can also include a restarting procedure, in which the auto-balancing starts from a different initial solution, in doing so, it avoids the stagnation in local minima. This balancing strategy is applied during the preprocessing stage of the CFD algorithm until an optimum solution is reached. Note that each algebraic kernel may have a different distribution since the individual performance depends on how much of the bandwidth can be profited by the function. The overhead introduced by the auto-balancing method is negligible because the CFD algorithms need $\mathcal{O}(10^5 \sim 10^6)$ of iterations for converge.

3.4.3 Multi ARM SoC SpMV implementation

The underlying MPI parallelization of the present unstructured CFD code is based on a domain decomposition approach. Therefore, the initial mesh $\mathcal{M}$, is divided into P non-overlapping submeshes $\mathcal{M}_0, ..., \mathcal{M}_{P-1}$ by means of a graph partitioning tool. For each MPI process, the corresponding unknowns of the system can be categorized into different sub-sets for a given i-th subdomain:

- *owned* unknowns are those associated to the nodes of $\mathcal{M}_i$;
- *external* unknowns are those associated to the nodes of other submeshes;
- *inner* unknowns are the owned unknowns that are coupled only with other owned unknowns;
- *interface* unknowns are those owned unknowns which are coupled with external unknowns;
- *halo* unknowns are those external unknowns which are coupled with owned unknowns.

To facilitate the communication schemes a block reordering is applied in order to group inner, interface and halo elements. By doing so, the interface rows are stored contigously in memory, minimizing the indirect accesses and the indexes for identify them.

Therefore, the communication episodes, involved in the halo update, consist only in the non-blocking point-to-point MPI communications `MPI_Isend`, `MPI_Irecv`, `MPI_Waitall`. Similarly than in high-end hybrid nodes, the communication costs can be partially hidden by using overlapping strategies.

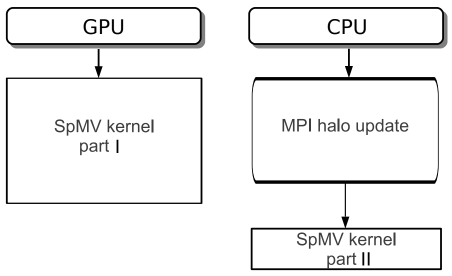


Figure 3.1: Execution model OSpMV with adapted distribution

The overlapping SpMV (OSpMV) is based in our previous experience working with discrete CPU/GPU systems [19]. However, certain modifications need to be introduced in the execution model for adjusting it to the characteristics of the ARM-based nodes. The main change consist in performing part of the calculations in the CPU. This is suitable in the Mont-Blanc nodes, because in the unified memory space it is not necessary to execute additional data transfers from CPU to GPU.

The tabu search algorithm is utilized for determining the optimal workload. In the parallel version, the algorithm considers as CPU-work both MPI communications and SpMV computations. The minimal load of the SpMV for the CPU is the interface part of the matrix, because that part can only be operated after the communication process is finished.

The remaining kernels do not need to introduce changes, since the load distribution within the nodes is not affected by the parallelization process. In AXPY there are not communications involved so the results are the same than operating in a single-node mode. For DOT product the parallel reduction is performed by `MPI_Allreduce`, and since this operation can not be overlapped there is no need to redistribute the workload.

3.5 Numerical Experiments

The three main algebraic kernels that compose our CFD algorithm are the AXPY, DOT and SpMV operations. The performance of this algebraic kernels has been measured and analyzed independently for each computing device. Moreover, a hybrid implementation that combines CPU and GPU execution has been tested as well. The benchmark case known as ASMO car was selected for testing our kernel implementation. The test case consist in simulating the turbulent flow around the ASMO car geometry (Re = 7×10^5) using a 5.5M unstructured mesh, the implementation details can be found in Chapter 2. The relative weight of the operations is shown in Figure 3.2. As expected, the profiling demonstrates that SpMV is the dominant operation with 82% of contribution to the algorithm. This result is in agreement with Table 3.1 that shows the number of SpMV calls during our fractional step. On the other hand, the three basic algebraic kernels sum up to 98% of the time-step execution time. This fact validates our approach of focusing our attention in the development of the main algebraic kernels. Note that we are not running any simulation in Mont-Blanc nodes, but we have used the estimation of performance proposed in Chapter 2. The numerical experiments have been carried out on the Mont-Blanc prototypes described in Section 3.3. The code has been implemented adopting an execution model in which three different frameworks coexist: OpenCL, OpenMP and MPI.

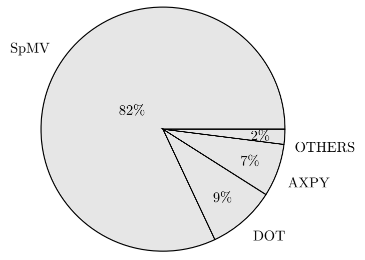


Figure 3.2: Relative weight of the algebraic kernels in a CFD simulation

3.5.1 Single node results

First of all, the net performance of the main kernels was tested for different workloads in the range of a traditional CFD application. The Mont-Blanc prototypes are equipped with only 1GB of RAM memory per node, limiting the size of the problem that fits on them. In the case of SpMV, the laplacian matrix and the sliced ELLPACK format have been chosen as the most appropriate for our application context.

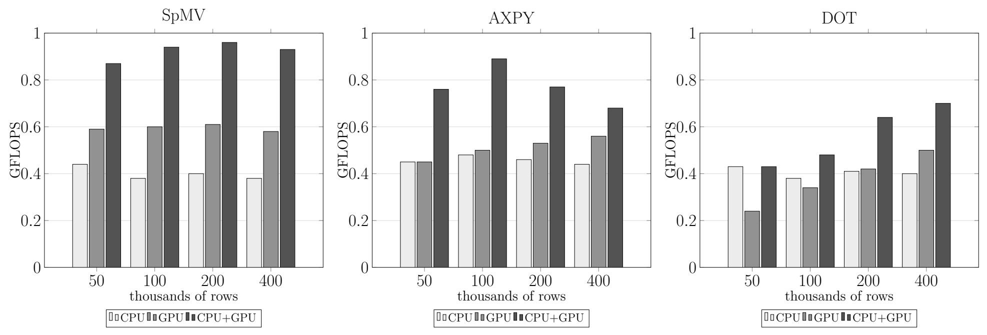


Figure 3.3: Net performance of the SpMV, AXPY and DOT respectively

The net performance and the benefits of the hybrid approach are depicted in Figure 3.3, the results are compared with respect to the single-CPU and single-GPU execution. Notice that the kernels are memory bounded, and in this ARM-nodes the memory bandwidth is the same for the CPU and the GPU (physically shared memory). Therefore, it is expected to obtain about the same net performance in the single execution for each computing device. However, this trend is not totally in agreement with the plots for different reasons in each kernel.

For the SpMV, the discrepancy in the results is explained by the different execution models of CPU (MIMD) and GPU (stream processing + SIMD). Our conclusion is that the stream processing model of the GPU harness more efficiently the bandwidth as explained in detail in Chapter 2. In fact, the ARM-CPU memory hierarchy consist only in a two level small size cache (L2 cache is 1MB), resulting in an optimal bandwidth achievable for unrealistic small workloads. On the other hand, the ARM-GPU requires less active threads for achieving maximum occupancy, in comparison with the discrete GPUs. Both issues explain why the net performance of the device is nearly constant when increasing the workload. The average speedup of the GPU with respect to the CPU for the SpMV is $1.5\times$ in such meshes.

The AXPY and DOT product are vectors operations that are based in coalesced memory accesses. In fact, all the bandwidth of the CPU can be harnessed because of the good memory locality of those kernels. Therefore, the CPU can achieve nearly the same net performance than the GPU.

For the dot product the main difference is that is composed by a reduction operation. In the case of the GPU, this reduction operation is performed by means of allocating a shared memory that can be read by all the threads within a block. As difference of discrete GPUs, the shared memory in ARM-GPUs is not located in the streaming multiprocessors, but in the the global memory. Therefore, the reduction operation penalizes the GPU execution. The net performance in this case only outperforms the CPU when the workload is large enough (200K and 400k). In such cases, the relative weight of the reduction operation is smaller with respect to the total computation time. For those large cases, the GPU achieved an average speedup of 1.15× with respect to the CPU execution.

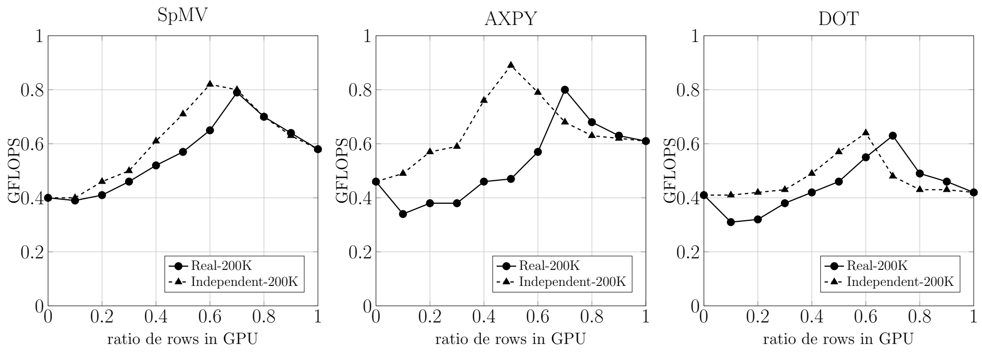


Figure 3.4: Auto-balancing for the kernels in the 200K case

Our auto-balance heterogeneous implementation of the kernels presented in Section 3.4.2 utilizes both devices concurrently, obtaining the best results for every case. The average speedups with respect to the best case of single execution for the SpMV, AXPY and DOT are 1.60×, 1.38× and 1.40× respectively. The work load is distributed between both computing devices by means of a tabu search algorithm. In optimal circumstances, when the performance in single-CPU and single-GPU are similar, the use of the hybrid algorithm should report ≈ 2× speedup. However, the best hybrid case obtained a speedup of ≈ 1.85×. In order to explain this, we decided to study the load balancing algorithm for the case of workload 200*K*.The results of the tabu search algorithm for the three kernels are illustrated in Figure 3.4. In the plots the x-coordinate denotes the percentage of workload in the GPU, 1.0 stands for fully GPU execution and 0.0 fully CPU execution. The dashed line represents the expected performance for the balancing algorithm based in the separated execution of the kernels in CPU and GPU. On the other hand, the continued line portraits the real performance obtained when both kernels are executed concurrently. Note that based in the separated execution, the SpMV and DOT optimal workload consist in assigning more elements to the GPU. While for the AXPY the optimal would be an equal distribution of workload for both computing unit. However, there is a disagreement when calculating the workload based the concurrent execution. In such case, for all kernels the optimal execution time is achieved when a larger portion of the work is executed in the GPU. This results in variation of the peak net performance, which it is smaller than the calculated based in separate execution. This behavior when running concurrent kernels is explained by the dynamic frequency scaling incorporated in Mont-Blanc prototypes. The ARM architecture is a technology developed for working without a cooling fan, therefore when a threshold of temperature is reached the CPU throttles back and the expected performance degrades. As consequence, the hybrid algorithm is forced to assign more rows to the GPU for finding a optimal solution. The concurrent execution of our hybrid algorithm propitiates the rises in the temperature of the chip because both devices are computing at the same time.

3.5.2 Parallel results

The main constrain of the mobile based supercomputers is the interconnection of the nodes. InfiniBand is not supported yet, therefore minimizing the impact of the communication episodes is highly important. In our CFD kernels, DOT product requires a collective communication that cannot be avoided; AXPY do not need communication at all; and the SpMV demands a point to point communication before operating the interfaces elements of the matrix. Therefore, for AXPY and DOT operations we can profit the same load balancing approach utilized when working in single node. On the other hand, for SpMV we focused our attention in the implementation of an overlapping strategy that hide part of the communications stage. This approach incorporates the weight of the communication time in the CPU when performing the tabu search for finding the optimal load balance. As a result, the new distribution consist in increasing the workload in the GPU whenever the relative weight of the communications get higher. The new ratio of rows is shown in Figure 3.5. The plot shows the new ratio for three mesh sizes 1.6M, 3.2M and 6.4 when engaging up to 64 nodes Note that in accordance with our overlapping model the ratio is limited to 1.0. This ratio represents executing the inner part of the matrix in the GPU, while the interface part is processed by the CPU. We can find a minimum number of nodes n when the ratio becomes 1.0. This means that when engaging n or more nodes, the cost of the communications is higher than the product of the inner matrix on the GPU. In the plot is shown that this is produced when working with a workload per node $\leq$ 200K. This inflection point is found when engaging 8,16 and 32 nodes for meshes of sizes 1.6M, 3.2M and 6.4M respectively

Next we compare our OSpMV with a non-overlapping SpMV implementation (NSpMV). The NSpMV consist in first executing the halo update in the CPU, and

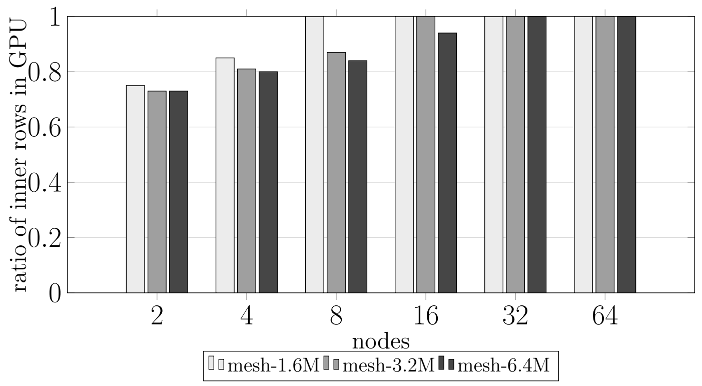


Figure 3.5: Distribution of rows for the OSpMV with different meshes

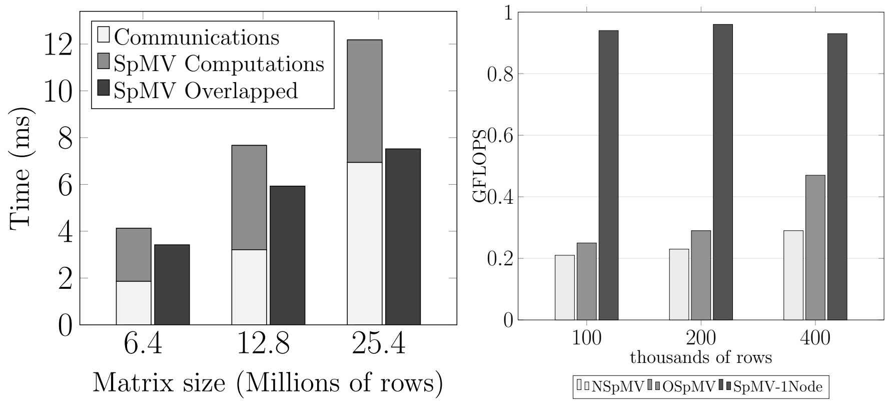


Figure 3.6: Left: Execution times of NSpMV and OSpMV. Right: Net performance of the NSpMV and OSpMV compared with single node SpMV

right after perform the hybrid approach of single node to all the matrix. Figure 3.6 (left) shows the execution time of both algorithms when engaging 64 nodes for three mesh sizes 6.4M, 12.8M and 25.4M. In such cases, the OSpMV obtains a speedup of 1.2×, 1.3× and 1.6× that can be explain by its capability of hiding part of the communication costs. The main difference with single node execution is that the CPU most of the time is performing communications instead of computations. In the right side of the figure, we compare the net performance of the parallel algorithms with the SpMV execution in a single node. In such case, the performance is degraded in 25%, 31% and 50% for the workloads 100K, 200K and 400K respectively. This demonstrates how important is the degradation produced by the communicaton in the Mont-Blanc nodes.

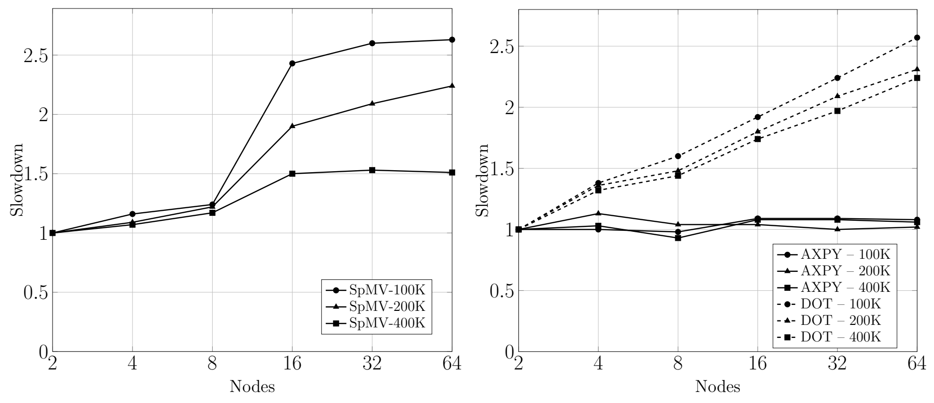


Figure 3.7: Weak speedup of the algebraic kernels. Left: SpMV , Right: AXPY and DOT

The scalability of the algorithm is analyzed by means of the weak speedup depicted in Figure 3.7. The test consist in keeping constant the workload per node while gradually engaging up to 64 Mont-Blanc nodes. Three different workloads in the range of a CFD simulation were selected: 100K, 200K and 400K. The overlapping strategy was chosen for the SpMV. In the cases running up to 8 nodes, the overlapping approach effectively hides part of the communication with a maximum slowdown of 1.3× in the performance with respect to the execution engaging 2 nodes. For larger cases, the domain decomposition introduces an overhead due to the partition of the mesh. In such cases, it is more likely to find sub-domains which are totally surrounded by other domains. Consequently, the halos and interfaces nodes of such sub-domains are larger and slowdown the whole communication episode. Note that this negative effect of the domain decomposition is a constant overhead that is observed when working with 16 nodes or more. Such negative issue is minimized when using a larger workload per node, because the relative weight of communications decreases. In any case, the overlapped SpMV algorithm scales nearly linear before and after this leap. The maximum slowdowns when engaging 64 nodes for the workloads of 100K, 200K and 400K per ARM-node are 2.6×, 2.24× and 1.53×, respectively. In the right part of the Figure 3.7 is shown the scaling of the AXPY and DOT kernels. As expected, AXPY shows a perfect scaling since the absence of communications, and DOT performance is characterized by a nearly linear degradation produced by the collective communication.

The parallel efficiency of the strong speedup for the algebraic kernels is shown in Figure 3.8. There different meshes were tested 1.6M, 3.2M and 6.4M. The meshes remain constant while gradually engaging up to 64 Mont-Blanc nodes. Results are in agreement with the observations made for the weak speedup. In contrast to discrete GPU execution, for the SpMV the occupancy effects do not play an important role here because its performance is nearly identical for the workload tested, as shown in Figure 3.3. Therefore, we can conclude that the main drawback of the prototypes are the communication episodes. Good scaling results are shown when engaging up to 8 nodes where an 80% efficiency is attained. From 16 nodes an up, the negative effects of the communications produce a rapidly decrease in performance. As a result, in the best case scenario engaging 64 nodes achieves only 40% of the expected efficiency.

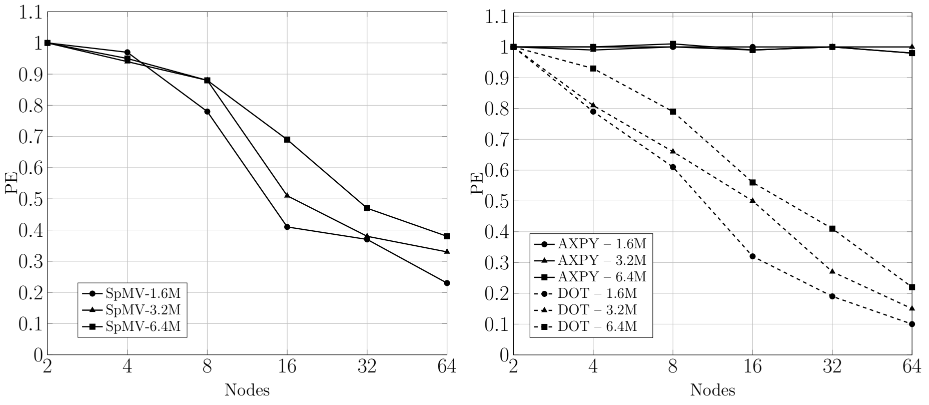


Figure 3.8: Parallel Efficiency of the algebraic kernels. Left: SpMV , Right: AXPY and DOT

In the future, the interconnection between ARM-nodes is expected to improve, and may be closer to the current supercomputers. For illustrating this point, we provide an estimation of the parallel performance if the nodes were interconnected by means of an InfiniBand QDR of 40 Gbps. Figure 3.9 shows that when working with the 6.4M mesh, the parallel efficiency engaging 64 nodes would be more than 80%. By doing so, the scalability of the prototypes would be as good as the traditional high-end supercomputers. However, we must consider that the ARM CPU+GPU runs up to $\approx 20\times$ slower than the Tesla GPUs, which are not the faster GPUs in the market. Consequently, for obtaining the same performance in an ARM-based cluster it would be necessary engaging up to 20 times more nodes. This would require a better parallel efficiency than the current high-end nodes.

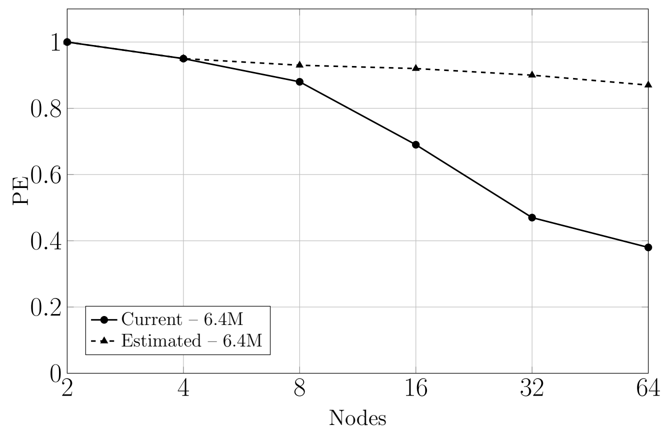


Figure 3.9: Parallel Efficiency of the simulation and estimation with better network

3.5.3 Energy efficiency

The main bottleneck in the development exascale computing systems is the energy cost. So, the FLOP per watt is a metric with major relevance when analyzing a HPC system. Such metric is used by the Green 500 list for indexing the top energy efficient supercomputer according to the LINPACK benchmark. With this in mind, we proceeded to compare the Mont-Blanc prototypes with a current high-end hybrid supercomputer. Minotauro is a 128-nodes hybrid CPU/GPU supercomputer. Each node is composed by two NVIDIA M2090 and two Intel hexacore CPUs. The nodes are interconnected by means of an InfiniBand QDR interface. Both implementation are based in our algebraic-based CFD code and properly tuned for the different architectures of the systems. The calculations have considered a 6.4M cells mesh, engagging up to 64 nodes. The net performance and the energy efficiency are shown in Figure 3.10.

Minotauro's net performance is in average $20\times$ better than the Mont-Blanc prototypes. This huge difference can be explained manly by three factors: discrete GPUs have $13\times$ faster bandwidth; Minotauro nodes does not suffer from frequency downscaling; and the better network infrastructure for connecting the nodes. A qualitative approximation of the FLOP per watt ratio can be obtained by considering the factory power requirements for each node. The estimation considers that a Mont-Blanc fused CPU+GPU consumes 5W, and Minotauro's node configuration with Xeon+M2090 requires 300W. Note that by doing so, we are not counting the network or refrigeration consumption. The results are shown in the right part of the figure. Under such circumstances, the Mont-Blanc nodes are up to $2.4\times$ more efficient than Minotauro

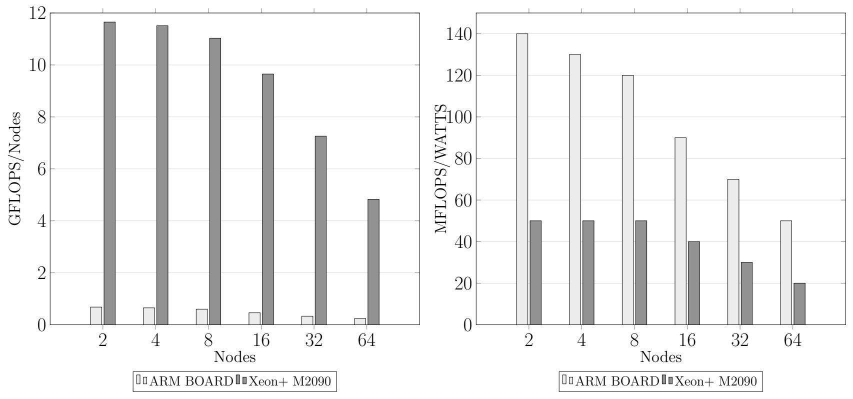


Figure 3.10: The net performance comparison between both implementations

nodes, even considering the limiting factors aforementioned. The main reason is that the ARM CPU+GPU architecture is developed for using reduced power supply in order to increase the battery usage of the mobile devices. Therefore using the same number of nodes on both systems, we conclude that Mont-Blanc would be more efficient, but the results would take $20\times$ higher execution time. This shows us the potential of the utilization of ARM technology in HPC systems. Note that the advances in this low power processors are estimulated by strong market trends. Therefore, future improvements in this technology are expected, encouraging us into continuing exploring this line of investigation.

3.6 Conclusions

High performance computing is exploring new energy efficient technologies aiming to reach the exascale paradigm. In this chapter we present a study of the performance of TermoFluids code on the Mont-Blanc mobile-based prototypes. Our solution is based in the specific implementation of the portable model proposed in Chapter 2. Note that scientific computing in such embedded architectures is still in its early stages, therefore there are not algebraic libraries that facilitate the implementation yet. Consequently, our focus was the implementation of the three main algebraic kernels that compose our computations: SpMV, AXPY and DOT. Such kernels are memory bounded operations, then its performance mostly depends of the memory bandwidth of the computing device where they run.

In Mont-Blanc prototypes, the CPU and GPU are equipped with the same memory bandwidth, so similar performance was expected for the kernels in sequential execution. However, the SpMV shows a 1.5× speedup in average of the GPU versus the CPU implementation. This confirms that the stream processing model of GPUs better harnesses the bandwidth for the SpMV as explained in detail in Chapter 2.

Our main contribution in the implementation of the kernels consists in a hybrid implementation of them for simultaneity enggaging both devices of the Mont-Blanc prototypes. This approach is possible because of two reasons: the physically shared memory of the mobile architecture that avoids PCI-e communication episodes between GPU and CPU, and the similar net performance of both computing devices that makes it worth the sharing. The problem consists in finding the best workload distribution to maximize the net performance. Our solution has been the development of an auto-balancing algorithm based in a tabu search that tunes the kernels workload in a pre-processing stage. Following this approach the SpMV, AXPY and DOT have been accelerated in average by 1.64× , 1.60× and 1.74×, respectively.

In the parallel execution, the load distribution includes the inter-node communication that is performed by the CPU. Therefore, the communications are overlapped with computations on the GPU that increases its workload to balance the cost of the communication performed by the CPU. Several parallel experiments have been performed with meshes of different sizes and engaging up to 64 nodes. The best parallel efficiency achieved on 64 nodes has been 40% for a mesh of 6.4M nodes. However the communication technology of the Mont-Blanc nodes is still based on the Gigabit network and we estimate that the performance would be much higher with an state of the art Infiniband network.

Summarizing, we found two main limitations for running CFD in the mobile architecture. Firstly, the slow network in comparison with the current supercomputers because Infiniband is not supported yet. Secondly, error correction code is not incorporated in the mobile chips, this makes unfeasible the execution of large scale simulations because the correctness of the results is not guaranteed. However, many of the aforementioned limitations were encountered in the early days of the GPU computing and were improved during the evolution of such technology. Finally, by means of a qualitative approximation we conclude that at running our CFD code the Mont-Blanc nodes are in average 2.4× more energy efficient than a current hybrid high-end supercomputer such as MinoTauro of the BSC. It is expected that mobile technology continues its evolution due to the strong market trends. Therefore, we consider that this technology has the potential of becoming an important part of the future supercomputing technology.

References

[1] J. Dongarra et al. The international exascale software project roadmap. In *International Journal of High Performance Computing Applications*, volume 25, pages

3–60, 2011.

[2] D.E. Keyes. Exaflop/s: the why and the how. In *Comptes Rendus Mécanique*, volume 339, pages 70–77, 2011.

[3] Ranking of the supercomputers according energy efficiency. Webpage: http://www.green500.org, 2015.

[4] Ramirez A. The mont-blanc architecture. In *International Supercomputing Conference (ISC 2012), Hamburg, Germany*, 2012.

[5] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez. Tibidabo: Making the case for an arm-based hpc system. *Future Generation Computer Systems*, 2013.

[6] Rajovic N., Rico A., Vipond J., Gelado I., Puzovic N., and Ramirez A. Experiences with mobile processors for energy efficient hpc. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 464–468, 2013.

[7] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero. Supercomputing with commodity cpus: are mobile socs ready for hpc? In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13). ACM*, volume 40, page 12, 2013.

[8] K. P. Saravanan, P. M. Carpenter, and A. Ramirez. A performance perspective on energy efficient hpc links. In *28th ACM international conference on Supercomputing (ICS '14)*, pages 313–322, 2014.

[9] Grasso I., Radojkovic P., Rajovic N., Gelado I., and Ramirez A. Energy efficient hpc on embedded socs: Optimization techniques for mali gpu. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 19–23, 2014.

[10] D. Goddeke, D. Komatitsch, M. Geveler, D. Ribbrock, N. Rajovic, N. Puzovic, and A. Ramirez. Energy efficiency vs. performance of the numerical solution of pdes: An application study on a low-power arm-based cluster. *Journal of Computational Physics*, 237:132–150, 2013.

[11] Ross J.A., Richie D.A., Park S.J., Shires D.R., and L.L. Pollock. A case study of opencl on an android mobile gpu. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6, 2014.

[12] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva. Stream processing based modeling for cfd simulations in new hpc infraestructures. *Journal of Parallel Computing*, 2015.

[13] Khronos OpenCL Working Group. The opencl specification version 2.0. Webpage: https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, 2014.

[14] OpenMP Architecture Review Board. Openmp application program interface version 4.0. Webpage: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

[15] Snir M., Otto S. W., Huss-Lederman S., Walker D. W., and Dongarra J. J. *MPI: The Complete Reference*. MIT Press Cambridge, 1995.

[16] Gronqvist J. and Lokhmotov A. Optimising opencl kernels for the arm mali-t600 gpus. ARM Tech Report, 2014.

[17] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. *High Performance Embedded Architectures and Compilers Lecture Notes in Computer Science*, 5952:111–125, 2010.

[18] F. Glovera. Tabu search - part 1. *ORSA Journal on Computing*, 2:190–206, 1989.

[19] G.Oyarzun, R. Borrell, A. Gorobets, and A. Oliva. Mpi-cuda sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. *Computers & Fluids*, 92:244–252, 2014.

4

Poisson solver for Peta-scale simulations with one FFT diagonalizable direction

Abstract. Problems with some sort of divergence constraint are found in many disciplines: computational fluid dynamics, linear elasticity and electrostatics are examples thereof. Such a constraint leads to a Poisson equation which usually is one of the most computationally intensive parts of scientific simulation codes. In this work, we present a memory aware auto-tuned Poisson solver for problems with one Fourier diagonalizable direction. This diagonalization decomposes the original 3D system into a set of independent 2D subsystems. The proposed algorithm focuses on optimizing the memory allocations and transactions by taking into account redundancies on such 2D subsystems. Moreover, we also take advantage of the uniformity of the solver through the periodic direction for its vectorization. Additionally, our novel approach automatically optimizes the choice of the preconditioner used for the solution of each frequency subsystem and dynamically balances its parallel distribution. Altogether constitutes a highly efficient and robust HPC Poisson solver.

4.1 Introduction

Divergence constraints are ubiquitous in physical problems. Under certain assumptions, they follow from basic conservation principles such as the mass conservation, the electrical charge conservation or the conservation of probability in quantum mechanics. Such a constraint leads to a Poisson equation for a some sort of scalar potential. Hence, it is not surprising that the Poisson equation plays a fundamental role in many areas of science and engineering such as computational fluid dynamics (CFD), linear elasticity, electrostatics and quantum mechanical continuum solvation models. Furthermore, it is usually one of the most time-consuming and difficult to parallelize parts of scientific simulation codes. Therefore, the development of efficient and scalable Poisson solvers is of great interest.

On the other hand, the sustained growth of the computing capacity of modern high performance computing (HPC) systems is given by the combination of two factors. Firstly, as the clock frequency is constrained by physical limitations the number of computing units (i.e. the concurrency) keeps increasing. This trend requires more and more scalable algorithms with higher degree of parallelism Secondly, driven by power limitations heterogeneous architectures have become popular in the last years. The heterogeneity is expressed at the node level with the introduction of massive parallel co-processors in addition to the hosting multicore CPUs. And at the CPU level with an increasing percentage of the net performance relying on vectorization, supported by wider vector lengths This requires adaptation of the algorithms to a heterogeneous (hybrid) architecture and even more complex parallel model that combines principally different kinds of parallelism. Namely, MIMD (multiple instruction multiple data) and SIMD (single instruction multiple data). In this context, we focus our attention on the development of a parallel Poisson solver flexible enough to run efficiently on different kind of parallel systems.

Regarding the applications, the present work is restricted to CFD problems. In particular, direct numerical simulations (DNS) and large-eddy simulations (LES) of incompressible turbulent flows. The following four aspects, which are also relevant in the context of this paper, are commonly present in many DNS/LES applications:

- The Poisson equation has to be solved repeatedly with different right-hand-side terms (for DNS/LES problems the number of time-steps can easily reach $\mathcal{O}(10^6)$), while the system matrix remains constant. Hence, a pre-processing stage with large computing demands can be accepted.
- Wall-bounded flows and/or flows around internal obstacles are common in most of the applications. Therefore, in order to solve all relevant turbulent scales near the walls, arbitrary unstructured meshes are required.
- The solution obtained in the previous time step(s) can be used as an initial guess for iterative solvers in order to accelerate the convergence.
- Periodicity in at least one direction is of interest in many cases.

For flows fulfilling the last property the Fourier diagonalization [1] in the periodic direction(s) is the best choice. The uniformity of the grid in each of such directions imposed by the method is suitable with the isotropic nature of the flow along it. Fourier diagonalization allows the original three-dimensional (3D) Poisson equation to decompose into a family of independent two-dimensional (2D) systems of equations. On this basis, several approaches can be adopted for (i) the parallelization strategy in the periodic direction and (ii) the choice of the parallel solver(s) for the 2D problems. Roughly speaking, their choice depend on the size of the problem and the computational architecture. Our successive adaptions have been mainly motivated by the irruption of new supercomputers with a clear tendency to increase the number of processing units without increasing but rather decreasing the amount of RAM memory available per core. For instance, the strategy adopted for small problems and reduced number of CPUs was a sequential approach in the periodic direction and a direct Schur-complement (DSD) based solver for the 2D frequency subsystems [2,3]. Then, for bigger problems it was necessary to adopt a hybrid strategy combining DSD for some frequencies with an iterative solver [4] for the others. This was mainly due to the RAM requirements of the DSD method. Alternatively, the range of applicability of the DSD could be extended by using an efficient parallelization in the periodic direction [5]. In both cases, scalability tests up to $\mathcal{O}(10^4)$ shown a good performance. These successive improvements in the Poisson solver led to the possibility to compute bigger and bigger simulations. Starting from the simulation of a turbulent air-filled differentially heated cavity at different Rayleigh

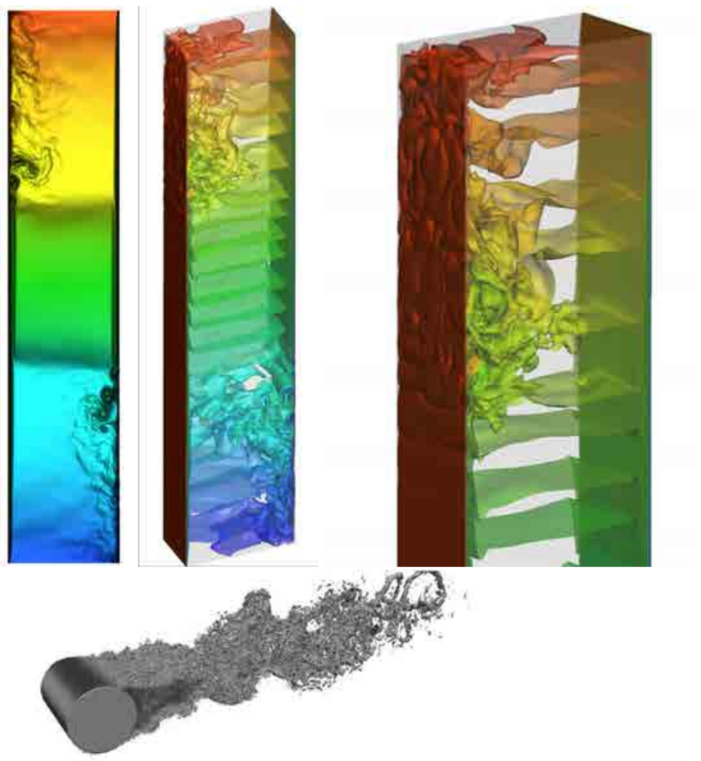


Figure 4.1: Examples of DNS simulations with one periodic direction. Top: turbulent air-filled differentially heated cavity [8,9]. Bottom: flow around a circular cylinder [13].

(Ra) number [6,7], many cutting-edge DNS simulations have been computed in the last decade. This initial works were carried out on a Beowulf PC cluster using up to 36 CPUs. Subsequent works [8, 9] were carried out on the first version of the MareNostrum supercomputer in the Barcelona Supercomputing Center (BSC). That time it was the number one European supercomputer and was ranked fourth in the Top500 list. These new DNS simulations were carried out using up to 512 CPUs (see Figure 4.1, top). More recent examples of DNS simulations can be found in [10] and, for unstructured meshes, in [11–13]. Figure 4.1 (bottom) displays the results presented in [13]: this simulation was carried out using up to 5000 CPUs on the MareNostrumIII in the BSC.

In this context, the present work proposes several improvements and adaptations of the algorithm for Peta-scale simulations on modern HPC systems. The most remarkable new features are threefold: (i) the algorithm has been evolved to a fully iterative mode in order to avoid memory constraints derived from the memory requirements of direct factorization methods. (ii) it optimizes the memory allocations and transactions by taking into account redundancies on the set of 2D frequency subsystems, (iii) it also takes advantage of the uniformity of the solver through the periodic direction for its vectorization and (iv) automatically optimizes the choice of the preconditioner used for the solution of 2D problems and dynamically balances its parallel distribution. Altogether constitutes a highly efficient and robust HPC Poisson solver.

The rest of the paper is arranged as follows. In Section 4.2, the discretization method is briefly presented. The basic ideas for the solution of Poisson systems derived from discretizations with one periodic direction are described in Section 4.3. The parallelization strategy is discussed in Section 4.4. The novel optimizations for the fully iterative version are presented in Section 4.5. In Section 4.6 are shown some numerical experiments performed on the Bluegene/Q Vesta supercomputer. Finally, relevant results are summarized and conclusions are given in Section 4.7.

4.2 Numerical methods

4.2.1 Problem geometry and topology

In this work, geometric discretisations obtained by the uniform extrusion of generic 2D meshes are considered. Periodic boundary conditions are imposed in the extruded direction. Consequently, the resulting linear couplings of the Poisson equation in such a direction result into circulant submatrices. Since the proposed algorithm do not impose any restriction for the initial 2D mesh it is suitable for unstructured meshes. Nevertheless, this lack of structure leads to a more complex data management. An illustrative example of such a geometric discretisation is displayed in Figure 4.2.

The following notation is used. The initial 2D mesh and the 1D uniform discretisation of the periodic direction are referred as $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$, respectively. The total number of nodes is $N := N_{2d}N_{per}$, where N_{2d} and N_{per} are the number of nodes in $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$, respectively. For simplicity, we assume that N_{per} is an even number. The constant mesh step in $\mathcal{M}_{per}$ is Δ_{per}. Two indexes define a node on the resultant 3D mesh $\mathcal{M}$, namely the positions in $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$. Hence, two different node orderings are used: the *2D-block-order* and the *1D-block-order*. They are lexicographical orders of the Cartesian products $\mathcal{M}_{2d} \times \mathcal{M}_{per}$ and $\mathcal{M}_{per} \times \mathcal{M}_{2d}$, respectively. Using

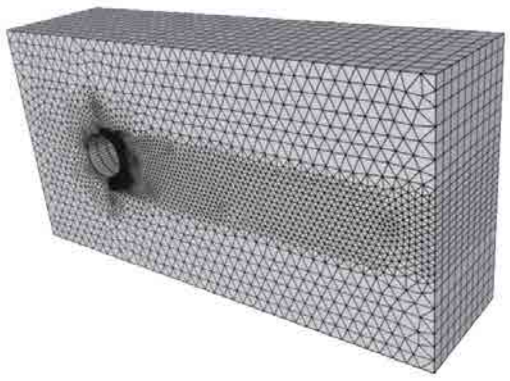


Figure 4.2: 3D mesh around a cylinder generated by the uniform extrusion of a 2D unstructured mesh.

the *2D-block-order*, a scalar field $v \in \mathbb{R}^N$ reads

$$v \equiv \left[v_0^{2d}, ..., v_{(N_{per}-1)}^{2d} \right], \tag{4.1}$$

whereas using the *1D-block-order* it becomes

$$v \equiv \left[v_0^{per}, ..., v_{(N_{2d}-1)}^{per} \right], \tag{4.2}$$

where $v_k^{2d} \in \mathbb{R}^{N_{2d}}$ and $v_k^{per} \in \mathbb{R}^{N_{per}}$ are the k-th *plane of the extrusion* and the k-th *span-wise* subvectors, respectively.

4.2.2 Poisson equation

The simulation of turbulent incompressible flows of Newtonian fluids is considered. Under these assumptions the velocity field, u, is governed by the Navier-Stokes (NS) and continuity equations

$$\begin{aligned} \partial_t u + (u \cdot \nabla) u - \frac{1}{Re} \Delta u + \nabla p &= 0, & (4.3) \\ \nabla \cdot u &= 0, & (4.4) \end{aligned}$$

where *Re* is the dimensionless Reynolds number. In an operator-based formulation, the finite volume spatial discretisation of these equations reads

$$\begin{aligned} \Omega \frac{d\mathbf{u}_h}{dt} + C\left(\mathbf{u}_h\right)\mathbf{u}_h + D\mathbf{u}_h + \Omega G p_h &= 0_h, \qquad (4.5) \\ M\mathbf{u}_h &= 0_h, \qquad (4.6) \end{aligned}$$

where $\mathbf{u}_h$ and p_h are the velocity and pressure fields defined in the nodes of the mesh $\mathcal{M}$, Ω is a diagonal matrix with the size of the control volumes, $C(\mathbf{u}_h)$ and D are the convective and diffusive operators and, finally, M and G are the divergence and gradient operators, respectively. In this paper, a "symmetry-preserving"/"energy conserving" discretisation is adopted. Namely, the convective operator is skew-symmetric ($C(\mathbf{u}_h) + C(\mathbf{u}_h)^* = 0$), the diffusive operator is symmetric positive-definite and the integral of the gradient operator is minus the adjoint of the divergence operator ($\Omega G = -M^*$). Preserving the (skew-)symmetries of the continuous differential operators when discretising them has been shown to be a very suitable approach for DNS [7,14,15].

For the temporal discretisation, a second-order explicit one-leg scheme is used. Then, assuming $\Omega G = -M^*$, the resulting fully-discretised problem reads

$$\begin{aligned} \Omega \frac{\mathbf{u}_h^{n+1} - \mathbf{u}_h^n}{\delta t} &= R\left(\frac{3}{2}\mathbf{u}_h^n - \frac{1}{2}\mathbf{u}_h^{n-1}\right) + M^* p_h^{n+1}, \qquad (4.7) \\ M\mathbf{u}_h^{n+1} &= 0_h, \qquad (4.8) \end{aligned}$$

where $R(\mathbf{u}_h) = -C(\mathbf{u}_h)\mathbf{u}_h - D\mathbf{u}_h$. The pressure-velocity coupling is solved by means of a classical fractional step projection method [16,17]. In short, reordering the equation (4.7), an expression for $\mathbf{u}_h^{n+1}$ is obtained,

$$\mathbf{u}_h^{n+1} = \mathbf{u}_h^n + \delta t \Omega^{-1} \left(R\left(\frac{3}{2}\mathbf{u}_h^n - \frac{1}{2}\mathbf{u}_h^{n-1}\right) + M^* p_h^{n+1} \right), \qquad (4.9)$$

then, substituting this into (4.8), leads to a Poisson equation for p_h^{n+1},

$$-M\Omega^{-1}M^* p_h^{n+1} = M\left(\frac{\mathbf{u}_h^n}{\delta t} + \Omega^{-1} R\left(\frac{3}{2}\mathbf{u}_h^n - \frac{1}{2}\mathbf{u}_h^{n-1}\right)\right), \qquad (4.10)$$

that must be solved once at each time-step.

4.2.3 Discrete Laplace operator

The Laplacian operator of equation (4.10),

$$\mathsf{L} = -M\Omega^{-1}M^*, \qquad (4.11)$$

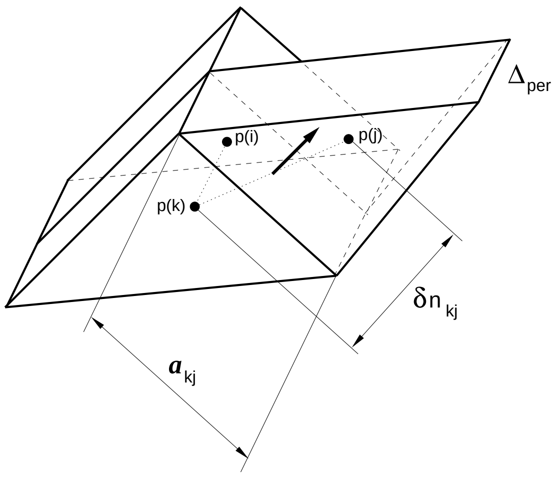


Figure 4.3: Elements of the geometric discretisation.

is by construction symmetric and negative-definite. Its action on p_h is given by

$$[\mathsf{L}p_h]_k = \sum_{j \in Nb(k)} A_{kj} \frac{p_h(j) - p_h(k)}{\delta n_{kj}}, \tag{4.12}$$

where $Nb(k)$ is the set of neighbors of the k'th node. A_{kj} is the area of f_{kj}, the face between the nodes k and j, and $\delta n_{kj} = |n_{kj} \cdot v_{kj}|$, where v_{kj} and n_{kj} are the vector between nodes and the normal unit vector of f_{kj}, respectively (see Figure 4.3). For details about the spatial discretization the reader is referred to [18].

The set $Nb(k)$ can be split into two subsets: $Nb(k) = Nb_{per}(k) \cup Nb_{2d}(k)$, where $Nb_{per}(k)$ and $Nb_{2d}(k)$ refer to the neighbor nodes along the periodic direction and in the same plane of the extrusion, respectively. In this way, the expression (4.12) becomes

$$\begin{aligned}[\mathsf{L}p_h]_k = &\sum_{i \in Nb_{per}(k)} A_{ki} \frac{p_h(i) - p_h(k)}{\Delta_{per}} \\ &+ \Delta_{per} \sum_{j \in Nb_{2d}(k)} a_{kj} \frac{p_h(j) - p_h(k)}{\delta n_{kj}},\end{aligned} \tag{4.13}$$

where a_{kj} is the length of the edge of f_{kj} contained in $\mathcal{M}_{2d}$ (see Figure 4.3). This can be written in a more compact form by means of the Kronecker product of matrices.

Using the *1D-block-order*, the Laplacian operator of the equation (4.13) reads

$$\mathsf{L} = (\Omega_{2d} \otimes \mathsf{L}_{per}) + \Delta_{per}(\mathsf{L}_{2d} \otimes \mathsf{I}_{N_{per}}), \tag{4.14}$$

where $\mathsf{L}_{2d} \in \mathbb{R}^{N_{2d} \times N_{2d}}$ and $\mathsf{L}_{per} \in \mathbb{R}^{N_{per} \times N_{per}}$ are the Laplacian operators discretised on the meshes $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$, respectively; $\Omega_{2d} \in \mathbb{R}^{N_{2d} \times N_{2d}}$ is the diagonal matrix representing the areas of the control volumes of $\mathcal{M}_{2d}$, and $\mathsf{I}_{N_{per}}$ is the identity matrix of size N_{per}. With the above-mentioned conditions (uniformly meshed periodic direction), L_{per} results into a symmetric circulant matrix of the form

$$\mathsf{L}_{per} = \frac{1}{\Delta_{per}} circ(-2, 1, 0, \cdots, 0, 1). \tag{4.15}$$

This allows to use a Fourier diagonalisation algorithm in the periodic direction. Note that this particular solution corresponds to a second-order finite volume discretization (see [18], for instance). However, the proposed algorithm relies on the fact that the Laplacian operator has the structure given in Eq.(4.14). This is the case for most of the existing numerical approximations of the Laplacian operator for problems with one periodic direction.

4.3 FFT based Poisson solver

In this section the basic ideas for the solution of Poisson systems with one Fourier diagonalizable direction are presented. As mentioned above, the problem under consideration reads

$$\mathsf{L}x_i = b_i \qquad i = 1, ..., N_t, \tag{4.16}$$

where the Laplacian operator, L, remains constant during the simulation and N_t is the total number of time-steps. Thus, the computational cost (per time-step) of any preprocessing stage is reduced by N_t times. Typically, for CFD applications applications $N_t = 10^5 \sim 10^6$; therefore, in general, the pre-processing costs become negligible.

Since the couplings in the periodic direction are circulant matrices, the initial system (4.16) can be diagonalized by means of a Fourier transform. As a result, it is decomposed into a set of N_{per} mutually independent 2D subsystems, drastically reducing the arithmetical complexity and the RAM memory requirements (see next subsection). Different aspect to be taken into account on the solution of the resulting 2D subsystems are considered in subsection 4.3.2.

4.3.1 Fourier diagonalisation

Any circulant matrix is diagonalizable by means of the discrete Fourier transform (DFT) of the same dimension [19, 20]. Then, the circulant matrix, L_{per}, defined in equation (4.15) verifies

$$\mathsf{F}^*_{N_{per}} \mathsf{L}_{per} \mathsf{F}_{N_{per}} = \Lambda, \tag{4.17}$$

where $\mathsf{F}_{N_{per}}$ and $\mathsf{F}^*_{N_{per}}$ are the N_{per}-dimensional Fourier transform and its inverse/adjoint, respectively; and $\Lambda = diag(\lambda_0, \lambda_1, ..., \lambda_{N_{per}-1})$ is the resultant diagonal matrix. A general expression for the eigenvectors can be found in [19, 20], in this particular case

$$\lambda_k = -\frac{2}{\Delta_{per}} \left(1 - \cos\left(\frac{2\pi k}{N_{per}} \right) \right) \qquad k = 0, ..., N_{per} - 1. \tag{4.18}$$

Then, if the unknowns are labeled adopting the *1D-block-order*, the operator $(\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})$ transforms all the *span-wise* subvectors, v_k^{per}, of any field, v, defined in $\mathcal{M}$, from the *physical* to the Fourier *spectral* space; $(\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}})$ carries out the inverse transformation. Applying the same change-of-basis to L, the Laplacian operator in the *spectral* space, $\widehat{\mathsf{L}}$, is obtained [1]:

$$\begin{aligned} \widehat{\mathsf{L}} &= (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}}) \mathsf{L} (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}}) \\ &= (\Omega_{2d} \otimes \Lambda) + \Delta_{per} (\mathsf{L}_{2d} \otimes \mathsf{I}_{N_{per}}). \end{aligned} \tag{4.19}$$

Comparing the last expression term-by-term with equation (4.14) it is observed that the change-of-basis affects only to the couplings in the periodic direction, whereas the couplings in the non-periodic directions are not modified. This is a consequence of the mesh uniformity in the periodic direction.Then, switching to the *2D-block-order*, $\widehat{\mathsf{L}}$ reads

$$\widehat{\mathsf{L}} = (\Lambda \otimes \Omega_{2d}) + \Delta_{per} (\mathsf{I}_{N_{per}} \otimes \mathsf{L}_{2d})) = \bigoplus_{k=0}^{N_{per}-1} \widehat{\mathsf{L}}_k, \tag{4.20}$$

where

$$\widehat{\mathsf{L}}_k = \lambda_k \Omega_{2d} + \Delta_{per} \mathsf{L}_{2d} \qquad k = 0, ..., N_{per} - 1. \tag{4.21}$$

Note that the matrices $\widehat{\mathsf{L}}_k$ only differ in the eigenvalue, λ_k, multiplying the diagonal contribution Ω_{2d}.

Therefore, the original system (4.16) is decomposed into a set of N_{per} mutually independent 2D systems

$$\widehat{\mathsf{L}}_k \hat{x}_k^{2d} = \hat{b}_k^{2d} \qquad k = 0, ..., N_{per} - 1, \tag{4.22}$$

[1]Given matrices A, B, C and D, with appropriate size, $(\mathsf{A} \otimes \mathsf{B}) \cdot (\mathsf{C} \otimes \mathsf{D}) = \mathsf{AC} \otimes \mathsf{BD}$

where each system, hereafter denoted as *frequency system*, corresponds to a frequency in the Fourier space. In summary, the process to solve the Poisson system is detailed in Algorithm 1.

Algorithm 1:

1. Transform the right-hand-side b, $\hat{b} = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})b$
2. Solve the the *frequency systems*, $\widehat{\mathsf{L}}_k \hat{x}^{2d}_k = \hat{b}^{2d}_k$
3. Restore the solution vector: $x = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}})\hat{x}$

At this point, some relevant issues must be addressed. Namely,

1. *Fourier decomposition of real-valued problems.* Since the subvectors b^{per}_k are real-valued, the corresponding discrete Fourier coefficients are paired as follows

$$\left[\hat{b}^{per}_k\right]_i = \left[\hat{b}^{per}_k\right]^*_{N_{per}-i} \qquad i = 1, ..., N_{per}-1. \tag{4.23}$$

Thus, the 2D subvectors $\hat{b}^{2d}_k$ meet

$$\hat{b}^{2d}_k = (\hat{b}^{2d}_{N_{per}-k})^* \qquad k = 1, ..., N_{per}-1. \tag{4.24}$$

On the other hand, the eigenvalues of the real-valued L_{per}, defined in equation (4.18), fulfill the following property

$$\begin{aligned} \lambda_0 &= 0, \\ \lambda_k &= \lambda_{N_{per}-k} \qquad k = 1, ..., N_{per}-1. \end{aligned} \tag{4.25}$$

Hence, plugging the two previous identities into equation (4.21) leads to

$$\begin{aligned} \widehat{\mathsf{L}}_0 &= \mathsf{L}_{2d}, \\ \widehat{\mathsf{L}}_k &= \widehat{\mathsf{L}}_{N_{per}-k} \qquad k = 1, ..., N_{per}-1. \end{aligned} \tag{4.26}$$

Finally, the last equation together with equation (4.24) imply that the solution of the *frequency systems* are paired as follows

$$\hat{x}^{2d}_k = (\hat{x}^{2d}_{N_{per}-k})^* \qquad k = 1, ..., N_{per}-1. \tag{4.27}$$

Therefore, recalling that N_{per} is an even number, the solution of $N_{per}/2-1$ of the *frequency systems* is directly obtained by taking complex conjugates.

2. *Complex systems*. The subvectors $\hat{b}_0^{2d}$ and $\hat{b}_{N_{per}/2}^{2d}$ are real-valued whereas the rest of subvectors $\hat{b}_k^{2d}$ have non-null imaginary components [19, 20]. This implies that the $N_{per}/2-1$ *paired* systems have a complex-valued solution. Nevertheless, since the coefficients of the matrices $\widehat{\mathsf{L}}_k$ are real, they can be solved as follows:
$$\widehat{\mathsf{L}}_k \left[\mathsf{Re}(\hat{x}_k^{2d}) \mid \mathsf{Im}(\hat{x}_k^{2d}) \right] = [\, \mathsf{Re}(\hat{b}_k^{2d}) \mid \mathsf{Im}(\hat{b}_k^{2d}) \,], \tag{4.28}$$
where $k = 1, ..., N_{per}/2-1$. Therefore, summing up, N_{per} real-valued 2D systems need to be solved in total.

3. *Fast Fourier Transform*. The inverse and forward Fourier transformations can be carried out by means of a FFT algorithm; here the implementation in [21] is used. This reduces the complexity of steps 1 and 3 of Algorithm 1 from $O((N_{per})^2 N_{2d})$ to $O(N_{per}\log_2(N_{per})N_{2d})$.

4. *Conditioning of the frequency systems*. The *frequency systems* are ordered by descending condition number. That is,
$$\kappa\left(\widehat{\mathsf{L}}_k\right) > \kappa\left(\widehat{\mathsf{L}}_{k+1}\right) \qquad k = 0, \cdots, \frac{N_{per}}{2}. \tag{4.29}$$
This follows from equation (4.21), and the ordering of the eigenvalues defined in (4.18): $0 \le i < j \le N_{per}/2 \Rightarrow 0 \le \lambda_i < \lambda_j$.

4.3.2 Solution of the frequency systems

Once the FFT algorithm has been applied, we must focus on the efficient solution of the set of decoupled 2D subsystems given in Eq.(4.22). In our previous work [5], we considered a direct Schur-complement based Decomposition method (DSD). Schur-complement based algorithms are non-overlapping decomposition methods [4, 22–25] that can be used for the parallel solution of linear systems. In particular, our DSD implementation is based on the definition of an interface subset of variables which decouples the different subdomains, the resulting decoupled local problems are solved by means of a Cholesky factorization and the interface system is solved by means of an explicit evaluation of its inverse, for a detailed explanation see [5]. Although the DSD algorithm has been successfully used to perform DNS simulations on different supercomputers, engaging up to 5120 CPU-cores and meshes with up to 600M nodes [13], there are some critical limitations to go beyond these figures.

The first and most important constraint is related to the RAM memory requirements: the size of the interface subsystem grows with the number of processes engaged on its solution and, as an explicit evaluation of the inverse is used to solve them, the memory requirements become unfordable. An approach to mitigate this problem consists in solving the most ill-conditioned frequency systems by means of the DSD method and using an iterative solver for the rest [4]. However, in the context of petascale simulations, engaging $O(10^5)$ CPU-cores, the DSD has to be scaled up to thousands of CPU-cores requiring unfordable memory resources. Apart from the memory problems, there are other issues such as the increasing cost of collective communication when P and N_{2d} increases that may prevent the efficient use of the DSD solver for large-scale problems.

Therefore, recalling the objective of extending the solver to petascale simulations, the DSD solver is replaced by a preconditioned Conjugate Gradient (PCG) [22]. The memory requirements are drastically reduced and, the scalability largely extended. Nonetheless, in lower scale simulations, the direct approach generally represents a faster and obviously a more robust solution.

The PCG algorithm is very well-known and can be found in [22], for instance. However, there are several features of the set of problems given in Eq.(4.22) that may affect the convergence of the algorithm. Namely, as mentioned in Eq.(4.29), the frequency systems are ordered by descending condition number, κ. The number of iterations needed to converge a Krylov-subspace method like PCG is closely related with κ. Well-conditioned systems (κ keeps close to unity) converge easily whereas they tend to degrade quickly when the system becomes ill-conditioned ($\kappa \gg 1$). Therefore not all the frequency systems will have the same computing cost so this must be taken into account when they are distributed among the parallel processes.

For example, in Figure 4.4 are shown, for different simulations of the flow around circular cylinder, the number of iterations required by the PCG solver as a function of the relative number of frequency, defined as

$$\xi(i, N_{per}) = \frac{2i}{N_{per}} \qquad i = 0, ..., \frac{N_{per}}{2}. \tag{4.30}$$

The lowest frequencies, which couple larger parts of the domain, require more iterations and thus more computing time. For further details on this study see [5]. All the optimizations developed for the new fully iterative approach are exposed in Section 4.5

4.4 Domain decomposition

The parallelisation of the solver is based on a geometric domain decomposition into P subdomains, one for each parallel process. The partition of $\mathcal{M}$ is carried out by

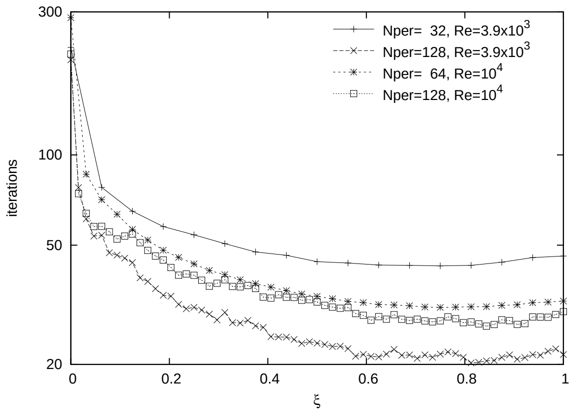


Figure 4.4: Number of iterations needed by the PCG depending on the relative number of frequency for different configurations of the flow around circular cylinder.

dividing $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$ into P_{2d} and P_{per} parts respectively, being $P = P_{2d}P_{per}$. This is referred as a $P_{2d} \times P_{per}$-partition. To exemplify it, a 2×2 and a 4×1-partitions of a mesh are displayed in Figure 4.5. The parallelisation of Algorithm 1 can be divided in two parts: (i) the parallelisation of steps 1 and 3, which are the change-of-basis from the *physical* to the *spectral* space and vice versa; and (ii) the parallelisation of step 2, which is the solution of the *frequency systems*.

Looking at step 1, the r.h.s of the *frequency systems*, $\hat{b}$, is given by

$$\hat{b} = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^{*}_{N_{per}})b. \tag{4.31}$$

Actually, this is not more than N_{2d} mutually independent Fourier transformations. Since a distributed memory parallelisation for the FFT is out of consideration, the *span-wise* component of the mesh is not partitioned. Thus, $\mathcal{M}_{2d}$ is divided into P subdomains and a $P \times 1$-partition of $\mathcal{M}$ follows. In this way, the *span-wise* subvectors of any field are not split between different processes, and a sequential FFT algorithm [21] can be used. An identical reasoning is applied to the change-of-basis from the *spectral* to the *physical* space, choosing the same $P \times 1$-partition. To obtain a balanced partition of $\mathcal{M}_{2d}$, the graph partitioning tool METIS [26] is used.

On the other hand, in the step 2 of the Algorithm 1, the solution of the *frequency systems* (4.28) is obtained as follows

$$\hat{\mathsf{L}}_k \hat{x}_k^{2d} = \hat{b}_k^{2d} \qquad k = 0, ..., \frac{N_{per}}{2}, \tag{4.32}$$

where a linear solver is used to solve each system. In this case, the $P \times 1$-partition chosen for the steps 1 and 3 can be sub-optimal if P_{2D} is too large according to the strong scalability of the linear solver being used for the frequency systems. Thus, partitions with $P_{per} > 1$ may be necessary to kept P_{2d} in the region of linear scalability of the linear solver. In this case, the N_{per} frequencies to be solved are divided into P_{per} subsets, and groups of $P_{2D} = P/P_{per}$ processes are used to solve the frequencies of each subset. The number of frequency systems contained in the k'th subset is referred as $N_{per,k}$. Note that in the iterative approach, in order to keep a balanced workload distribution, not all the subsets may have the same number of frequency systems because the frequency systems differ in the solution cost.

Therefore, since in general the optimal partitions for the change-of-basis (steps 1 and 3) and for the solution of the *frequency systems* (step 2) are different, two partitions are used in the parallelisation. As a consequence, two redistributions of data between those partitions are needed. Hence, the following algorithm replaces Algorithm 1:

Algorithm 2:

1. Evaluate $\hat{b} = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})b$ on the $P \times 1$-partition.
2. Redistribute $\hat{b}$ from the $P \times 1$- to the $P_{2d} \times P_{per}$-partition (collective comm.).
3. Solve the the *frequency systems*, $\hat{\mathsf{L}}_k \hat{x}^{2d}_k = \hat{b}^{2d}_k$, on the $P_{2d} \times P_{per}$-partition.
4. Redistribute $\hat{x}$ from the $P_{2d} \times P_{per}$- to the $P \times 1$-partition (collective comm.).
5. Evaluate $x = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}})\hat{x}$ on the $P \times 1$-partition.

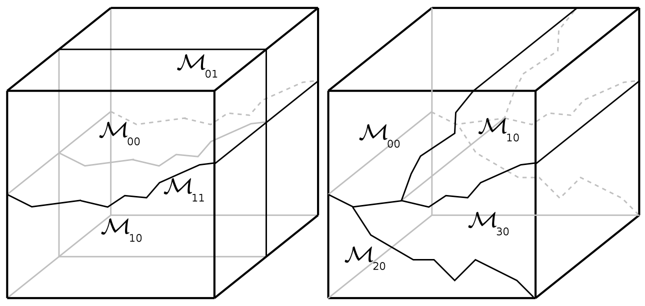


Figure 4.5: Illustration of a 2×2 (right) and a 4×1 (left) partitions of a mesh.

In order to simplify the redistributions of data (steps 2 and 4), a multilevel partition strategy is used. To do so, the P-partition of $\mathcal{M}_{2d}$, used in steps 1 and 5, is obtained from the P_{2d}-partition used in the step 3 by dividing each of its subdomains in P_{per} parts. An example is shown in Figure 4.5: the 2D subdomains in the right part, are directly obtained by splitting the 2D subdomains in the left. As a result, in this example, when redistributing the data between these two partitions, two independent transmissions are done involving the subdomains $\mathcal{M}_{00}, \mathcal{M}_{01}$ and $\mathcal{M}_{00}, \mathcal{M}_{10}$ on the one hand, and the subdomains $\mathcal{M}_{10}, \mathcal{M}_{11}$ and $\mathcal{M}_{20}, \mathcal{M}_{30}$ on the other. In the general case, P_{2d} independent transmissions need to be done, and P_{per} parallel processes are involved in each of them. These collective communications are performed by means of the MPI_Alltoall routine.

Note that, in Algorithm 2, increasing P_{per} has two counteracting effects on the computing costs: they increase for steps 2 and 4, because more processes are involved in the redistributions of data; whereas it benefits step 3 when P_{2d} is beyond the scalability limit of the 2D solver. Therefore, for each problem and computing platform an optimal P_{per} needs to be found.

4.5 Optimizations for the iterative approach

4.5.1 Storage formats

The data structures used in the PCG algorithm are sparse matrices and vectors. As shown in section 4.3.1, after the Fourier diagonalization, the Laplacian matrix is decomposed into a set N_{per} matrices that differ only in the eigenvalue λ_k multiplying the diagonal contribution Ω_{2d}.

$$\widehat{\mathsf{L}}_k = \lambda_k \Omega_{2d} + \Delta_{per} \mathsf{L}_{2d} \qquad k = 0, ..., N_{per} - 1. \tag{4.33}$$

A naive implementation would be storing independently each one of the N_{per} frequency systems. This approach facilitates the implementation of the algorithm because it relies on standard functions and data structures. However, it does not take advantage of the data redundancies resulting from the Fourier diagonalization. Our strategy consists in storing all the frequency systems on a single data structure avoiding redundancies, and creating the corresponding unified kernel which has a higher FLOP per by ratio. The set of matrices of frequency systems is represented by the Laplacian matrix $\Delta_{per}\mathsf{L}_{2d}$ and the diagonal matrix Ω_{2d}, both of dimension N_{2d}; and the vector λ_k of dimension N_{per} containing the eigenvalues of the Fourier operator.

The vectors involved in the solution of the frequency systems are dense data structures, therefore the only variant regarding its storage is in the ordering of the variables. In our implementation we use the *1D-block-order* described in Section 4.2.1.

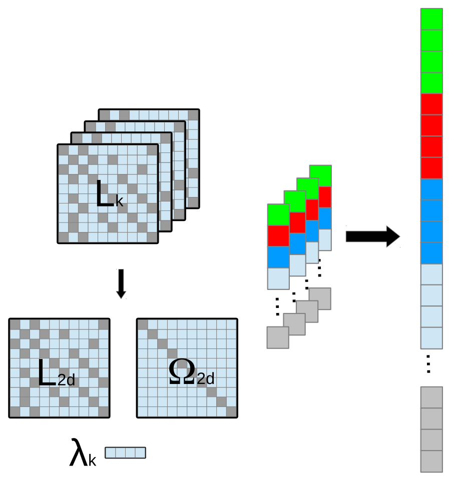


Figure 4.6: Left: Unified storage format. Right: Vectors arrangement

This arrangement propitiates regular accesses to memory through the periodic direction. Figure 4.6 illustrates the unified storage format for the frequency systems and the corresponding vectors.

A compressed storage of any sparse matrix requires at least the individual storage of the coefficients values and its corresponding columns indexes. The number of the non-zero entries is determined by the mesh geometry and the discretization scheme. In our application context, a 2.5D mesh is composed by triangles extruded through the periodic direction forming prismatic cells. We use a second order scheme for the spatial discretization, consequently, the 2D systems resulting from the decomposition contain 4 entries per row. Table 4.1 shows in detail the minimal number of bytes necessary to represent the set of frequency systems with both the naive and our novel approach.

Data Structure	Naive	Unified approach
Values (double)	$N_{per} \times (4N_{2d}) \times 8$	$(4N_{2d}) \times 8$
Columns (Int)	$N_{per} \times (4N_{2d}) \times 4$	$(4N_{2d}) \times 4$
Eigenvalues (double)	–	$N_{per} \times 8$
Total (bytes)	$(48 \times N_{2d} \times N_{per})$	$(8N_{per} + 48N_{2d})$

Table 4.1: Summary of SpMV memory usage for both storage approaches

Generally the N_{2d} is several times larger than N_{per}, for this cases the new approach is almost N_{per} times more efficient in memory usage than explicitly storing each frequency system. Hence, the low memory footprint makes suitable the new approach when running simulations on systems-on-chip platforms, like Bluegene/Q, where the memory space is a scarce resource.

The SpMV kernel has to be adapted to the new storage format. Our implementation consists in computing the product by $\Delta_{per}\mathsf{L}_{2d}$ simultaneously for all the N_{per} frequency systems, while adding the corresponding contribution to the diagonal of each frequency system. This process is explicitly described in Algorithm 3:

Algorithm 3 Unified SpMV

```
1: for i in $N_{2d}$ do
2:     for j in $[\mathsf{L}_{2d}]_i$ do
3:         $[b]_i = 0$
4:         for k in $[0, N_{per})$ do
5:             $b_{ik} += \mathsf{L}_{2d\,ij} x_{jk}$
6:             if $i = j$ then
7:                 $b_{ik} += \lambda_k \Omega_{2d\,i} x_{jk}$
8:             end if
9:         end for
10:    end for
11: end for
```

Where b and x are arrays of dimension $N_{per}N_{2d}$, and the subindex ik refers to the position $iN_{per} + k$ in the arrays. On the other hand, $[\mathsf{L}_{2d}]_i$ refers to the set of column indexes of the non-zero entries in the *ith* row of L_{2d}. Apart of reducing the global memory requirements, a key aspect of this implementation is the reduction of the memory traffic. Each coefficient of the matrix $\mathsf{L}_{2d\,ij}$ is only fetched once from the RAM to the cache for all the frequency systems, this derives in important computing time savings as shown in Section 4.6.

4.5.2 Vectorization

The unified storage format used for the set of frequency systems implies uniform memory accesses through the subsets of N_{per} components aligned in the periodic direction. Consequently, random accesses to the vector x are reduced, minimizing the cache misses, and the prefetching functions to speedup RAM accesses become more effective. Moreover, the kernels executes the same operation through all the variables aligned in the periodic direction, making suitable the use of the SIMD model by means of vectorization.

In the case of Blue Gene/Q systems, the IBM XL C/C++ compiler supports SIMD extensions. The vectorization can be generated by adding some compilation flags (automatically) or by using vector data types and the corresponding SIMD instructions (manually).

To handle automatic vectorization is used the compiler flag `-qsimd=auto`, which indicates the compiler to enable vector registers when possible. In addition, the pragma directives `# pragma disjoint()` need to be embedded in the code specifying the parts where there is no pointer aliasing.

On the other hand, manual vectorization requires of code refactoring, the critical parts of the code are rewritten using the `vector4double` data type (quad) and vector intrinsic operations such as:

- `vec_ld` : loads data from a regular data type into quads
- `vec_st` : stores data from quads to a regular data type
- `vec_mult` : performs the multiplication of two quads
- `vec_madd` : executes the axpy operation with quads

These functions perform 4 instructions in a single CPU-core cycle taking advantage of coalesced memory accesses. Another important aspect that influences the performance is the memory alignment, the data is stored and loaded in groups of 32-byte words (four doubles), therefore the minimum data transaction is produced when the memory address requested corresponds to the beginning of a word. Misaligned data requests introduce additional load instructions and shift or permute operations that degrade the performance of the vectorized code. Our SpMV vectorized kernel is shown in Algorithm 4. The vector variables are written with a v at the beginning of the name. In addition, the definition of a temporal quad vt_i is necessary to calculate the contribution of the eigenvalues. The new kernel requires more calls to intrinsic functions than the traditional implementation. However, the inner loop that sweeps through the periodic directions have been reduced in four times. As shown in the numerical experiments, the vectorized kernel accelerates up to $3\times$ the

SpMV.

Algorithm 4 SpMV Vectorized $b = Lx$

```
for i in $N_{2d}$ do
    $v\Omega_{2di}$ = vec_ld $(0, \Omega_{2di})$
    for j in $[\mathsf{L}_{2d}]_i$ do
        $[vb]_i = 0$
        $vt_i = 0$
        $v\mathsf{L}_{2dij}$ = vec_ld $(0, \mathsf{L}_{2dij})$
        for k in $N_{per}/4$ do
            $vx_{jk}$ =vec_ld$(0, x_{jk})$
            $vb_{ik}$=vec_madd$(vx_{jk}, v\mathsf{L}_{2dij}, vb_{ik})$
            if $i = j$ then
                $v\lambda_k$ = vec_ld $(0, \lambda_k)$
                $vt_i$ =vec_mult$(v\lambda_k, v\Omega_{2di})$
                $vb_{ik}$=vec_madd$(vx_{jk}, vt_i, vb_{ik})$
            end if
        end for
    end for
    vec_st($vb_{ik}$,0,$b_{ik}$)
end for
```

4.5.3 Communication reduction

As mentioned in the previous section, the N_{per} frequency systems that result from the Fourier diagonalization, are divided into P_{per} sub-sets containing $N_{per,k}$ subsystems each, and P_{2d} processors are assigned to the solution of each sub-set. Therefore, for the solution of each frequency system are engaged P_{2d} parallel processes. In our approach each operation of the PCG solver is performed simultaneously for the $N_{per,k}$ frequency systems composing a subset. As a consequence, the the all-to-all and point-to-point communications required by the norm, dot and SpMV operations are grouped and executed synchronously for all the P_{2d} processes, the benefit of this strategy is the reduction of the inter-core communications by a factor of $N_{per,k}$.

4.5.4 Auto-tuning

In section 4.3.2, it is shown that not all the frequency systems have the same solution costs. This means that a uniform partition of the set of frequency systems would derive in a significant imbalance, being the subset with the lowest frequencies the most expensive. Our approach to solve this problem is a dynamic load balance. The cost for the solution of each subset of frequency systems is monitored and the partition is adapted periodically, however a residual imbalance is tolerated in order to avoid an excessive overhead produced by the balance process. In particular, in our application context, once the flow reaches the statistically stationary regime, the solution cost (i.e. iterations) of the frequency systems remains almost constant for the rest of the simulation. Therefore, once a balanced state is achieved, additional balance operations are rarely required for the remaining steps of the simulation. Using an asymmetric partition of the frequency systems requires some changes in the communication pattern, because different processors need to send and receive different amounts of data. In particular, the change of basis from the physical to the spectral space and vice-versa requires substituting the `MPI_Alltoall` communications by `MPI_Alltoallv`. Changing the distribution of the frequency systems requires also to reevaluate the preconditioner for the systems that are redistributed, this produces an overhead but, as mentioned, in our application context the load balance process could be considered as "runtime preprocessing" which is only executed on the initial stages of the time integration process. The second aspect that is automatically tuned is the choice of the preconditioner for each subset of frequency systems. A unique preconditioner is adopted for each subset in order to favor the vectorization. However, different subsets may be more efficiently solved by different preconditioning methods. In general, for the lowest frequencies is more optimal an accurate preconditioner since those are more ill-conditioned systems. While, the highest frequencies are much more diagonal dominant, therefore the Jacobi diagonal scaling performs very well. In the current version of our algorithm we combine two preconditioners, the sparse Approximate Inverse (AIP) [27] and the Jacobi diagonal scaling. A greedy algorithm based in alternate the use of the preconditioners and its parameters is utilized to estimate the best configuration. In the same way than the balancing, the preconditioners tuning takes place during the first steps of the simulation and its costs become negligible in our application context.

4.6 Numerical experiments

The numerical experiments of this study were performed on the Blue Gene/Q Vesta supercomputer of the Argonne Leadership Computing Facility (ALCF), this is a test and development platform used as a pad for researchers to the largest ALCF supercomputer Mira (ranked 5th in the Top500 list). Vesta has two computer racks that sum up a total of 32,768 cores with a peak performance of ≈ 0.5 PFlops.

4.6.1 Vectorization

Figure 4.7 shows the GFLOPS achieved with the vectorized SpMV using our novel storage format versus the GFLOPS achieved with the naive approach that consists in multiplying separately one frequency system after the other. The test case is the Laplacian matrix discretized on meshes generated by the extrusion of a 2D mesh with 577K nodes, N_{per} varies from 4 to 32. The performance of the naive approach is independent of N_{per}, because increasing N_{per} is just repeating more times the same kernel. On the contrary, the performance of the unified SpMV improves with N_{per}, because the larger it is N_{per} the larger is the uniform part of the problem in which the vectorization and memory prefetching produce acceleration. Since the vectorization is performed using vector datatypes composed of four doubles (quads). When N_{per} is multiple of 4 there is a perfect alignment and all the components of the multiplying vector fetched to the cache are effectively used, this results on the peaks of performance observed in the figure. The speedup of the unified SpMV versus the naive approach ranges between 1.4× and 3.5×.

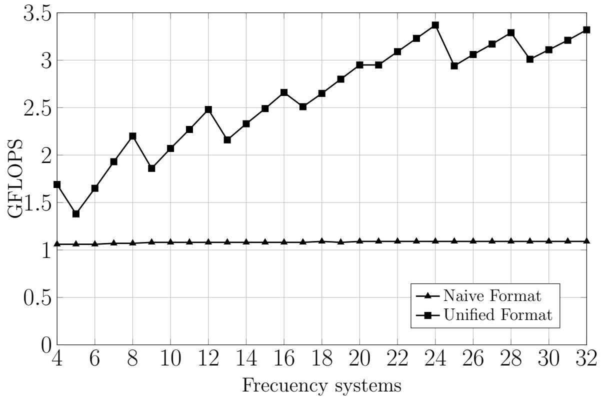


Figure 4.7: Unified SpMV versus naive approach for meshes generated by the extrusion of a 2D grid with 577K nodes

Figure 4.8 shows the speedup of the unified versus the naive approach of both the SpMV and the CG with diagonal scaling (CG-diag). The performance improvement on the SpMV benefits the CG solver, but the improvement is limited by Amdahl's law since the relative weight of the SpMV on the CG-diag is around 60%. Using other methods such as the Approximate Inverse Preconditioner results in higher speedups versus the naive approach.

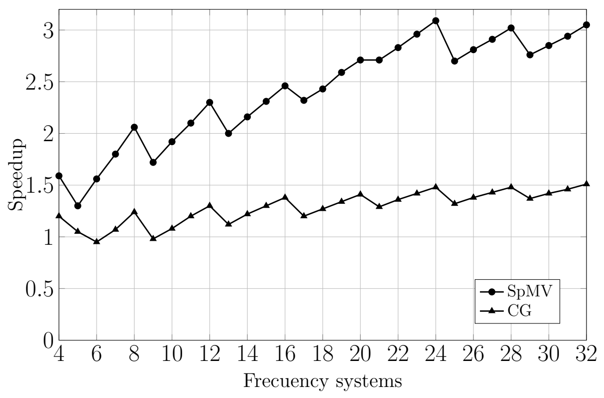


Figure 4.8: The speedup of cg and unified SpMV for meshes generated by the extrusion of a 2D grid with 577K nodes.

4.6.2 Autotuning

Figure 4.9 illustrates the auto-tuning process. Tests have been performed on a mesh of about 90 Million nodes composed of 128 planes of 700K nodes each. This was one of the intermediate meshes used on the LES simulations performed on our study of the Drag crisis of the flow around circular cylinder [13]. In particular, the results shown are for a 512×4 partition. In (a) is shown the imbalance reduction achieved with the auto-balance process used to define the distribution of the frequency systems, in this case the Diagonal Jacobi preconditioner is used for all blocks of frequency systems. The imbalance is measured as the maximum difference on the solution of different subsets of frequency systems, divided by the average time. In (b) is shown the initial and final distribution of planes, as expected, the block containing the larger frequencies are more loaded that the blocks containing the lower ones since the solution costs grow with the frequency number. Finally (c) shows the solution time obtained for the same test case with different preconditioning configurations. The first column corresponds to the case where Jacobi diagonal scaling is used for all the blocks of frequency systems, AIP1 refers to the case where only the first block is solved with the AIP preconditioner and the rest with the Jacobi diagonal scaling,for a AIP2 the two first blocks are solved with the AIP preconditioner and for AIP3 the first three blocks. The auto-tuning process ends when at increasing the number of blocks solved with the AIP preconditioner the solution time does not reduce. Therefore, in this particular case, the final configuration would be AIP2. Changing the preconditioning configuration has some associated costs: the set up for the new preconditioner and re-balance the distribution of frequency systems. But this can be considered as a runtime setup that does not require interruptions on the simulation process. In particular in this case the speedup achieved from the initial to the final configuration is 2.3×.

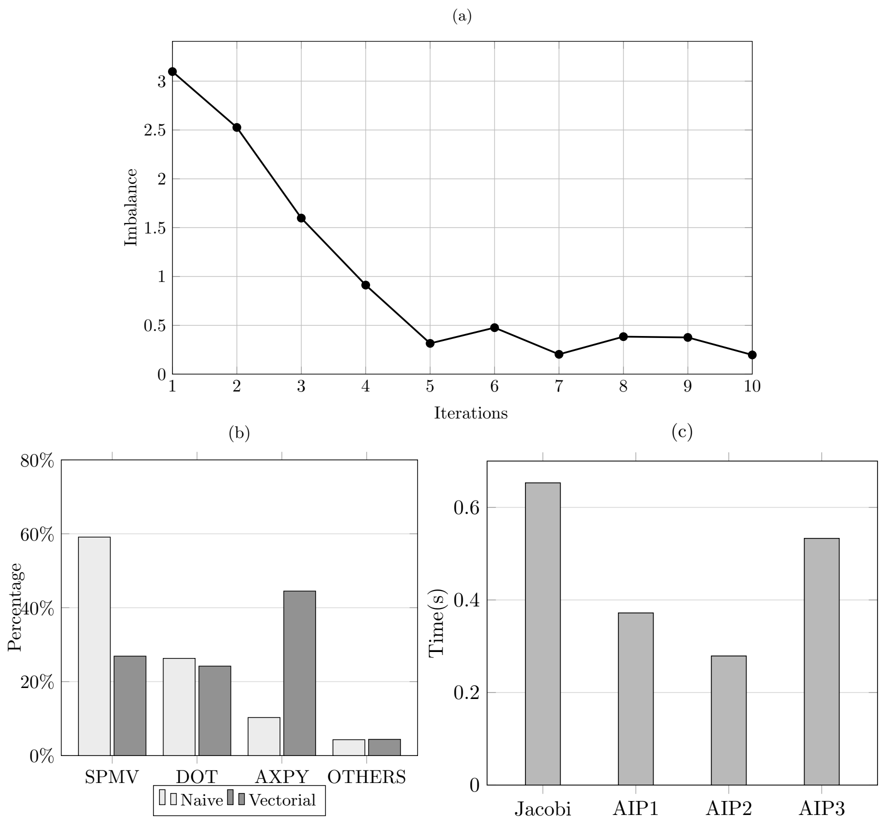


Figure 4.9: Auto-tuning process: (a) Load Imbalance reduction, (b) Initial an final distribution of frequency systems, (c) Preconditioner choice.

4.6.3 Scalability

A strong scalability test has been performed comparing our previous direct approach FFT-DSD, in which the frequency systems where solved by means of a Direct Schur-complement based Decomposition (DSD) [5], and the iterative solution here proposed (FFT-PCG). The same test case of the previous subsection is used here. The direct approach has been used in other supercomputers for LES simulations on meshes with up to 300M nodes [13], however 2GB of RAM per core where used in those simulations. The reduced size of of 1Gb per core on Vesta supercomputer, allowed us to use only 8 of the 16 CPU-cores of the Vesta nodes. For this reason, we could only attest the performance of the FFT-DSD approach up to 8192 CPU-cores. In fact, this limitations are one of the reasons to evolve our solver. Results show that in the region where the FFT-DSD is applied (wasting half of cores allocated) it is faster than our new iterative approach. However the iterative approach scales better, initially (using 2048 CPU-cores) the FFT-DSD is 35% faster but with 8192 CPU-cores this difference reduces at 16%, finally using 16384 CPU-cores the iterative approach overcomes the FFT-DSD by 29%. The overall parallel efficiency of the FFT-PCG approach is 77%. Note that the load per CPU-core in the last case is only of about 5500 cells, which is a very small load for the Vesta CPU-cores, this fact glimpses a great scalability potential of the code.

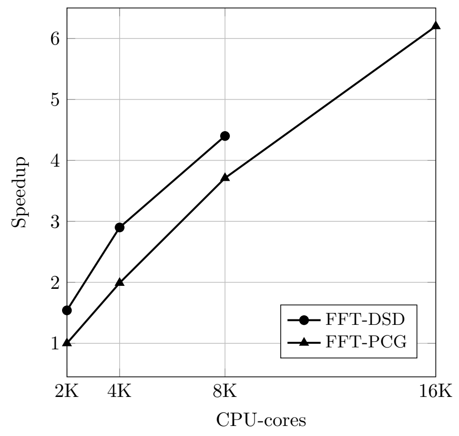


Figure 4.10: Strong speedup test for the direct (FFT-DSD) and iterative (FFT-PCG) approaches.

4.7 Concluding remarks

This chapter presents the efforts to evolve our Poisson solver for simulations with one FFT diagonalizable direction in order to align our strategy with the evolution of supercomputing systems. This evolution brings larger number of parallel processes involved in a single task and less RAM memory per parallel process.

Our previous strategy, was based on a direct solution of all or part of the mutually independent frequency systems that result from the Fourier diagonalization by means of a Direct Schur-complement based Decomposition (DSD). This strategy has showed highly efficient and robust, and has been applied to DNS and LES simulations, engaging up to 5120 CPU-cores and meshes with up to 600M nodes. However, both the reduced RAM memory per parallel process and the increase of the memory requirements to solve the interface system in the DSD method, become an overwhelming wall for its further scalability.

We have evolved into a purely iterative strategy, by solving all the frequency systems by means of a Preconditioned Conjugate Gradient method. In this new implementation we have focused on optimizing the memory allocations and transactions and on taking advantage of the regularity of the memory accesses and operations through the periodic direction for its vectorization. The speedup achieved with the vectorization of the SpMV kernel, for which a specific format has been developed, averages 1.6$\times$, with peaks of about 3$\times$ when perfect alignment is achieved and enough frequency systems are operated simultaneously. Since the SpMV is the dominant kernel of the simulation code, a potential acceleration of all the code turns up, specially on the explicit parts of it. In the Poisson solver, we have observed that the acceleration of the Jacobi preconditioned CG averages 1.2$\times$ with peaks of 1.5$\times$, with respect of solving each frequency system separately. This result is consistent with the relative weight of the SpMV within the linear solver.

The second focus of our algorithm design has been the auto-tuning capabilities. The iterative solution of the frequency systems has variable cost according to the conditioning of each system. In general, the lower frequencies couple larger parts of the domain and require more iterations. On the other hand, the optimal preconditioning requirements of the frequency systems differ, the higher frequencies are strongly diagonal dominant and Jacobi diagonal scaling performs very well but the lower require a more accurate approximation. In order to deal with these variable situation, that depends on the physical problem being considered and the computing system engaged, we have developed a run-time auto-tuning that adjusts both aspects on the time integration process of the simulation without requiring user intervention neither the simulation interruption. Finally the strong scalability of the new algorithm has been successfully attested up to 16384 CPU-cores.

The reduced memory requirements of our new approach, its demonstrated scalability and auto-tuning capabilities, and its good performance compared with the direct approach previously used in several HPC systems, make it a highly efficient and portable code adapted to the characteristics of ongoing HPC systems.

References

[1] R. W. Hockney. A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *Journal of the Association for Computing Machinery*, 12:95–113, 1965.

[2] M. Soria, C. D. Pérez-Segarra, and A.Oliva. A Direct Schur-Fourier Decomposition for the Solution of the Three-Dimensional Poisson Equation of Incompressible Flow Problems Using Loosely Parallel Computers. *Numerical Heat Transfer, Part B*, 43:467–488, 2003.

[3] F. X. Trias, M. Soria, C. D. Pérez-Segarra, and A. Oliva. A Direct Schur-Fourier Decomposition for the Efficient Solution of High-Order Poisson Equations on Loosely Coupled Parallel Computers. *Numerical Linear Algebra with Applications*, 13:303–326, 2006.

[4] A. Gorobets, F. X. Trias, M. Soria, and A. Oliva. A scalable parallel Poisson solver for three-dimensional problems with one periodic direction. *Computers & Fluids*, 39:525–538, 2010.

[5] R. Borrell, O. Lehmkuhl, F. X. Trias, and A. Oliva. Parallel Direct Poisson solver for discretizations with one Fourier diagonalizable direction. *Journal of Computational Physics*, 230:4723–4741, 2011.

[6] M. Soria, F. X. Trias, C. D. Pérez-Segarra, and A. Oliva. Direct numerical simulation of a three-dimensional natural-convection flow in a differentially heated cavity of aspect ratio 4. *Numerical Heat Transfer, part A*, 45:649–673, April 2004.

[7] F. X. Trias, M. Soria, A. Oliva, and C. D. Pérez-Segarra. Direct numerical simulations of two- and three-dimensional turbulent natural convection flows in a differentially heated cavity of aspect ratio 4. *Journal of Fluid Mechanics*, 586:259–293, 2007.

[8] F. X. Trias, A. Gorobets, M. Soria, and A. Oliva. Direct numerical simulation of a differentially heated cavity of aspect ratio 4 with Ra-number up to 10^{11} - Part I: Numerical methods and time-averaged flow. *International Journal of Heat and Mass Transfer*, 53:665–673, 2010.

[9] F. X. Trias, A. Gorobets, M. Soria, and A. Oliva. Direct numerical simulation of a differentially heated cavity of aspect ratio 4 with Ra-number up to 10^{11} - Part II: Heat transfer and flow dynamics. *International Journal of Heat and Mass Transfer*, 53:674–683, 2010.

[10] J. E. Jaramillo, F. X. Trias, A. Gorobets, C. D. Pérez-Segarra, and A. Oliva. DNS and RANS modelling of a Turbulent Plane Impinging Jet. *International Journal of Heat and Mass Transfer*, 55:789–801, 2012.

[11] I. Rodríguez, R. Borrell, O Lehmkuhl, C. D. Pérez-Segarra, and A. Oliva. Direct numerical simulation of the flow over a sphere at $Re = 3700$. *Journal of Fluid Mechanics*, 679:263–287, 2011.

[12] I. Rodríguez, O. Lehmkuhl, R. Borrell, and A. Oliva. Direct numerical simulation of a NACA0012 in full stall. *International Journal of Heat and Fluid Flow*, 43:194–203, 2013.

[13] O. Lehmkuhl, I. Rodríguez, R. Borrell, J. Chiva, and A. Oliva. Unsteady forces on a circular cylinder at critical Reynolds numbers. *Physics of Fluids*, page 125110, 2014.

[14] R. W. C. P. Verstappen and A. E. P. Veldman. Symmetry-Preserving Discretization of Turbulent Flow. *Journal of Computational Physics*, 187:343–368, 2003.

[15] Y. Morinishi, T.S. Lund, O.V. Vasilyev, and P. Moin. Fully Conservative Higher Order Finite Difference Schemes for Incompressible Flow. *Journal of Computational Physics*, 143:90–124, 1998.

[16] A. J. Chorin. Numerical Solution of the Navier-Stokes Equations. *Journal of Computational Physics*, 22:745–762, 1968.

[17] N. N. Yanenko. *The Method of Fractional Steps*. Springer-Verlag, 1971.

[18] F. X. Trias, O. Lehmkuhl, A. Oliva, C.D. Pérez-Segarra, and R.W.C.P. Verstappen. Symmetry-preserving discretization of Navier-Stokes equations on collocated unstructured meshes. *Journal of Computational Physics*, 258:246–267, 2014.

[19] R. M. Gray. Toeplitz and Circulant Matrices: A review. *Foundations and Trends in Communications and Information Theory*, 2:155–239, 2006.

[20] P.J.Davis. *Circulant Matrices*. Chelsea Publishing, New York, 1994.

[21] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[22] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, second edition, 2003.

[23] S. Kocak and H. U. Akay. Parallel Schur complement method for large-scale systems on distributed memory computers. *Applied Mathematical Modelling*, 25:873–886, 2001.

[24] Natalja Rakowsky. The Schur Complement Method as a Fast Parallel Solver for Elliptic Partial Differential Equations in Oceanography. *Numerical Linear Algebra with Applications*, 6:497–510, 1999.

[25] Y. Saad and M. Sosonkina. Distributed Schur complement techniques for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21:1337–1356, 2000.

[26] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1999.

[27] Y. Saad E. Chow. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal of Scientific Computing*, 19:995–1023, 1998.

5 Conclusions and Future research

5.1 Conclusions

The objective of this thesis was the development of strategies to facilitate the adaptation of the computing approach of our unstructured CFD code to the new high performance computing trends, and in particular to the incoming hybrid architectures. In this context, our main contribution was the development of a highly portable and modular CFD implementation model and its optimized implementation for various leading edge systems . Our solution consisted in substituting stencil data structures and kernels by algebraic storage formats and operators. The result is that around 98% of our CFD computations are based in three basic algebraic kernels: SpMV, AXPY and DOT, providing a high level of modularity and a natural portability to the code. By doing so, the most intensive computing part of the algorithm and the complex architecture tuning is isolated on a low level portable API. Moreover, in its current state, new models and physics can be easy implemented through this API. The portability of our model has been tested with positive results in Minotauro and Mont-Blanc supercomputers, for which specific optimizations have been developed on the basic algebraic kernels. Additionally, some concepts learned from the refactoring processes have been applied on the development of a Poisson solver capable of exploiting peta-scale computing systems.

A brief list of the main contribution of this thesis is listed by chapters:

- **MPI-CUDA Sparse Matrix-Vector Multiplication for the Conjugate Gradient Method with an Approximate Inverse Preconditioner**:

  The context of application was the numerical resolution of the Navier Stokes equations in the simulation of incompressible flows. In particular, we focused our attention into porting the Poisson equation to GPUs, because it dominates the computing costs. Two major issues were identified when working with hybrid CPU/GPU clusters: i) the node configuration is usually asymmetric, this means that exists more CPU-cores than GPUs; ii) there is no direct link between discrete devices, therefore communications are negatively affected by PCI-express speed. Our solution for the asymmetric configuration is based in a two level domain decomposition. A master-worker execution model has been proposed in order manage the data flow between the CPU-only and CPU-GPU parallelization modes. This scheme consist of two functions for scattering and gathering data between the two level topologies. The relative weight of this operations is minor in comparison with the computing costs of the PCG solver. On the other hand, the main characteristic of the hybrid SpMV developed is its overlapping strategy that allows to hide part of the data transfer overhead, produced by the device-to-host and MPI communications, behind calculations on GPUs. Performance tests, engaging different numbers of hybrid nodes and unstructured meshes of different sizes, have demonstrated speedups of the hybrid PCG solver of around $3.7\times$ compared to the CPU-only solver. The corresponding speedup of the overall CFD algorithm replacing the linear solver by the new version would be up to $2.4\times$. Moreover, it has been demonstrated that the performance of general purpose sparse libraries such as cuSPARSE can be improved by application specific in-house SpMV kernels implemented considering specific information of the matrix. This performance is directly linked with the sparse matrix storage format being used. The positive results obtained encouraged us to explore the portability of the whole time integration phase into GPUs.

- **Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers**:

  The contribution of this chapter is twofold. Firstly, we propose a portable modeling for LES of incompressible turbulent flows based on an algebraic operational approach. The main idea is substituting stencil data structures and kernels by algebraic storage formats and operators. The result it that around 98% of computations are based on the composition of three basic algebraic kernels: SpMV, AXPY and DOT, providing a high level of modularity and a natural portability to the code. Note that optimal implementations of such three kernels can be found in many software libraries. Among the three algebraic kernels the SpMV is the dominant one, it substitutes the stencil iterations and the non-linear terms, such as the convective operator, are rewritten as two consecutive SpMVs. Aiming a high level of modularity and portability for our code is motivated by the current disruptive technological moment, with a high level of heterogeneity and where many computing models are under consideration without any clarity of which one is going to prevail.

The second objective has been the implementation of our model to run on heterogeneous clusters engaging both CPU and GPU coprocessors. This objective is motivated by the increasing presence of accelerators on HPC systems, driven by power efficiency requirements. We have analyzed in detail the implementation of the SpMV kernel on GPUs, taking into account the general characteristics of CFD operators. Our in-house implementation based on an sliced ELLPACK format outperforms other general purpose libraries such as MKL on CPUs and cuSPARSE on GPUs. Moreover, for multi-GPU executions we have developed a communication-computations overlapping strategy. On the other hand, for the AXPY and DOT kernels no specific optimizations have been developed because they are application-independent kernels with optimal implementations in libraries such as cuSPARSE.

Finally, several numerical experiments have been performed on the MinoTauro supercomputer of the Barcelona Supercomputing Center in order to understand the performance of our code on multi-GPU platforms, and compare it with multi-core executions. First we have profiled the code for both implementations showing that certainly 98% of time is spent on the three main algebraic kernels. Then, we have focused on the SpMV kernel, showing its memory bounded nature, and how the throughput oriented approach of GPUs better harnesses the bandwidth than the low latency oriented approach of CPUs. The result is that although the bandwidth ratio between both devices is 4.4 the speedup of the GPU vs the CPU implementation reaches up to 8$\times$ in our tests. Then the benefits of the overlapping strategy for multi-GPU executions has been tested, showing that large part of the communication (86%) can be hidden. We have also included strong and week speedup tests up to 128 GPUs, in general good PE is achieved if the workload per GPU is kept reasonable. Considering the overall time-step the multi-GPU implementation outperforms the multi-CPU one by a factor ranging between 4$\times$ and 8$\times$ depending on the local problem size. Finally, we have shown that the performance of our code can be very well estimated by only analyzing the three main kernels separately.

- **Hybrid auto-balanced implementation of parallel unstructured CFD kernels in Mont-Blanc ARM-based platforms**:

  High performance computing is exploring new energy efficient technologies aiming to reach the exascale paradigm. In this chapter we present a study of the performance of TermoFluids code on the Mont-Blanc mobile-based prototypes. Our solution is based in the specific implementation of the portable model proposed in Chapter 2. Note that scientific computing in such embedded architectures is still in its early stages, therefore there are not algebraic libraries that facilitate the implementation yet. Consequently, our focus was the implementation of the three main algebraic kernels that compose our computations: SpMV, AXPY and DOT. Such kernels are memory bounded operations, then its performance mostly depends of the memory bandwidth of the computing device where they run.

In Mont-Blanc prototypes, the CPU and GPU are equipped with the same memory bandwidth, so similar performance was expected for the kernels in sequential execution. However, the SpMV shows a 1.5× speedup in average of the GPU versus the CPU implementation. This confirms that the stream processing model of GPUs better harnesses the bandwidth for the SpMV as explained in detail in Chapter 2.

Our main contribution in the implementation of the kernels consists in a hybrid implementation of them for simultaneity engaging both devices of the Mont-Blanc prototypes. This approach is possible because of two reasons: the physically shared memory of the mobile architecture that avoids PCI-e communication episodes between GPU and CPU, and the similar net performance of both computing devices that makes it worth the sharing. The problem consists in finding the best workload distribution to maximize the net performance. Our solution has been the development of an auto-balancing algorithm based in a tabu search that tunes the kernels workload in a pre-processing stage. Following this approach the SpMV, AXPY and DOT have been accelerated in average by 1.64× , 1.60× and 1.74×, respectively.

In the parallel execution, the load distribution includes the inter-node communication that is performed by the CPU. Therefore, the communications are overlapped with computations on the GPU that increases its workload to balance the cost of the communication performed by the CPU. Several parallel experiments have been performed with meshes of different sizes and engaging up to 64 nodes. The best parallel efficiency achieved on 64 nodes has been 40% for a mesh of 6.4M nodes. However the communication technology of the Mont-Blanc nodes is still based on the Gigabit network and we estimate that the performance would be much higher with an state of the art Infiniband network.

Summarizing, we found two main limitations for running CFD in the mobile architecture. Firstly, the slow network in comparison with the current supercomputers because Infiniband is not supported yet. Secondly, error correction code is not incorporated in the mobile chips, this makes unfeasible the execution of large scale simulations because the correctness of the results is not guaranteed. However, many of the aforementioned limitations were encountered in the early days of the GPU computing and were improved during the evolution of such technology. Finally, by means of a qualitative approximation we conclude that at running our CFD code the Mont-Blanc nodes are in average 2.4× more energy efficient than a current hybrid high-end supercomputer such as MinoTauro of the BSC. It is expected that mobile technology continues its evolution due to the strong market trends. Therefore, we consider that this technology has the potential of becoming an important part of the future supercomputing technology.

- **Poisson solver for Peta-scale simulations with one FFT diagonalizable direction**:

  This chapter presents the efforts to evolve our Poisson solver for simulations with one FFT diagonalizable direction in order to align our strategy with the evolution of supercomputing systems. This evolution brings larger number of parallel processes involved in a single task and less RAM memory per parallel process.

  Our previous strategy, was based on a direct solution of all or part of the mutually independent frequency systems that result from the Fourier diagonalization by means of a Direct Schur-complement based Decomposition (DSD). This strategy has showed highly efficient and robust, and has been applied to DNS and LES simulations, engaging up to 5120 CPU-cores and meshes with up to 600M nodes. However, both the reduced RAM memory per parallel process and the increase of the memory requirements to solve the interface system in the DSD method, become an overwhelming wall for its further scalability.

  We have evolved into a purely iterative strategy, by solving all the frequency systems by means of a Preconditioned Conjugate Gradient method. In this new implementation we have focused on optimizing the memory allocations and transactions and on taking advantage of the regularity of the memory accesses and operations through the periodic direction for its vectorization. The speedup achieved with the vectorization of the SpMV kernel, for which a specific format has been developed, averages 1.6×, with peaks of about 3× when perfect alignment is achieved and enough frequency systems are operated simultaneously. Since the SpMV is the dominant kernel of the simulation code, a potential acceleration of all the code turns up, specially on the explicit parts of it. In the Poisson solver, we have observed that the acceleration of the Jacobi preconditioned CG averages 1.2× with peaks of 1.5×, with respect of solving each frequency system separately. This result is consistent with the relative weight of the SpMV within the linear solver.

  The second focus of our algorithm design has been the auto-tuning capabilities. The iterative solution of the frequency systems has variable cost according to the conditioning of each system. In general, the lower frequencies couple larger parts of the domain and require more iterations. On the other hand, the optimal preconditioning requirements of the frequency systems differ, the higher frequencies are strongly diagonal dominant and Jacobi diagonal scaling performs very well but the lower require a more accurate approximation. In order to deal with these variable situation, that depends on the physical problem being considered and the computing system engaged, we have developed a run-time auto-tuning that adjusts both aspects on the time integration process of the simulation without requiring user intervention neither the simulation interruption. Finally the strong scalability of the new algorithm has been successfully attested up to 16384 CPU-cores.

The reduced memory requirements of our new approach, its demonstrated scalability and auto-tuning capabilities, and its good performance compared with the direct approach previously used in several HPC systems, make it a highly efficient and portable code adapted to the characteristics of ongoing HPC systems.

5.2 Future Research

The portable implementation model proposed in this thesis has been developed for DNS and LES simulation of incompressible turbulent flows. Following the same principles our intention is to expand the model to more physical phenomena and algebraic solvers. In this context, the main subjects to be explored in the near future are:

- **Poisson solver with one FFT diagonalizable direction in GPUs:** Chapter 4 shows a novel data arrangement for the vectorization of the periodic direction. The same approach is suitable for GPU accelerators, but instead of vectorization, each thread should be responsible of perform the periodic computations. As a result, the arithmetic intensity of the algorithm increases since each coefficient of the matrix is reused as the number of periodic planes.

- **Preconditioners:** In Chapters 1 and 2, the Jacobi and Approximate inverse preconditioners are used for accelerating the solution of the Poisson equation. However, other preconditioners suitable with the algebraic implementation model should be study as well [1]. For instance, the linelet preconditioner [2] has shown good results when dealing with body-fitted meshes, and seems suitable for its hybrid implementation as well.

- **Hybrid precision:** The new architectures have been conceived with an imbalance between double and single precision components. The tendency is to have at least twice more single precision ALUs in GPU devices. Therefore, CFD community must consider the possibility of using mix precision algorithms in order to benefit of all the computing resources available. This means that the most computational intensive parts are calculated in single precision and its solution is corrected in double precision. Since CFD problems are memory bounded an acceleration of at least $2\times$ is estimated due the fact of moving half of the bytes and perform the calculations in half of time. Early attempts at providing mixed precision solvers based in iterative refinement can be found in [3,4].

- **Multiphysics in GPUs:** New algorithms need to be studied for the incorporation of multi-physics (radiation, combustion, etc) into our portable implementation model. Some algorithms that were discarded for CPU execution because of its heavy computational costs, may be suitable for GPU execution. Lattice Boltzmann lighting models [5] seem to be perfect for GPUs since they provide highly parallel sections that can be exploited by the massively parallel processors. In combustion multi-dimensional simulations, the current research lines are focused in the development of hybrid strategies that exploits the explicit and implicit parts of the simulation, combining the strengths of the different CPU and GPU architectures [6].

- **Next generation of accelerators:** Our portable implementation model for CFD proposed in Chapter 2 has structured our code in a way that is very easy to test its performance in any platform. Therefore, our objective is to follow very closely the supercomputing trends and in particular the novel pre-exascale supercomputers based in the next generation of accelerators.

References

[1] Ruipeng Li and Yousef Saad. Gpu-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[2] Orlando Soto, Rainald Lohner, and Fernando Camelli. A linelet preconditioner for incompressible flow solvers. *International Journal of Numerical Methods for Heat & Fluid Flow*, 13(1):133–147, 2003.

[3] Hartwig Anzt, Vincent Heuveline, and Bjorn Rocker. Mixed precision iterative refinement methods for linear systems: Convergence analysis based on krylov subspace methods. In Kristjan Jonasson, editor, *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 237–247. Springer Berlin Heidelberg, 2012.

[4] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl.*, 21(4):457–466, 2007.

[5] James Westall Robert Geist. Lattice boltzmann lighting models. In *GPU Computing Gems*, chapter 25, pages 381–399. Elsevier, 2011.

[6] Yu Shi, William H. Green, Hsi-Wu Wong, and Oluwayemisi O. Oluwole. Accelerating multi-dimensional combustion simulations using gpu and hybrid explicit/implicit ode integration. *Combustion and Flame*, 159(7):2388 – 2397, 2012.

Appendix A

Computing resources

This appendix lists different parallel computing systems where the codes developed in this thesis have been executed. The access to most of these equipments is obtained gaining competitive calls, in which the evaluated criteria are both the scientific relevance of the presented projects and the capability of the software to obtain a good parallel performance with the offered resources.

A.1 JFF supercomputer, Terrassa

Figure A.1: JFF supercomputer

JFF supercomputer consists in 168 computer nodes with 2304 cores and 4.6 TB of

RAM in total. There are 128 nodes with 2 AMD Opteron 2376 quad-core processors at 2.3GHz and 16 GB of RAM linked with the infiniband DDR 4X network, and 40 nodes with 2 AMD Opteron 6272 16-core processors at 2.1 GHz and 64 GB RAM linked with the infiniband QDR 4X network.

A.2 Minotauro supercomputer, Barcelona

Figure A.2: Minotauro supercomputer

Minotauro supercomputer, from the Barcelona Supercomputing Center (BSC), is a hybrid cluster composed by 128 Bull B505 blades each blade with the following configuration. Each node contains 2 Intel E5649 (6-Core) processors at 2.53 GHz, 24 GB of RAM and 2 M2090 NVIDIA GPU cards. Nodes are linked by means of 14 links of 10 GbitEth to connect to BSC GPFS Storage. The peak performance is estimated in 185.78 TFlops.

A.3 Vesta supercomputer,Chicago

Figure A.3: Vesta supercomputer

Vesta is an IBM Blue Gene/Q supercomputer at the Argonne Leadership Computing Facility and it is part of the MIRA supercomputer project. It is utilized for test and development platform, serving as a launching pad for researchers planning to use larger supercomputers of Argonne facilities. The supercomputer is based in System-on-Chip architecture, each node is composed by 16 1600 MHz PowerPC A2 cores and 16 GB RAM. The interconnection is based in a 5D Torus Proprietary Network. In total it is possible to engagge up to 32768 cores and a theoretical peak of 357.776 TFlops.

A.4 Lomonosov supercomputer, Moscow

Lomonosov supercomputer, from the Research Computing Center in the Moscow State University, is a supercomputer with $35,360$ cores and 60 TB of RAM installed in 5100 nodes. There are 4420 nodes with 2 quad-core Intel EM64T Xeon 5570 at 2930 MHz and and 12 GB RAM, and 680 nodes with 2 6-core Intel EM64T Xeon 5670 at 2930 MHz and 12 GB of RAM. The interconnection network is an Infiniband QDR.

Figure A.4: Lomonosov supercomputer

A.5 K100 supercomputer, Moscow

K100 supercomputer, from the Keldysh Institute of Applied Mathematics of the Russian Academy of Science (KIAM RAS), is composed of 64 nodes with 2 6-core Intel Xeon X5670 at 2930 MHz, 3 NVIDIA Fermi 2050 and 96 GB of RAM each one. Implying in total 768 CPU cores, 192 GPUs and 6 TB of RAM. The interconnection of nodes is carried out by means of an Infiniband QDR networks and a PCI-Express.

Figure A.5: K100 supercomputer

A.6 Curie supercomputer, Paris

Figure A.6: Curie supercomputer

The Curie supercomputer, owned by the Grand Equipement National de Calcul Intensif (GENCI) and operated into the Très Grand Centre de Calcul (TGCC) by the Commissariat à l'énergie atomique (CEA), is the first French Tier0 system open to scientists through the French participation into the Partnership for Advanced Computing in Europe (PRACE) research infrastructure.

Curie is offering 3 different partitions of x86-64 computing resources, in the context of this thesis the "Curie hybrid nodes" have been used. The hybrid nodes cluster is composed of 16 bullx B chassis with 9 hybrid GPUs B505 blades, in each blade there are 2 quad-core Intel Westmere 2.66 GHz and 2 Nvidia M2090 T20A, in total 1152 CPU cores and 288 GPUs. In both cases and InfiniBand QDR Full Fat Tree network is used for the interconnection of nodes.

A.7 Mont-Blanc prototypes, Barcelona

Mont-Blanc project is an ARM-based system that aims to set the basis for achieving exascale computing. Each node is composed by SoC Samsung Exynos 5 Dual, the chip consist of 1 dual core Cortex-A15 CPU and 1 ARM Mali T-604 GPU. It is the first cluster based in mobile technology and therefore its configuration is in constant development.

Figure A.7: MontBlanc supercomputer

Appendix B

Main publications in the context of this thesis

International Journal Papers:

- G. Oyarzun, R. Borrell, A. Gorobets, O. Lehmkuhl, and A. Oliva. Direct Numerical Simulation of Incompressible Flows on Unstructured Meshes Using Hybrid CPU/GPU Supercomputers. *Procedia Engineering*, Volume 61, 2013, Pages 87–93.

- G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva. MPI-CUDA sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. *Computer and Fluids*, Volume 92, 2014, Pages 244–252.

- G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva. Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers. *SIAM journal of scientific computing*, submitted 2015.

International Conference Papers:

- G. Oyarzun, R. Borrell, O. Lehmkuhl, and A. Oliva. Hybrid MPI-CUDA approximate inverse preconditioner. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

- G. Oyarzun, R. Borrell, A. Gorobets, O. Lehmkuhl, and A. Oliva. Direct Numerical Simulation of Incompressible Flows on Unstructured Meshes Using Hybrid CPU/GPU Supercomputers. In *Parallel Computational Fluid Dynamics*, Changsha (China), May 2013.

- G. Oyarzun, R. Borrell, A. Gorobets, O. Lehmkuhl, and A. Oliva. Hybrid MPI-CUDA Implementation of Unstructured Navier-stokes Solver focusing on CG

Preconditioners. In *Parallel Computational Fluid Dynamics*, Trondheim (Norway), May 2014.

- G. Oyarzun, R. Borrell, A. Gorobets, O. Lehmkuhl, and A. Oliva. LES Unstructured Finite Volume CFD Simulations Engaging Multiple Accelerators. In *Parallel Computational Fluid Dynamics*, Montreal (Canada), May 2015.