UNIVERSITÉ DE STRASBOURG

École Doctorale de Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire des Sciences de l'Ingénieur, de l'Informatique et de l'Imagerie - UMR7357

THÈSE présentée par

Clément Flint

soutenue le : **2 octobre 2024**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**

Discipline / Spécialité : **Informatique**

Compression de données efficace pour les solveurs d'EDP haute performance

Vers des simulations de dynamique des fluides efficientes dans des environnements à mémoire limitée

THÈSE dirigée par :

M. Stéphane Genaud	Professeur, Université de Strasbourg
M. Philippe Helluy	Professeur, Université de Strasbourg

RAPPORTEURS :

M. Hartwig Anzt	Professeur, Technische Universität München
M^{me} Laura Grigori	Professeur, École Polytechnique Fédérale de Lausanne

AUTRES MEMBRES DU JURY :

M. Eric Goncalvès	Professeur, École Nationale Supérieure de Mécanique et d'Aérotechnique
M. Yannick Hoarau	Professeur, Université de Strasbourg

INVITÉS :

M. Bérenger Bramas	Chargé de Recherche, Inria

Contents

List of Figures

List of Tables

Listings

Acknowledgements

First and foremost, I dedicate this thesis to my family, whose unwavering support and encouragement have been my constant foundation throughout this journey. My deepest gratitude goes to my wife, Alicia, whose love and patience sustained me through the countless hours of research and writing. During this time, she also gave birth to our son, Leonard, displaying an incredible strength and resilience that continue to inspire me daily.

To my parents, brothers, and sisters, I extend heartfelt thanks for always being there for me, offering their support and guidance without hesitation. Their sacrifices have played an essential role in helping me reach this milestone, and I am deeply proud to be part of such a wonderful family.

I am also grateful to my friends, who have been steadfast sources of encouragement and positivity. The shared experiences, both online and in the so-called "real world," have been priceless, providing moments of laughter and respite. From our unusual adventures in urban exploration to the excitement of our crypto endeavors, these memories are ones I will cherish. I look forward to many more shared adventures in the future.

To my thesis advisors, I owe profound thanks. Philippe Helluy, whose intuition formed the foundation of this thesis, has been an exemplary figure, not only in his scientific insights but also in his kindness and humanity. His ability to convey complex mathematical concepts with clarity has significantly shaped my understanding and communication of ideas across different scientific communities. Stéphane Genaud, my co-advisor, has been an exceptional mentor. His experience in management and the thoughtful feedback he provided throughout this journey have been invaluable. I greatly appreciate the time we have shared, and I am thankful for his dedication to my work.

A special note of gratitude goes to Bérenger Bramas, whose mentorship has been instrumental from the beginning. His guidance convinced me to take the plunge into academia, and his expertise in high-performance computing has been a crucial asset throughout the development of this thesis. I am truly grateful for his ongoing support and encouragement.

Finally, I would like to thank my colleagues, whose camaraderie and collaboration have made my time in the laboratory an enriching and rewarding experience. Our discussions have broadened my perspectives, and I feel fortunate to have worked alongside such talented and dedicated individuals.

Abstract

This thesis explores the challenges of conducting large-scale Computational Fluid Dynamics (CFD) simulations on modern High-Performance Computing (HPC) platforms, with a particular focus on Graphics Processing Units (GPUs). We present novel strategies and optimizations that aim to push the boundaries of current CFD simulation capabilities, particularly in the context of memory management.

We provide a foundational overview of fluid simulation methods and discuss the state-of-the-art methods for managing high memory requirements. Building upon this groundwork, we propose a relatively simple yet effective single-GPU implementation that achieves satisfactory performance. We then extend this implementation to a multi-GPU framework thanks to task-based runtime systems, such as PaRSEC and StarPU.

In PaRSEC, we develop a new feature in the PTG DSL to allow for more flexibility in the definition of tasks and show that it can be used to express a stencil computation elegantly with no visible cost on performance. In StarPU, we develop a generic stencil solver that can run on a distributed environment and show that we can achieve high scalability. Finally, we improve the Heteroprio scheduler of StarPU by introducing AutoHeteroprio, a fully automatic scheduler that can adjust the priorities of the tasks at runtime, contrary to Heteroprio, which requires manual priority assignment.

In a second part, we focus on using explicit data compression to achieve better memory efficiency. We begin by designing wavelet schemes, tailored for CFD simulations, and show that they can achieve high compression ratios with minimal loss in simulation accuracy. We then tune the wavelet-based compression for high compression throughput on GPUs and show that effective memory gains can be achieved without compromising simulation accuracy nor performance.

Résumé

Dans cette thèse, nous explorons les défis liés à la réalisation de simulations de dynamique des fluides (CFD) à grande échelle sur des plates-formes informatiques modernes de haute performance, en mettant particulièrement l'accent sur les unités de traitement graphique (GPU). Nous présentons des stratégies et des optimisations novatrices visant à repousser les limites des capacités actuelles de simulation CFD, en particulier dans le contexte de la gestion de la mémoire.

Nous fournissons un aperçu fondamental des méthodes de simulation des fluides et discutons des méthodes de pointe pour gérer les fortes exigences en mémoire. En nous appuyant sur ces bases, nous proposons une implémentation relativement simple mais efficace sur un seul GPU qui atteint des performances satisfaisantes. Nous étendons ensuite cette implémentation à un cadre multi-GPU grâce à des systèmes de runtime basés sur des tâches, tels que PaRSEC et StarPU.

Dans PaRSEC, nous développons une nouvelle fonctionnalité sur le DSL PTG pour permettre plus de flexibilité dans la définition des tâches et montrons qu'il peut être utilisé pour exprimer un calcul de stencil de manière élégante sans coût visible sur les performances. Dans StarPU, nous développons un

solveur de stencil générique qui peut fonctionner sur un environnement distribué et montrons que nous pouvons atteindre une grande scalabilité. Enfin, nous améliorons l'ordonnanceur Heteroprio de StarPU en introduisant AutoHeteroprio, un planificateur entièrement automatique qui peut ajuster les priorités des tâches à l'exécution, contrairement à Heteroprio, qui nécessite un attribution manuelle des priorités.

Dans un deuxième volet, nous nous concentrons sur l'utilisation de la compression de données explicite pour obtenir une meilleure efficacité mémoire. Nous commençons par concevoir des schémas d'ondelettes, adaptés aux simulations CFD, et montrons qu'ils peuvent atteindre des taux de compression élevés avec une perte minimale de précision de simulation. Nous réglons ensuite l'algorithme de compression pour obtenir un meilleur un débit de compression sur les GPU et montrons que des gains mémoire effectifs peuvent être obtenus sans compromettre la précision ni les performances de la simulation.

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) simulations are a type of numerical simulation used to study the behavior of fluids. It is a topic of high interest in various fields, such as aeronautics, meteorology, and environmental sciences. As the hardware capabilities of modern computers have increased, the CFD community has been able to simulate more complex and realistic scenarios. However, the hardware landscape has also become more complex, requiring increased knowledge and expertise to efficiently use the available resources.

Graphics Processing Units (GPUs) have become a popular choice for accelerating CFD simulations, as they offer high computational power and energy efficiency. However, one downside of using GPUs is that they have a limited amount of memory. Once the memory requirements of a simulation exceed the available memory of a GPU, new strategies must be developed to ensure a correct execution of the simulation. One common strategy is to split the simulation across multiple GPUs, which introduces new challenges related to data movement and synchronization. Another strategy is to split the data into smaller chunks and use a back-and-forth approach to process one chunk at a time. However, this second strategy is generally slow and inefficient on GPUs, as all the data must transit through the (relatively) slow PCIe bus.

In this thesis, we aim to propose new solutions to achieve better memory efficiency when the data size exceeds the memory capacity of the system. We, hence, put a particular emphasis on the memory constraints, which are often overlooked at the expense of computational performance.

1.1 How High-Performance Computing is key to CFD simulations

High-Performance Computing (HPC) is a field of computer science that deals with the development of algorithms and software to solve complex problems on large-scale systems. HPC systems are used in a wide range of fields, from scientific research to industrial simulations. One such field is CFD, where the requirements for computational power and memory can be extremely high due to the complexity of some simulations.

To be able to simulate these scenarios accurately, it is essential to have access to powerful computing resources and to use them efficiently. This is why the HPC community has developed a wide range of tools and techniques to help researchers and engineers make the most of the available hardware. In the context of CFD simulations, HPC can help reduce the time and cost of simulations, allowing researchers and engineers to explore more scenarios and make better decisions.

1.2 Motivations of this thesis

GPUs are a popular choice for accelerating CFD simulations due to their high computational power and energy efficiency. However, despite their computational power, GPUs have a limited amount of memory, which can be a bottleneck for large simulations. This limitation entices the CFD community to lean towards using the most recent GPUs, which have more memory.

In modern data centers, the trend is to include GPUs with a large amount of memory, such as the NVIDIA A100 GPU, which can have up to 80 GB of memory, or the more recent NVIDIA H100 GPU, which can have

up to 188 GB of memory. However, even with such capacities, large-scale CFD simulations can still exceed the available memory. For instance, a 3D grid of $3696 \times 3696 \times 3696$ cells with single-precision floating-point numbers would overflow the memory of an NVIDIA H100 GPU. In practice, increasing simulation sizes can easily lead to memory overflows, even on the most recent GPUs. Even if the grid fits within the memory of a recent GPU, an argument can be made regarding the price efficiency of using such GPUs.

GPU	Memory Size	Price	Price/GB
NVIDIA GeForce RTX 4080	16 GB	1390$	86.88$/GB
NVIDIA P100	16 GB	198$	12.38$/GB
NVIDIA V100	32 GB	3683$	115,09$/GB
NVIDIA A100	40 GB	7949$	198.73$/GB
NVIDIA H100	80 GB	43989$	549,86$/GB

Table 1.1: Prices of different NVIDIA GPUs, gathered on Amazon on April 9$^{\text{th}}$, 2024 and their cost per GB of memory.

Table 1.1 shows the prices of different NVIDIA GPUs and their cost per GB of memory. This is an approximate measure, as GPU prices can vary depending on the retailer and the region. Moreover, a significant portion of a GPU's price is due to the computational power it provides, not just the memory. However, it gives an idea of the cost of memory on different GPUs. We see that on data center GPUs (P100, V100, A100, H100), the cost per GB of memory increases with the memory size and the generation of the GPU. The GeForce RTX 4080, a consumer GPU, is shown for comparison. Although it is less "price efficient" than the P100, it is still better than the other GPUs.

Given these price differences, one can wonder if it is worth buying a recent GPU with a large amount of memory rather than multiple lower-end GPUs. The same consideration applies to the computational capabilities of the GPUs. The more recent GPUs are more powerful, but the price per GFLOP is also higher. This raises questions about the rationale for using the most recent GPUs for CFD simulations, as the cost of memory (and computational power) is higher. However, making efficient use of hardware can be challenging, and the cost of developing efficient software can be higher than the cost of the hardware itself. This is why the HPC community invests significant effort in designing efficient "generic" methodologies to allow larger projects to make better use of the available hardware.

In this thesis, we follow this trend by focusing on reducing the memory footprint of CFD simulations. While most of the existing literature does not tackle the issue of system memory limitations, there are practical cases where the memory footprint is the limiting factor [298, 103]. We believe that reducing the memory requirements of CFD simulations not only unlocks the potential for larger simulations but also allows for more efficient overall simulations due to reduced data movement and cache pressure. We hope that the concepts developed throughout this thesis will be exploited by the CFD community to design more memory-efficient simulations and thereby relieve the pressure on hardware.

1.3 Contributions

This thesis examines the computational and memory challenges in high-performance computing environments, with a focus on conducting CFD simulations on multi-GPU systems. We begin with a detailed analysis of various fluid simulation methods and their computational implementations, setting the stage for the subsequent exploration of advanced concepts.

One of the primary obstacles in leveraging GPU architectures for CFD simulations is their restricted memory capacity, which limits the scale of simulations. In contrast to conventional strategies that predominantly rely on Adaptive Mesh Refinement (AMR) for memory management, our research proposes an innovative method utilizing explicit data compression to significantly decrease the memory requirements of CFD simulations. This method not only tackles the pressing issue of memory constraints but also introduces new strategies for handling memory in high-resolution simulations.

Our exploration into achieving high performance on GPUs includes a thorough examination of stencil computation optimizations. We emphasize the role of task-based runtime systems, particularly PaRSEC and StarPU, in improving the efficiency of task scheduling and execution. Key achievements include the

introduction of a new feature for the PTG DSL in PaRSEC and the creation of a generic stencil solver for StarPU, enhancing the capabilities of distributed, multi-GPU environments for CFD simulations.

Additionally, we present AutoHeteroprio, an innovative scheduler for StarPU designed to automatically determine task priorities using heuristic algorithms, optimizing resource use. This scheduler has been evaluated against existing systems, showing enhanced performance in various conditions, which highlights its effectiveness in heterogeneous computing settings.

A significant contribution of this thesis is the design of a high-performance compression scheme tailored for CFD simulations. Considering the specific requirements of CFD simulations, such as mass conservation and polynomial filtering, we customized our compression techniques accordingly. The implementation of wavelet-based compression schemes represents a key advancement, demonstrating potential for substantial memory savings without compromising the accuracy of simulations.

Further, we optimized wavelet-based compression for use in GPU shared memory, aiming for an ideal balance between compression ratio and computational performance. This work addresses the memory limitations of contemporary GPUs and sets new benchmarks for executing extensive CFD simulations.

In summary, this thesis offers a comprehensive strategy for improving the efficiency and feasibility of large-scale CFD simulations on multi-GPU systems. By addressing both the computational and memory challenges, we provide a solid foundation for future research, with the goal of enabling more complex and accurate fluid dynamics simulations in various scientific and engineering fields.

Chapter 2

Scientific groundwork and current State-of-the-Art

2.1 Numerical analysis

2.1.1 Numerical discretization

Numerical discretization is a cornerstone methodology in CFD that translates the physical phenomena of fluid flow into a format that can be processed by computers. This technique involves breaking down the continuous domain of fluid motion into a discrete set of points or cells, enabling the approximation of fluid dynamics equations, such as the Navier-Stokes equations, through numerical methods. The essence of numerical discretization lies in its ability to convert complex differential equations governing fluid flow into algebraic equations that are solvable using computational resources. Without numerical discretization, it would be tedious, if not impossible, to simulate the intricate behavior of fluid flows in real-world applications.

The concept of discretization applies to different aspects of numerical simulations:

- **Discrete Domain:** The continuous fluid domain is divided into a finite number of small, discrete elements or volumes. For spatial discretization, these are often referred to as grid points (in the case of finite difference methods) or cells (in the context of finite volume methods). The choice of grid size and topology significantly influences the accuracy and computational cost of simulations;
- **Temporal Discretization:** Similar to spatial discretization, the time domain is divided into discrete intervals. The simulation progresses by calculating the state of the fluid at each time step, advancing from initial conditions towards the solution over time;
- **Discretization Schemes:** There are several approaches to numerical discretization, each with its own merits and applicability. The Finite Difference Method (FDM) approximates derivatives at grid points using differences between neighboring points. The Finite Volume Method (FVM) integrates conservation laws over discrete volume elements, ensuring the conservation of physical quantities. The Lattice Boltzmann Method (LBM), on the other hand, models fluid flow through the interactions of fictitious particles on a lattice, offering advantages in dealing with complex boundaries and multiphase flows.

Throughout this work, the simulation domain is discretized into a regular grid of cells, with each cell representing a volume element. This choice of topology is common due to its simplicity and efficiency, but there are other discretization methods that we do not consider in this work. We consider a $L_x \times L_y$ domain, where L_x and L_y are the lengths of the domain in the x and y directions, respectively. The domain is divided into $n_x \times n_y$ cells, resulting in a grid spacing of $\Delta x = \frac{L_x}{n_x}$ and $\Delta y = \frac{L_y}{n_y}$ in the x and y directions, respectively. For the sake of simplicity, but also for convenient numerical properties, we assume that $\Delta x = \Delta y$. Hence, we have a grid of small square cells $C_{i,j}$:

$$C_{i,j} =]i\Delta x, (i+1)\Delta x[\times]j\Delta y, (j+1)\Delta y[, \quad i \in [0, n_x - 1], j \in [0, n_y - 1], \tag{2.1}$$

assuming that the origin of the domain X is in the square $]0, L_x[\times]0, L_y[$. Naturally, these definitions can be extended to the 3D case by adding a third dimension z and a grid spacing Δz.

Each cell $C_{i,j}$ contains a set of conservative variables $W_{x,y}$ that represent the physical properties of the fluid at that location. The conservative vector $W_{x,y,t}$ is a function of the space variables x and y and the time variable t and is the unknown of the system. The conservation laws are expressed as a partial differential equation (PDE) of the form:

$$\partial_t W + \nabla \cdot (Q(W)) = 0, \tag{2.2}$$

where $\nabla\cdot$ is the divergence operator and $Q(W)$ is the flux of the system of conservation laws.

The divergence operator is defined by

$$\nabla \cdot (Q(W)) = \frac{\partial}{\partial x} Q^x(W) + \frac{\partial}{\partial y} Q^y(W), \tag{2.3}$$

where Q^x and Q^y are two application from $\mathbb{R}^m$ to $\mathbb{R}^m$.

Then, depending on the numerical method used, we define a set of discrete shift vectors N along the directions of the grid which are used for approximating the flux of the system of conservation laws. If we use a regular time step Δt, the W vector can be represented as $W^n_{x,y}$, where n is the time step. The solution at time step $n+1$ can then be approximated using a formula of the form:

$$W^{n+1}_{x,y} = \varphi(W^n_{x+N_{x,1},y+N_{y,1}}, W^n_{x+N_{x,2},y+N_{y,2}}, \ldots, W^n_{x+N_{x,d},y+N_{y,d}}), \tag{2.4}$$

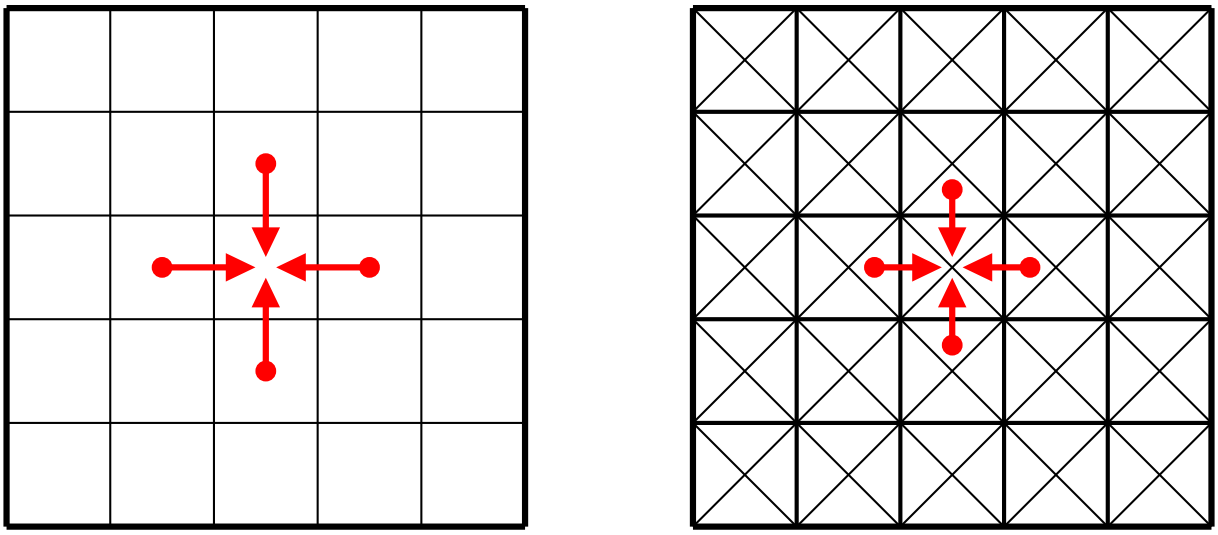
where φ is a function from $\mathbb{R}^{m\times d}$ to $\mathbb{R}^m$ and d is the number of velocities in N. This formulation varies greatly depending on the numerical method used, but the general idea remains the same: the value of each cell is computed based on the values of its neighbors.

Throughout this work, we have used two numerical methods: the Finite Volume (FV) method and the Lattice Boltzmann Method (LBM). The FV method is close to the framework we have just described and is widely used in CFD simulations [93, 157, 149]. In the LBM framework, the formulation is different, as it is based on a physical kinetic interpretation of the Navier-Stokes equations [67, 294, 182]. The literature on LBM is too vast to be summarized here. We, hence, refer to the synthesis book of Succi [256] for a general introduction to LBM. The approach we use in our experiment is based on the BGK (Bhatnagar-Gross-Krook) model, which offers multiple practical advantages in practice [38]. The contributions by Bouchut et al. [47, 48] and Aregba-Natalini et al.[17] are also of particular interest to delve into this approach. Additionally, the work of Krüger et al. [139] provides a comprehensive overview of the LBM method and its applications. For the purpose of this thesis, the important aspect of LBM is that the implementation relies on a vectorial kinetic representation, rather than the conservative vectors W. This has computational implications that are discussed in Sections 2.2.1 and 3.2.3.

Overall, what is important for high-performance computing is the similarity between the different numerical methods. Formula 2.4 shows a clear pattern for memory accesses. At each time step, the new value of each cell is computed based on the values of its neighbors and/or its own value. This type of algorithm falls into the class of stencil algorithms, also known as Iterative Stencil Loops (ISL) when performed iteratively. In this work, we will always assume that the data is stored in a regular grid and that the computation is done performing a stencil computation on this grid. The next section will delve into the specifics of ISL and explain more practical aspects of this type of algorithm.

2.1.2 Iterative Stencil Loops

Iterative Stencil Loops (ISL) constitute the focus of this work. ISL are a class of algorithms that are widely used in scientific computing, particularly in the context of CFD simulations. ISL are characterized by their regular data access patterns and their reliance on local information to compute the values of the cells in the grid. The basic idea of ISL is to iterate over a regular grid of data, and at each iteration, to compute the value of each cell based on the values of its neighbors. This computation is typically done using a fixed set of operations, which are applied to each cell in the grid.



(a) One value per cell. (b) 4 values per cell.

Figure 2.1: Examples of 2D stencils. The start of the arrows indicate the data that is read, while the end of the arrows indicate where the result of the stencil is stored. Example 2.1a shows a simple 2D stencil, where the computation of each cell depends on its four neighbors. Example 2.1b shows an example where each cell has four values (here, potentially the kinetic fluxes) and the computation of each cell depends on one value from each of the four neighbors.

Figure 2.1 shows two examples of 2D stencils, which illustrate the basic idea of ISL. We can see that there is an interconnected pattern of data dependencies, which makes it impossible to perform the computation in place in the general case. Instead, a common approach is to use two grids, one for the input data and one for the output data, and to alternate between them at each iteration. Some stencils, however, can be computed in place due to particular data access patterns or strategic use of intermediate results.

The regularity of the data access patterns in ISL makes them well-suited for parallel execution on modern hardware, such as multi-core CPUs and GPUs [306, 303, 297]. In particular, GPUs are well-suited for running ISL, as they are designed to efficiently execute large numbers of parallel threads. The regularity of the data access patterns in ISL allows for efficient memory access and data reuse, which are crucial for achieving high performance on GPUs.

The basis of the ISL is the stencil, which indicates the pattern of data access and computation that is applied to each cell in the grid. The stencil is defined by a set of offsets, which indicate the relative positions of the neighbors of a cell. Depending on the shape of the stencil, the used hadware and the specific problem being solved, different optimization strategies can be applied to improve the performance of the ISL. In general, these optimization strategies rely on improving the efficiency of memory access or data reuse. Chapter 3 provides various strategies and examples of stencil optimizations.

As the optimizations are problem-specific, a common approach is to rely on automatic optimization tools to generate efficient code [232, 287, 240, 242, 241, 158]. Among these tools, some rely on DSLs (Domain Specific Languages) to precisely describe the computation and the data access patterns. We refer, for instance, to AnyDSL [1], Forma [221], Devito [148], Snowflake [305], and Halide [213]. The use of specialized DSLs is the most common approach for implementing efficient stencils. Some drawbacks of using them include the difficulty of integrating them into existing codebases, the cost of learning a new language, and the lack of adaptability to particular behaviors that are not captured by the DSL. While the use of (at least partly) automatic tools is discarded in our work, the concepts developed in this thesis can be used in conjunction with these tools to further improve the performance of ISL.

The stencil approach has several advantages, but also multiple drawbacks. From the perspective of physical simulations, using a stencil approach can be inefficient for capturing particular fluid behaviors. For example, if the fluid velocity follows a particular and known pattern, it can become more effective to use a non-uniform mesh. Another example is when the fluid describes complex behavior in a small region of

the domain, while the rest of the domain is relatively simple. In this case, the most common approach is to use an Adaptative Mesh Refinement (AMR) method, which is described in Section 2.2.2. Finally, an important drawback of the stencil approach is that the required memory grows greatly as the resolution of the grid increases. Due to the regular structure of the grid, the memory usage grows quadratically (in 2D) or cubically (in 3D) with the resolution of the grid, which can become a limiting factor for large-scale simulations. Hence, the memory requirements can rapidly become the bottleneck of stencil-based simulations. This latter drawback is the central focus of this work. In the following section, we present the state of the art of techniques that aim at simulating large-scale CFD simulations in memory-constrained environments.

2.2 Reducing the memory footprint of CFD simulations

In this section, we delve into the current landscape of techniques that aim to increase the scale of CFD simulations that can be performed for a given amount of memory. Some of these techniques use explicit data compression algorithms. We detail these techniques in Section 2.2.3. However, using explicit data compression methods can be too costly in terms of performance for large-scale simulations. This is why most approaches adopt strategies that achieve effective memory gains without using explicit data compression. We present the most prevalent of these techniques in Section 2.2.1. The gains achieved by these techniques are often limited due to the regular structure of the grid and the inherent memory requirements of stencil-based algorithms. This is why a wide range of research has been conducted on designing more advanced numerical scheme that make better use of the available memory. The most popular framework for this is Adaptive Mesh Refinement (AMR), which is presented in Section 2.2.2.

2.2.1 Memory-efficient CFD Simulations

When the memory requirements approach the limits of available memory, it becomes important to focus efforts on minimizing the memory usage of the algorithm. Several strategies can be employed to accomplish this objective. Some of these strategies are tailored to specific numerical methods, whereas others have broader applications. In this section, we discuss the most prevalent approaches employed in practice to secure substantial memory savings, without altering the fundamental nature of the numerical method.

Firstly, a factor of 2 can be saved by using in-place computation. Indeed, without in place computation, two grids are required: one for the input data and one for the output data. In-place computations are not always possible and can be more complex to implement and costly in terms of performance. The LBM framework, for instance, is well-known for its potential to perform in-place computations. In Section 3.2.3, we present a naive implementation of an LBM scheme and briefly explain how it can be adapted to perform in-place computations. Welein *et al.* present this naive approach in its two forms: the *pull* and *push* approaches [289]. A natural extension of this approach, introduced by Bailey *et al.*, consists of using the *pull* version on even time steps and the *push* version on odd time steps (or vice versa) [23]. This extension allows to perform in-place computation and is referred to as the *AA-pattern* in the literature. Subsequently, Geier *et al.* introduced another in-place computation technique inspired by works from Neumann *et al.*, dubbed the *esoteric twist* [196, 104, 154]. The literature extensively discusses the advantages and drawbacks of different access patterns and their implications in terms of performance and memory usage [190, 114]. In this work, we choose to remain agnostic to the specific access pattern and do not assume in-place computation. We, however, acknowledge that some of our results could be further improved by using scheme-specific knowledge.

When in-place computations are not possible, another approach involves segmenting the grid into subgrids and performing computations on each subgrid separately. This technique allows for the grid to be stored in memory only once. The only additional memory required is for storing an extra subgrid for subgrid-level computations and the interfaces between subgrids. A more comprehensive discussion on this method is provided in Section 3.3. Though commonly applied to distribute stencil computations across multiple processing units, multiple studies have examined its effectiveness in reducing memory usage on single nodes [151, 278]. In Chapter 9, we make the similar obervation that effective memory gains can be achieved thanks to this method (despite not being the primary objective of the work). Valero-Lara extend this concept by optimizing the entire computational process for GPUs, achieving near-optimal performance [279].

Several studies have explored the use of mixed-precision arithmetic in the context of numerical simulations [155, 176]. The main goal of mixed-precision arithmetic is generally to use the hardware capabilities

more efficiently, but they can also lead to memory savings. These approaches generally rely on scheme-specific knowledge and use lower precision for parts of the computation that do not require high precision. We refer to the work of Antz *et al.* [2] and Higham [115] who provide a broad survey of the state of the art in mixed-precision arithmetic for scientific computing. The problem of mixed-precision approaches is that they can be complex to implement and require an understanding of the numerical scheme, for a modest gain in memory savings.

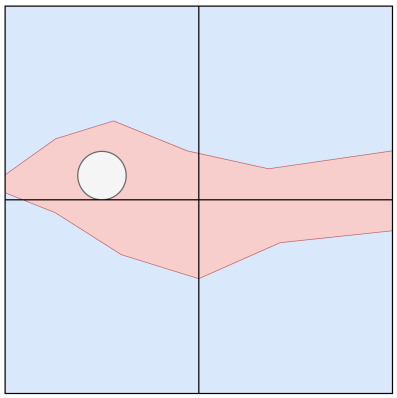
Overall, all the strategies discussed in this section are effective for achieving memory savings without compromising the core principles of the numerical method. They are advantageous due to their simplicity in implementation and minimal computational overhead. Nevertheless, the extent of memory reduction attainable through these strategies is often modest. To realize more significant memory reductions, more advanced techniques must be employed. A prevalent strategy involves altering the numerical scheme itself to optimize memory utilization. The next section will present the state of the art in this area, focusing on the Adaptative Mesh Refinement (AMR) framework.

2.2.2 Adaptative Mesh Refinement

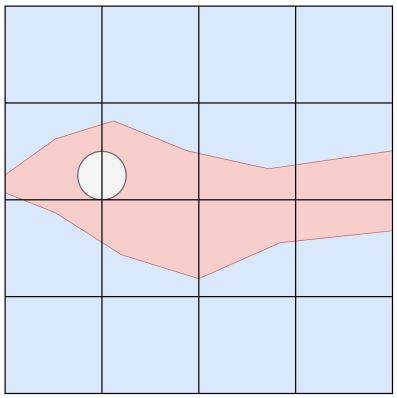
Even though our approach does not use Adaptative Mesh Refinement (AMR), we briefly review the state of the art of this method, as it is the most popular technique in computational fluid dynamics for compressing the computational domain. In this section, we aim to present the method, its advantages and drawbacks, the state of the art, and how it differs from our approach. It is important to keep in mind that it is a well-established technique whose literature is extensive and that we do not aim to provide a comprehensive review of the subject.

AMR techniques are commonly used to achieve "compression" in different numerical methods, such as FV and LBM schemes. We refer for instance to [192, 73] for some works using AMR in a FV context and to [300, 33, 34] for some works using AMR in a LBM context. We also refer to work of Harten [110] for a more generic approach to multiresolution schemes.

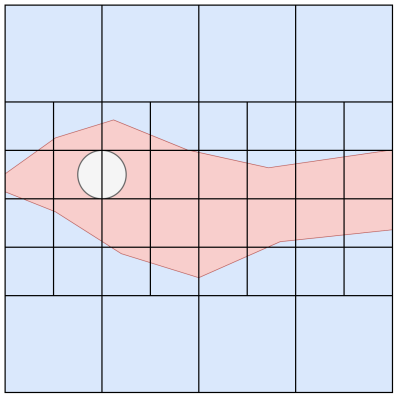
This is done by refining the mesh in regions where high resolution is required, while coarsening it in regions where lower resolution is sufficient. The refinement and coarsening of the mesh are done dynamically during the simulation based on some predefined criteria, such as gradients in the flow field or concentration field. The simulation space is then represented as a hierarchy of cells with varying levels of resolution, typically using a quadtree in 2D simulations and an octree in 3D simulations. While AMR can significantly reduce the amount of computational tasks and the memory footprint of CFD simulations, it requires additional effort to implement and maintain. In particular, it requires careful management of data structures and communication between different levels of refinement [192]. It is often stated that the primary goal of AMR is to reduce the computational cost of the simulation, but it also de facto reduces its memory usage. But the complexity of the implementation and the irregular memory access may often lead to disappointing speedups. The parallel load balancing of the AMR approach is also a delicate task [56, 62].



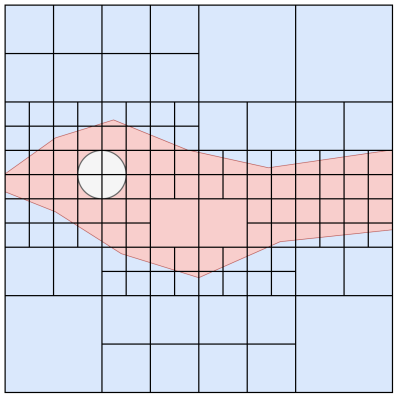
(a) Max depth = 1.



(b) Max depth = 2.



(c) Max depth = 3.



(d) Max depth = 4.

Figure 2.2: Illustration of the quadtree structure on a fake simulation of a fluid with two different states (red and blue) and a cylinder within the red region. The interfaces between the red and blue regions are defined as the region of interest (ROI) and are refined to a higher resolution than the rest of the domain. We show four different quadtrees with different maximum levels of refinement and we impose a face-balanced quadtree, there must be at most one refinement level difference between adjacent cells.

To illustrate the AMR method, consider Figure 2.2, which shows a fake simulation of a fluid with two different states (red and blue) and a cylinder within the red region. In this fake simulation, we identify two regions of interest (ROI): one at the boundary of the cylinder and the other between the red and the blue regions.

In a real simulation, this would mean that the behavior of the fluid in these regions is more critically important to the overall precision of the simulation than the rest of the domain. Figures 2.2a, 2.2b, 2.2c and 2.2d show the quadtree structure of the domain at different maximum levels of refinement. As we can see, the two regions of interest end up being refined to a higher resolution than the rest of the domain. By using a quadtree structure, AMR allows for the efficient allocation of computational resources by refining only the areas where it is necessary to achieve the desired level of accuracy, while keeping the rest of the mesh coarser. Some works, such as the one from Abgrall [4], use AMR approaches with unstructured meshes but are not applicable to the stencil-based approach.

AMR can be classified into two categories: *block-based* and *cell-based* [254]. In block-based AMR, the leaves of the quadtree correspond to blocks of cells, whereas in cell-based AMR, each leaf represents an individual cell [192, 73, 33, 34]. This distinction is crucial because it affects both the storage and communication of data between different refinement levels. In block-based AMR, inter-block communications are

typically done by duplicating data at the boundaries of blocks using ghost cells. Intra-block communication is typically performed implicitly by the stencil kernel. On the other hand, in cell-based AMR, all the communications with the neighbors rely on local analysis of the quadtree structure [299], making it more flexible but potentially at the cost of increased communication costs due to less regular data access. Overall, block-based AMR is more commonly used in LBM solvers, as the regularity within each block enables efficient kernel design. In this review, we only consider block-based AMR: each leaf of the quadtree corresponds to a contiguous block of cells (not an individual cell).

Although AMR is a well-established technique, it has its own drawbacks. In the following, we discuss three of the main challenges associated with AMR: designing the numerical scheme, handling the hierarchical structure and implementing it on GPUs.

Designing the numerical scheme As the mesh can be refined and coarsened dynamically, the numerical scheme must be designed with the AMR method in mind. The primary challenge lies in handling interfaces between cells at different levels of refinement, complicating the design for a physically consistent numerical scheme. Additionally, scheme-specific challenges and technicalities often arise, further complicating the implementation of AMR [197]. Consequently, while AMR represents the state of the art for achieving an optimal balance between accuracy and computational cost, it is significantly more complex to implement than a uniform grid.

Another challenge is defining regions of interest and the criteria that represent them. These criteria, often based on the physical properties of the fluid, are defined empirically, making it difficult to explain low accuracy in the results and ensure a desired property for the simulation.

Overall, there are numerous reasons why not all simulations can benefit from AMR. Despite being a better trade-off between accuracy and computational cost, AMR also has its own drawbacks, which is why it is generally only employed when performance is critical. This is why the stencil-based approach remains the most common method for CFD simulations.

Handling the hierarchical structure AMR relies on the use of a hierarchical data structure to store the different levels of refinement. In the classical implementation, the mesh is represented by a balanced quadtree or octree. The balance property ensures a 2:1 or 1:1 correspondence between cells at adjacent refinement levels. Some works aim to deviate from this paradigm to achieve better performance. For example, Freret et al. [100, 101] propose an anisotropic hierarchical structure. Unlike a quadtree, this anisotropic structure can contain non-square cells and allows for a more flexible refinement. In another work, Sætra et al. [224] use a hierarchical tree with few structural constraints. The used structure is extremely flexible and allows them to run an entire simulation on a GPU. Both works, however, ensure a 2:1 correspondence at most between cells at adjacent refinement levels.

This 2:1 correspondence is a strong convention that is widely used in the field of AMR due to its convenience and effectiveness. It will inevitably impose limitations on how the space can be refined. The example shown in Figure 2.2d illustrates one such limitation, where the top-left node of Figure 2.2c is divided into four subcells to maintain a face-balanced quadtree structure. This limitation (caused by the 2:1 correspondence requirement) can impact the efficiency of the simulation, as it can cause over-refinement in some regions of the domain due to their proximity to a highly refined region. This can result in unnecessary computational cost and memory usage. While it may be possible to mitigate these types of limitations to some extent, it is difficult to completely avoid them while maintaining the benefits of the hierarchical structure.

Finally, the irregular nature of the data caused by the hierarchical structure can make it difficult to efficiently distribute the workload. Multiple works propose different load balancing strategies to alleviate this issue [121, 210, 150, 56, 61]. The main challenge is to find a reasonable strategy that is cheap in terms of communications.

Overall, the main technical challenge associated with AMR is to reduce its computational overhead, in particular for large-scale simulations. Numerous works propose efficient algorithms to alleviate the overhead induced by the AMR method [230, 111, 245]. Thanks to a careful communication and structural design, these studies show that AMR simulations can exhibit good scalability, with a speedup of approximately x1.5 for each doubling of computational resources.

AMR on GPUs Implementing AMR on GPUs can be challenging due to several factors. One of the main challenges is the irregularity of the data caused by the hierarchical structure used by AMR. This irregularity can make it difficult to efficiently map the data to the GPU architecture, which is optimized for regular and homogeneous data access. Schive et al. [234] propose a hybrid CPU-GPU implementation of the AMR method for astrophysics (magneto-hydrodynamics) simulations. They use an octree structure that is kept up-to-date on the CPU, while the GPU is used to perform the physics computations. Sætra et al. [224] provide a 100% GPU implementation of the shallow-water equations using AMR. In their work, they use a general tree where each node contains the grid (tile) corresponding to their whole logical space (children included). The communication between nodes is ensured through the use of ghost cells in each tile. The regridding process (coarsening and refinement) is performed by computing a criterion for each tile. To compute the criterion efficiently, they use the GPU shared memory to perform a reduction operation on the tile. Overall, they achieve good hardware utilization and achieve speedups of more that x2.5 on their simulation, while preserving the accuracy of the results.

With the Daino framework, Wahib et al. [288] propose an automatic AMR GPU code generation. This high-level framework uses a compiler-based approach to automatically transform serial uniform mesh code annotated by the user into parallel adaptive mesh code optimized for GPU-accelerated hardware. The authors demonstrate the efficiency of Daino automated transformations by comparing the execution time of the generated code with that of hand-written AMR code. They also highlight the potential productivity gains of Daino by showing that auto-generating AMR code requires negligible lines of code compared to hand-written AMR implementations.

It is worth noting that various works aim at improving the performance of AMR on GPUs by taking advantage of the particular properties of a scheme [228] [235]. However, investing in the development of a specific AMR implementation for a given scheme is time-consuming and often not relevant for a project. Indeed, a more generic approach that is fast to implement while reaching a decent utilization of the available resources can be preferable.

In summary, the hierarchical structure required by AMR offers a powerful framework for efficiently distributing the computing power over a simulation space. However, it comes with a cost in terms of implementation and maintenance. A majority of existing numerical scheme implementations are based on a stencil approach, use a uniform grid, and are, therefore, not compatible with AMR.

In that respect, our approach is more convenient, as it does not require a hierarchical structure. Instead, our approach relies on a compression scheme that compresses a uniform grid. Therefore, it is directly compatible with numerous existing schemes that use the classical stencil approach. Finally, it is important to note that our approach is also fundamentally compatible with block-based AMR, as each (uniform) block can be treated independently.

2.2.3 Entropy coding and data compression

In CFD, data compression is typically used through multiscale analysis. We have seen in the previous section that this method induces a layer of complexity both in the mathematical model and in the implementation. Another approach is to use explicit data compression methods to reduce the memory footprint of the simulation.

The use of data compression in scientific computing has been a topic of interest for several years. Cappello *et al.* conducted a thorough study to assess the relevance of using lossy compression in scientific computing [63]. They identify use cases where lossy compression is advantageous, particularly when it comes to addressing memory bandwidth and storage limitations of computational hardware. In scenarios where memory bottlenecks significantly constrain the simulation, employing compression to alleviate these bottlenecks can be more advantageous than the overhead introduced by the compression process.

The idea of incorporating data compression in CFD simulations has been explored in several studies that we will survey in the following sections. We will refer to the concept of entropy, which is a measure of the information content of a message, as defined in *information theory* [246]. Because CFD data are not random, the entropy of the signal can be reduced by making assumptions about the data structure. The scientific literature on this topic differs from that of general data compression in that it aims to find the best assumptions on the signal for a given class of simulations to achieve the best compression ratio and/or

compression speed. In Section 2.2.3, we present a general overview of data compression in fluid simulations. Then, in Section 2.2.3, we present the current state of the art in the use of the Discrete Wavelet Transform in CFD data compression.

Data compression in fluid simulations

Fluid simulations often generate massive datasets, necessitating advanced data compression techniques to address memory and computational challenges. Compression techniques can be classified by several criteria, such as the nature of compression (lossy/lossless), processing mode (streaming/offline), the employment of predictive models, reliance on floating-point representations, and specific spatial structures considerations. Such categorizations facilitate a better understanding and comparison of different techniques within fluid simulations.

Lossless compression methods, being reversible, preserve the integrity of data, ensuring no impact on the physics simulation results. Conversely, lossy compression leads to some information loss. While these techniques can achieve higher compression ratios, the design of lossy compression must be meticulous to maintain result accuracy. One strategy for managing data loss in simulations is to identify regions of interest (ROIs) and permit data loss primarily in areas outside these specified regions [195]. Machine learning is an option for achieving lossy compression but offers limited guarantees on the accuracy of the results. It has, however, demonstrated potential for in situ visualization of CFD data [169].

Streaming compression methods, designed for real-time data processing, are preferred in large-scale simulations due to their reduced memory footprint. A rudimentary example of compression is the mixed-precision representation. This method provides limited compression ratios, while considerably impacting accuracy, rendering it less effective compared to other techniques [155]. A more effective approach is to use a generic compression algorithm that is known to be efficient on the used hardware (here, GPUs) [231, 204, 167, 250, 134, 304]. This approach benefits from incorporating domain-specific knowledge, which reduces the entropy of the data [162]. Typically, the streaming compression approach encompasses:

- **Predictor**: Estimates data from previously encoded points.
- **Difference Operator**: Computes the difference between the predicted and actual value, producing the *residual*.
- **Residual Coder**: Encodes residuals for compression, often using entropy coding.

In the context of fluid simulations, innovations in streaming data compression often originate from enhancements in these components [92, 217, 164]. Using prediction/difference pretreatments aims to reduce data entropy, making it more compressible. By allowing one of these components to be lossy, it becomes possible to further reduce entropy, often by setting a threshold and encoding residuals that exceed it.

Transitioning from streaming to offline compression, we explore techniques that process data in chunks or entirely. These techniques often achieve superior compression ratios but can demand more memory and sometimes entail higher computational costs. The Lorenzo predictor, which uses neighbors across N dimensions for data prediction, serves as a notable example of offline compression for CFD data [120, 168]. Though it is categorized as offline due to its multidimensional data access, with the right implementation, its memory usage can remain low. Other techniques that employ multidimensional prediction exist as well [99]. While these approaches primarily focus on compression speed, other methods aim to achieve higher compression ratios. The Discrete Wavelet Transform (DWT), known for its high compression ratios with CFD data, exemplifies this approach. The subsequent section discusses the Discrete Wavelet Transform, its role in CFD data compression, and reviews relevant studies on the topic.

Discrete Wavelet Transform

The Discrete Wavelet Transform (DWT) is widely recognized across various fields, especially in CFD data compression [192, 73, 33, 34]. In this context, wavelets with compact support, biorthogonality, and a design tailored for a multi-resolution approach are favored. These attributes enable the DWT to process both frequency and spatial data efficiently. DWT decomposes an input signal into two main components: the approximation, which captures low-frequency information, and the detail, representing high-frequency

nuances. This decomposition is performed iteratively, with each level of the DWT capturing a different frequency band. At the end of this process, the low details can be discarded, hence introducing loss, and fed to an entropy coder to achieve high compression ratios. In Chapter 7, we provide a more technical overview of the DWT and how it can be used in practice.

Researchers have extensively explored the use of DWT in CFD data compression [274, 129, 236]. These investigations establish the foundational knowledge supporting the use of DWT in CFD data compression. Because of its significant memory demand, DWT is not typically utilized as a standalone compression method [243]. Instead, it serves as a tool to manage different resolution levels in data, optimizing computational resource allocation. Prominent multi-level methods, such as AMR [35] and multigrid methods [53], share this concept but do not incorporate the DWT directly. Other methods explicitly integrate different DWT levels within their framework [108, 226, 227].

One foundational study by Cohen *et al.* [73] conducts the CFD computations directly on the adaptive wavelet structure. They perform a careful mathematical analysis showing that this approach is almost optimal in terms of memory occupation and algorithm complexity. This optimality is obtained thanks to the excellent compression and approximation properties of the wavelet transform and also to its suitability to hyperbolic conservation laws, where local perturbations propagate at finite speed. However, this mathematical analysis is rather theoretical and does not take into account the very irregular structure of the sparse wavelet representation. In practice, handling this structure generates an unacceptable overhead on modern GPUs, because of non-coalescent memory access. In this paper, we propose a more pragmatic approach which harnesses the compression rate of the DWT, but keeps as much as possible the very efficient memory pattern of the LBM stencil pattern.

The use of wavelets in explicit data compression is also very common. However, traditionally, the considerable overhead of DWT has made these methods more suitable for storing or visualizing results, rather than for direct compression of simulation data [43]. Many of these techniques are inspired by or based on the JPEG2000 standard [102]. The advent of modern GPUs, with their architectural improvements, marks a shift in this domain. These advancements enable more efficient use of DWT, making it a viable option for in situ compression in performance-critical simulations [212]. Our research aligns with this trend, aiming to leverage the DWT for in situ compression of CFD data on GPUs. In Chapter 9, we show that tuning the DWT for modern GPU architectures can achieve significant memory savings with little computational overhead.

2.3 Discussions

In this chapter, we have first introduced the framework of numerical discretization for CFD simulations. We have then explained that in cases when the underlying physical model uses a regular grid structure, the solving of the equations can be performed using a stencil-based approach. This approach is particularly well-suited for parallel execution on modern hardware, including GPUs. However, it also introduces challenges, such as the high memory requirements of the regular grid structure. A significant portion of our research addresses this challenge, notably in Chapters 7, 8, and 9.

The rest of our work is dedicated to performing efficient distributed CFD simulations on GPUs. Optimizing the performance of stencil-based computations is a topic of high interest, as it applies to a wide range of scientific and engineering applications. This aspect is already covered in a wide variety of works, which propose different strategies to optimize the performance of stencil computations. In our study, we focus on distribution of the workload rather than on the optimization of the algorithm at a single GPU level. In Chapters 4, 5, and 6, we present our contributions to this field.

While this work does not focus on the optimization of stencil algorithm, it is important to understand the basic principles and challenges within this framework. In the next chapter, we will present a technical discussion on how to perform efficient stencils on GPUs, as well as what framework is classically adopted for distributing the work. This chapter will serve as a basis for the following chapters, where we will present our contributions to the field of stencil computations on GPUs.

Chapter 3

Efficient fluid simulations on GPUs

As we have seen in the previous section, CFD simulations can often be implemented as stencil loops. All stencil-based algorithms share common characteristics, such as the need for data locality, and the need for efficient data transfers. In this section, we present the most common challenges that arise when implementing stencils on the GPU. We adopt a general approach, as the presented concepts are applicable to multiple shapes of stencils and grid layouts.

3.1 GPU Architecture and Processing Model

Graphics Processing Units (GPUs) are a type of processor that is optimized for data-parallel workloads. As their name suggests, they were originally designed for rendering graphics, but they have since been used for a wide variety of general-purpose computing tasks (GPGPU). In particular, the early 2010s saw a growing trend of using GPUs for scientific computing, and among them, for fluid simulations [57, 76]. This trend was driven by the increasing computational power of GPUs, which was not matched by the increase in CPU performance. However, GPUs use a different architecture and processing model than CPUs, which requires a different approach to programming. In this section, we present the specificities of GPU architecture and how they translate into programming challenges. The presented information is based on the architecture of NVIDIA GPUs, but most concepts are directly applicable to other vendors as well.

3.1.1 Hardware Architecture

The basic unit of computation in a GPU is the Streaming Multiprocessor (SM). A GPU consists of multiple SMs that operate in parallel. SMs can be assimilated to a CPU core, as they are relatively independent from each other, but their inner workings are quite different. An SM is composed of multiple CUDA cores (usually 64 or 128), which are another layer of parallelism. Each CUDA core can operate asynchronously from the others. However, the CUDA cores of an SM remain largely interconnected, as they share resources such as the register file, the shared memory, and the instruction cache. Finally, each CUDA core contains a pair of warps, which can be seen as the basic unit of work. A warp can be seen as an SIMD (Single Instruction, Multiple Data) unit, optimized for processing vectors of size 32 (threads), even if the actual hardware does not necessarily match this view. Pairs of warps are optimized for working together, as a larger SIMD unit (of 64 threads), but they can desynchronize to an extent, for example in case of divergent control flow.

The memory hierarchy of a GPU is also quite different from that of a CPU. Firstly, the registers are not physically bound to a specific CUDA core (or warp). The registers are stored in the register file and are dynamically allocated by the warp scheduler, which assigns registers to warps as they are needed. It is important to note that the register file, while shared by all CUDA cores within an SM, is not directly accessible as addressable memory; rather, it increases the available registers per unit.

On top of the register file, each SM has a shared memory (related to the L1 cache), which is a small, fast memory that is shared by all CUDA cores within the SM. This memory, consists of SRAM (Static Random-Access Memory) and is directly addressable. It can be leveraged to achieve higher memory efficiency, or as a scratchpad memory for inter-thread communication. If the shared memory is not used explicitly by

the kernel, it will be utilized automatically as a cache, thereby being referred to as the L1 cache. It is organized in banks, which can be accessed in parallel, but there are limitations to the number of concurrent accesses that can be made to the same bank. Improper access patterns can lead to bank conflicts, which can significantly reduce the performance of the kernel.

Finally, the GPU has a global memory, which is the main memory of the device. It is much larger than the shared memory and is accessible by all SMs. However, it consists of DRAM (Dynamic Random-Access Memory), which is slower than SRAM. It is optimized for *coalesced* memory accesses, which means that the memory accesses should be aligned and contiguous. This paradigm varies slightly between different generations of GPUs, but the general principle remains the same. Accesses to the global memory are handled by the memory controller, which is responsible for scheduling the memory requests and ensuring that the memory is used efficiently. Among other things, the memory controller can use the L2 cache to reduce the number of requests to the global memory.

There are also other types of memory, such as the constant memory, which is read-only and cached, and the texture memory, which is optimized for 2D and 3D spatial locality. These types of memory are, however, rarely used in scientific computing, and we will not discuss them further.

Figure 3.1: Turing architecture (GTX 16 series and RTX 20 series), provided by NVIDIA. The datacenter equivalent is the Volta architecture (V100).

To execute a kernel on the GPU, the program is pushed as bytecode to the device and dispatched to the Streaming Multiprocessors (SMs) for execution. This bytecode is specific to the GPU architecture and is generally not analyzed by the programmer. However, there exists an intermediate representation known as PTX (Parallel Thread Execution) assembly language, which is architecture-independent and allows for analysis and optimization by the programmer, although it is often unnecessary for most applications. Instead, programmers typically develop kernels in CUDA, a C++-like language, which offers a more user-friendly interface to the underlying complexities of the GPU. Nonetheless, understanding the fundamental hardware architecture is necessary to write efficient kernels. In the subsequent section, we briefly present the CUDA

programming model and the main challenges that arise when programming for the GPU.

3.1.2 Programming Interface

The CUDA programming model is based on the separation of the host and device code. The host code is executed on the CPU, while the device code is executed on the GPU. The host code can allocate/transfer memory from/to the device, launch (device) kernels, and is responsible for the overall control flow of the program. The device code corresponds to the GPU code that is actually executed on the device.

Let us base our discussion on the example of a basic SAXPY (Single-precision A*X Plus Y) kernel. The kernel is a simple function that computes the element-wise product of two vectors, and adds the result to a third vector. A possible CUDA implementation of this kernel is shown in Code 3.1. Then, the kernel must be launched from the host code, as shown in Code 3.2.

```
__global__ // __global__ indicates that the function is a kernel
void saxpy(int n, float a, float *x, float *y, float *out) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        out[i] = a * x[i] + y[i];
    }
}
```

Code 3.1: A simple CUDA kernel that performs SAXPY.

```
    // launch 1024 blocks of 256 threads each
    saxpy<<<1024, 256>>>(N, 2.0, x, y, out);
```

Code 3.2: Launching the SAXPY kernel from the host code.

The kernel launch must specify the number of blocks and the size of the blocks (the number of threads per block). The maximum number of blocks and threads per block is limited, among other things, by the amount of shared memory, the number of registers available on the device, and the requirements of the kernel. There is also a hard limit on them, which depends on the compute capability of the device. A block is a layer of parallelism that exists within an SM and is built at the start of the kernel launch. It is composed of a number of threads, which are given a unique identifier within the block. During the execution of the kernel, the blocks are distributed to the SMs and across the CUDA cores/warps in a way that is not directly controllable by the programmer. Hence, the affected hardware is not known in advance, and the programmer must rely on the runtime system to schedule the blocks.

The SAXPY kernel is a simple example with no data dependencies between threads, and the threads can be executed independently. Hence, This example is naturally parallel and the main challenge is to ensure that the memory accesses are efficient. We can measure the effective memory bandwidth of the kernel by calculating the number of bytes read and written, and dividing by the execution time:

$$\text{Bandwidth} = \frac{\text{Bytes read} + \text{Bytes written}}{\text{Execution time}} \tag{3.1}$$

Here, we have $2N$ float values read and N float values written, where N is the size of the vectors. The effective memory bandwidth measured on a P100 GPU is approximately 545 GB/s, which is approximately 75% of the peak memory bandwidth of the device (732 GB/s). The memory accesses are efficient because the consecutive threads (in terms of thread id) access contiguous memory locations, which corresponds to coalesced memory accesses. If we replace the index `i` by `threadIdx.x + blockIdx.x * blockDim.x` (inverting the order of the two terms), as the contiguous threads will access memory addresses separated by the size of a block (multiplied by the size of the data type). In this case, the measured effective memory bandwidth is approximately 178 GB/s, which is only 24% of the peak memory bandwidth of the device. Here, the unoptimal memory accesses induce a slowdown of approximately 3x on a P100 GPU, which is an expensive device optimized for high performance computing. On more consumer-oriented devices, the slowdown is typically higher.

Now let us introduce an example where the parallelism is not as straightforward.

```
__global__ void sum_naive(int n, float *x, float *out)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // Global thread id
```

```
    atomicAdd(out, x[tid]);
}
```

Code 3.3: A simple CUDA kernel that performs a reduction.

The kernel in Code 3.3 performs a sum of all the elements of an array and puts the result in a single location. The `atomicAdd` function lets us perform an atomic addition (here, in global memory), which ensures that the final result is correct.

However, this naive implementation leads to a contention on the memory location `out`, which leads to a dramatic slowdown. A better solution is to make use of block-level mechanisms to perform a lower-level reduction before requesting a global memory access (through `atomicAdd`). One such mechanism is the shared memory, which we mentioned in Section 3.1, which is fast and shared by all threads within a block. The shared memory can be leveraged to perform a block-level reduction, the result of which necessitates only a single global memory access to be added to the final result. The shared memory is used explicitly by declaring a variable with the `__shared__` keyword.

```
__global__ void sum_block_level_reduction(int n, float *x, float *out)
{
    assert(blockDim.x <= 256); // Assume that the block size is less than 256

    __shared__ float cache[256]; // Shared memory
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // Global thread id
    cache[threadIdx.x] = x[tid];

    for(int r=blockDim.x/2; r!=0; r/=2) {
        __syncthreads();
        if (threadIdx.x < r)
            cache[threadIdx.x] += cache[threadIdx.x + r];
    }

    if (threadIdx.x == 0)
        atomicAdd(out, cache[0]);
}
```

Code 3.4: A simple CUDA kernel that performs a block-level reduction.

Code 3.4 shows a possible implementation of this idea. Since we have no guarantee that the different threads will be executed in lockstep, we need to use the `__syncthreads()` function to ensure that all threads have finished writing to the shared memory before we start reading from it. The key in this implementation is that `atomicAdd`, which is a slow operation, is only called once per block, and not once per thread. The intermediate results of the reduction are stored in the shared memory, which is fast and shared by all threads within a block.

```
__global__ void sum_warp_level_reduction(int n, float *x, float *out)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    float accumulator = x[tid];
    int tid_warp = threadIdx.x%32; // Thread id within the warp

    // Split the remaining threads in half at each iteration
    for(int r=16; r>=1; r/=2) {
        int temp = *(int*)(&accumulator); // Cast to int
        // Get the value of the accumulator from the threads with tid >= r
        temp = __shfl_down_sync(0xFFFFFFFF, temp, r);
        if(tid_warp < r)
            accumulator += *((float*)(&temp)); // Interpret back to float
    }

    if(tid_warp == 0)
        atomicAdd(out, accumulator);
}
```

Code 3.5: A simple CUDA kernel that performs a warp-level reduction.

One way to avoid using `__syncthreads()` is to perform a warp-level reduction. Code 3.5 shows a possible implementation of this idea. Warps function in lockstep, which means that all threads within a warp execute the same instruction at the same time. This lets us avoid using `__syncthreads()`, which can be expensive as it introduces a barrier. Here, the inter-thread communication is performed using the `__shfl_down_sync` function, which is a warp-level communication function. It sends the values of the `accumulator` variable

from the threads with $tid \geq r$ to the threads with $tid < r$. In this implementation, the `atomicAdd` function is called once per warp. It, hence, uses more global memory accesses than the block-level reduction, but it can still be faster in practice.

GPU	Quadro T200	P100 (16GB)
Maximum (theoretical)	128 GB/s	732 GB/s
Naive (Code 3.3)	2.4 GB/s	1.7 GB/s
Block-level (Code 3.4)	57 GB/s	**117 GB/s**
Warp-level (Code 3.5)	**77 GB/s**	54 GB/s
Array size	536 MB	2 GB

Table 3.1: Effective memory bandwidth of the different reduction kernels on two different GPUs. The kernel with the best throughput is highlighted in bold.

Choosing between the block-level and warp-level approach is not necessarily straightforward. Table 3.1 shows the effective memory bandwidth of the different reduction kernels on a Quadro T200 and a P100 GPU. The block-level reduction is faster on the P100, while the warp-level reduction is faster on the Quadro T200. Without additional investigation, it is difficult to predict which approach will be the most efficient. The usual approach is to rely on thorough benchmarking to determine the best approach for a specific problem.

The presented examples show how the CUDA programming model can be used to express parallelism in a simple and straightforward way. Knowledge of the underlying hardware is beneficial to write efficient kernels, as we have seen with the examples of the SAXPY kernel and the reduction kernels. In the next section, we will show how the aforementioned concepts can be applied to obtain reasonably efficient stencil kernels on the GPU.

3.2 Implementing a numerical scheme on the GPU

Numerous research works have been dedicated to the implementation of efficient numerical schemes on the GPU [36, 233, 25] The challenges are mostly similar when a stencil-based approach is chosen, namely the need for efficient data transfers and efficient data reuse. In this section, we propose an implementation for a Magnetohydrodynamics (MHD) scheme on the GPU, which we will use as a textbook example to illustrate the challenges of implementing a stencil-based algorithm on the GPU.

3.2.1 Description of the equations

The scheme aims to simulate the behavior of a magnetized fluid, which is a topic of high interest in various fields. There are several methods to achieve this, with different properties and computational costs. We consider the MHD equations with Divergence Cleaning, also referred to as the MHD-DC equations [86]:

$$\partial_t \begin{pmatrix} \rho \\ \rho u \\ Q \\ B \\ \psi \end{pmatrix} + \nabla \cdot \begin{pmatrix} \rho u \\ \rho u \otimes u + (p + \frac{B \cdot B}{2}) I - B \otimes B \\ (Q + p + \frac{B \cdot B}{2}) u - (B \cdot u) B \\ u \otimes B - B \otimes u + \psi I \\ c_h^2 B \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{3.2}$$

where I is the identity matrix, ρ is the density, u is the velocity, Q is the total energy, B is the magnetic field, and ψ is the divergence cleaning potential. The velocity and magnetic field vectors are represented as follows:

$$u = (u_1, u_2, u_3)^T, \quad B = (B_1, B_2, B_3)^T, \tag{3.3}$$

whereas the pressure is described by the perfect-gas law with a constant polytropic exponent $\gamma > 1$,

$$p = (\gamma - 1) \left(Q - \rho \frac{u \cdot u}{2} - \frac{B \cdot B}{2} \right). \tag{3.4}$$

Under the condition that the magnetic field satisfies the divergence-free requirement,

$$\nabla \cdot B = 0, \tag{3.5}$$

and the potential ψ remains constant, the MHD-DC equations simplify to the standard MHD equations. Hence, the MHD-DC equations serve as an extension of the MHD equations, allowing for a non-zero divergence in the magnetic field.

We can represent the state of the system as a vector of $m = 9$ conservative variables:

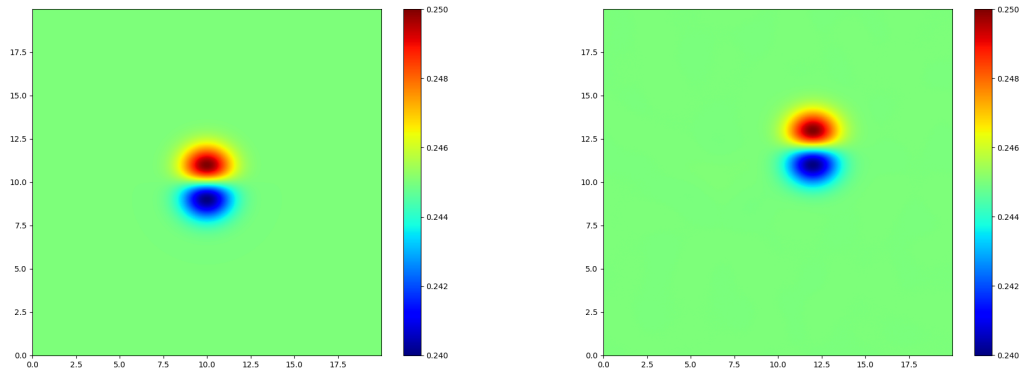
$$w = w(x,t) = \begin{pmatrix} \rho \\ \rho u \\ Q \\ B \\ \psi \end{pmatrix} \in \mathbb{R}^m. \tag{3.6}$$

The flux is given by

$$F(w,n) = \begin{pmatrix} \rho u \cdot n \\ \rho u \cdot nu + (p + \frac{B \cdot B}{2})n - B \cdot nB \\ (Q + p + \frac{B \cdot B}{2})u \cdot n - (B \cdot u)B \cdot n \\ u \cdot nB - B \cdot nu + \psi n \\ c_h^2 B \cdot n \end{pmatrix} \in \mathbb{R}^m, \tag{3.7}$$

where n is a vector of $\mathbb{R}^3$ of the velocity components. The flux is the quantity (of mass, momentum, energy, etc.) that is transported across a surface corresponding to the vector $n \in \mathbb{R}^3$.

These equations establish the foundation for the numerical scheme, applicable via a stencil-based approach within the LBM framework. Extensive discussions on the numerical method are available in several publications [86, 272, 24, 273, 19]. The implementation discussed herein draws upon the work by Baty et al. [29], which delivers an efficient GPU-based execution of the method, leveraging the LBM framework.



(a) Initial state. (b) State at $t = 10$.

Figure 3.2: Simulation of the MHD-DC equations performed on a 2048×2048 grid at $t = 0$ and $t = 10$. The plot shows the density field associated with the $(-1, 0)$ velocity component. The density within the vortex takes the form of a spiral, rolling in a clockwise direction. This structure is moving towards the top right corner at a constant speed.

The presented method offers multiple advantages. Firstly, there exist analytical solutions for the MHD equations, when the initial conditions are chosen appropriately. A frequently employed validation test is the Orszag-Tang vortex, a standard benchmark for MHD simulations [200, 77, 209]. A visual representation of this case is shown in Figure 3.2, where the density field is displayed at $t = 0$ and $t = 10$. On the other hand, a correct implementation should demonstrate a convergence rate of 2, which means that the error reduces by

a factor of 4 upon halving the grid size. This, combined with the fact that the analytical solution is known, provides a straightforward method for verifying the correctness of the numerical scheme.

In the subsequent section, we propose an LBM scheme for the MHD-DC equations, which we will implement on the GPU in Section 3.2.3.

3.2.2 Kinetic representation

To implement the above scheme, we use a kinetic representation of the conservative variables, following the LBM framework. Instead of relying on a grid of conservative variables, we use a grid of kinetic vectors, which represent the distribution of particles in the domain.

Let us define our velocity set $v_k = (v_k^1, v_k^2)^T$, $k = 1 \dots 4$, as:

$$v_1 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad v_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad v_3 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \quad v_4 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{3.8}$$

Each conservative variable w is related to the kinetic data by

$$w = \sum_{k=1}^{4} f_k. \tag{3.9}$$

The physical domain $\mathcal{D} =]0, L[\times]0, L[$ is discretized into a grid of $N \times N$ points, with a space step $\Delta x = L/N$. The domain is made periodic for the sake of simplicity, but boundary conditions such as Dirichlet or Neumann conditions can be implemented.

Since the velocities align with the mesh, we can from now on reason with the discretized grid only, where a cell at coordinates (i, j) corresponds to the physical space $[i\Delta x, (i+1)\Delta x] \times [j\Delta x, (j+1)\Delta x]$. We denote this discretized grid data as $f^n_{k,i,j}$, where i and j are the spatial indices, n is the time index, and $k = 1, \dots, 4$ is the kinetic velocity index. With this notation, each $f^n_{k,i,j}$ contains m values: the quantity of particles moving in the direction v_k in the cell (i, j) at time n. Hence, each spatial cell (i, j) contains $4m$ values: m for each kinetic direction. We also introduce $w^n_{i,j}$, which is the conservative state of the cell (i, j) at time n.

The goal of the numerical scheme is to provide the rules to compute f^{n+1} from f^n. Following the classical LBM approach, we rely on two steps: the transport step and the relaxation step. Let us first consider the transport step, which is a simple shift of the values:

$$\begin{cases} f^{n+1,\text{shift}}_{1,i,j} = f^n_{1,i+1,j}, f^{n+1,\text{shift}}_{2,i,j} = f^n_{2,i-1,j}, \\ f^{n+1,\text{shift}}_{3,i,j} = f^n_{3,i,j+1}, f^{n+1,\text{shift}}_{4,i,j} = f^n_{4,i,j-1}. \end{cases} \tag{3.10}$$

This step corresponds to the movement of particles along the lattice directions.

Then, the relaxation step handles the collision of particles. For this, the equilibrium function f^{eq}, which maps a conservative vector ($\mathbb{R}^m$) to a kinetic vector ($\mathbb{R}^{m\times 4}$) for each cell, is defined thanks to equation (3.7):

$$\sum_k f^{\text{eq}}_{k,i,j}(w^n_{i,j}) v_k \cdot n = F(w^n_{i,j}, n). \tag{3.11}$$

The conservative values $w^n_{i,j}$ are computed as in equation 3.9, but with the shifted values $f^{n,\text{shift}}$, i.e., the macroscopic state resulting from the transport step:

$$w^n_{i,j} = \sum_{k=1}^{4} f^{n,\text{shift}}_{k,i,j}. \tag{3.12}$$

Finally, the relaxation step combines the equilibrium and the transported values to compute the new approximation:

$$f^{n+1}_{k,i,j} = \omega f^{\text{eq}}_k(w^{n+1}_{i,j}) - (\omega - 1) f^{n+1,\text{shift}}_{k,i,j}, \tag{3.13}$$

where $1 \leq \omega \leq 2$ is a relaxation parameter. If $\omega = 1$, the scheme is very robust, entropy dissipative, but quite diffusive. If $\omega = 2$, the scheme is a second order in time approximation of the original equations, but unstable in shocks. In practice, we use an ω value close to 2, which is stable and achieves good accuracy.

In this section, we have described a LBM numerical scheme for the MHD equations with Divergence Cleaning, that approximates the solution of the MHD equations described in section 3.2.1. In the next section, we will show how this scheme can be implemented on the GPU.

3.2.3 Implementing the scheme on the GPU

To implement the MHD-DC scheme on the GPU, we aim to leverage the massive parallelism offered by the device. For simplicity, this discussion focuses on a 2D domain, reducing the conservative variables count to $m = 7$ by eliminating one dimension for both velocity and magnetic field vectors. Consequently, the scheme, adopting four distinct velocities, necessitates $q = 4 \times 7 = 28$ kinetic vectors, defining it as a D2Q28 scheme within LBM terminology.

The kernel includes the same two primary steps as the numerical scheme: the transport step and the relaxation step. In the GPU implementation, the transport step corresponds to reading the kinetic vectors from the neighboring cells, while the relaxation step includes the required computations to update the kinetic vectors and the final write.

The chosen layout for the lattice structure is a 3D array with dimensions `_NX` $\times$ `_NY` $\times$ 28. The array adheres to a row-major layout, prioritizing x as the fastest varying dimension, followed by y, and k as the slowest.

```
__global__
void time_step(const float *lattice_in, float *lattice_out) {
    for(int tid=blockIdx.x * blockDim.x + threadIdx.x; tid < _NX * _NY;
                                    tid+=gridDim.x * blockDim.x) {

        int x = tid % _NX;
        int y = tid / _NX;

        float fnow[28];
        // shift of values in domain
        get_flux_from_neighbors(fnow, lattice_in, x, y);

        // First order relaxation

        float wnow[7];
        f2w(fnow, wnow); // kinetic to conservative

        float fnext[28];
        w2f(wnow, fnext); // conservative to kinetic

        // second order relaxation
        const float omega = 2.f;
        for (int ik = 0; ik < 28; ik++)
            fnext[ik] = omega * fnext[ik] - (omega - 1) * fnow[ik];

        // Save the result to the output lattice
        for (int ik = 0; ik < 28; ik++)
            lattice_out[tid + ik * _NX * _NY] = fnext[ik];
    }
}
```

Code 3.6: CUDA pseudo-code for performing a time step of the MHD-DC scheme.

```
const int dir[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // The 4 velocities

__device__
void get_flux_from_neighbors(float *f, const float *lattice, float *f, int x, int y) {
    for (int d = 0; d < 4; d++) { // 4 velocities
        int x_neighbor = (x - dir[d][0] + _NX) % _NX;
        int y_neighbor = (y - dir[d][1] + _NY) % _NY;

        for (int iv = 0; iv < 7; iv++) { // 7 conservative variables
            int ik = d * 7 + iv;
            f[ik] = lattice[x_neighbor + y_neighbor * _NX + ik * _NX * _NY];
        }
    }
}
```

Code 3.7: CUDA pseudo-code for the function `get_flux_from_neighbors`.

The implementation of the MHD-DC scheme on the GPU is demonstrated in Codes 3.6 and 3.7. This approach ensures coalesced memory writes due to the indexing strategy, where `tid + ik * _NX * _NY` forms a contiguous block for different threads. This contiguity is achieved by appropriately setting `tid` to ensure threads are adjacent in memory.

Memory reads within the `get_flux_from_neighbors` function adopt a pattern as shown below:

```
lattice[x_neighbor + y_neighbor * _NX + ik * _NX * _NY].
```
(3.14)

Without considering the modulo operation, this equation simplifies to:

```
lattice[x + dir[d][0] + (y + dir[d][1]) * _NX + ik * _NX * _NY]
```
(3.15)

which further simplifies to:

```
lattice[tid + ik * _NX * _NY + dir[d][0] + dir[d][1] * _NX],
```
(3.16)

demonstrating a coalesced access pattern of `lattice[tid + constant]` for all threads within a given iteration, with `constant` being a constant. The modulo operation introduces uncoalesced accesses at the boundaries, but their impact on performance is minimal due to their infrequency.

The functions `w2f` and `f2w` are not depicted here, but they are central to the scheme. The `w2f` function computes f^{eq} from the conservative values w using equation (3.11), while the (`f2w`) function aggregates the kinetic vectors to conservative values (here, by summing across the four velocities as in equation 3.9).

For the sake of generality, we do not assume in-place computations, as some schemes cannot be implemented in-place. It can ben noted here, however, that this specific scheme could be implemented in-place, which would reduce the memory footprint. To do this, we can modify the global memory accesses so that the data are written to the same location they are read from. Then, the `get_flux_from_neighbors` function would need know the parity of the time step and read the fluxes either from the neighbors or from the cell itself. This in-place implemlentation would be referred to as an *AA-pattern* in the literature [23], while our naive implementation is known as a *pull* pattern [289]. The literature extensively refers to different access patterns that differ both in terms of execution time and memory requirements.

3.2.4 Evaluating the performance

To evaluate the performance of the MHD-DC scheme on the GPU, we measure the effective memory throughput on different GPUs.

GPU	P100	V100	A100
Maximum (theoretical)	732 GB/s	900 GB/s	1555 GB/s
Effective throughput	501 GB/s	769 GB/s	1270 GB/s
Efficiency	68%	85%	82%

Table 3.2: Observed effective memory bandwidth of the MHD-DC scheme on different GPUs. The efficiency is calculated as the ratio of the effective throughput to the theoretical maximum. The grid size is 2048×2048 on each configuration.

Table 3.2 shows the measured memory throughput and the theoretical maximum on different GPUs. The effective throughput is calculated as the number of bytes read and written divided by the execution time. We can see that the implementation is efficient, with a throughput relatively close to the theoretical maximum. Analyzing the performance in these terms is convincing, as it provides a clear upper bound on the performance of the kernel. Indeed, assuming that a global memory access is required for each read and write, it is impossible to achieve a higher throughput than the theoretical maximum. This methodological approach will be used thoroughly throughout this thesis to have an objective measurement of the performance of the different implementations.

This section has illustrated the implementation of a numerical scheme on the GPU, using the MHD-DC equations as an example. We have shown that, with this implementation, the GPU is nearly used at its full capacity, with a throughput close to the theoretical maximum. When implementing a numerical scheme on the GPU, the same thought processs is generally applicable, especially for schemes based on a regular grid. The limit of this framework is the size of the grid, as the memory capacity of the GPU puts a hard limit on the size of the grid that can be simulated.

In the case where the grid size exceeds the memory capacity of the GPU, a solution is to distribute the computation across multiple GPUs. However, distributing stencil computations introduces challenges. We will discuss these challenges throughout the rest of this chapter. In the following section, we explain the usual framework for distributing this type of computation across multiple GPUs.

3.3 Distributing the computation

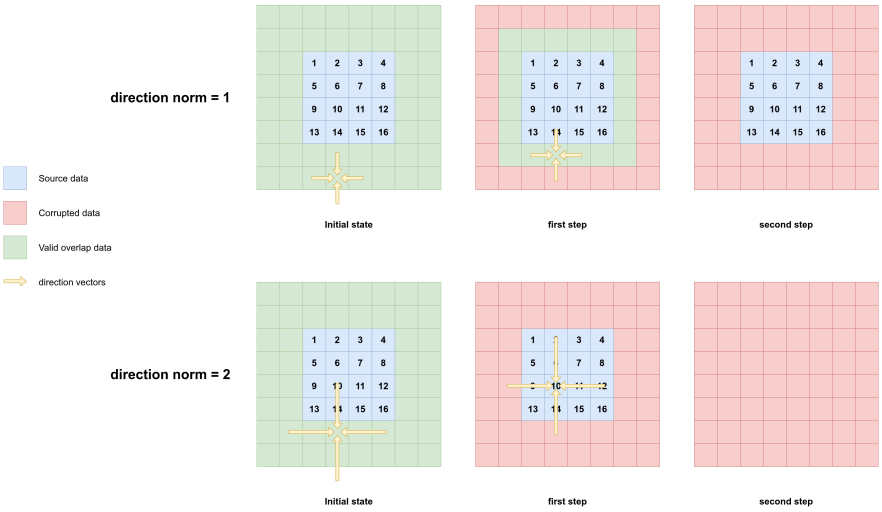


Figure 3.3: This example shows how bordering data are corrupted, depending on the direction vectors. The subgrid has an overlapping depth of 2 and a data size of 4×4. If the data are fetched (yellow arrows) from outside the subgrid or from a corrupted cell, the target cell becomes corrupted. When the norm of the direction vectors is 1, 2 steps can be performed without corrupting the main data, while a norm of 2 only allows for 1 step.

The typical approach to distributing stencil computations across multiple GPUs is to divide the domain into subgrids, each of which is assigned to a different GPU. Each subgrid contains extra data at its border that duplicate the values of its neighbors. These extra data are referred to as the *halo* of the subgrid. Some other names are used in the literature, such as *ghost cells*, *padding*, or *overlap*. The idea is to allow each subgrid to perform the stencil computation independently, without having to communicate with its neighbors. Figure 3.3, for instance, shows how the stencil can be performed independently from the neighbors, as long as the ghost cells are correctly updated. It shows two examples, one with a direction vector (stencil) of norm 1 and one with a direction vector (stencil) of norm 2. Both examples assume an overlapping depth of 2. We can see that depending on the shape of the stencil and the overlapping depth, the number of steps that can be performed without compromising the result varies. Throughout this work, we refer to different sizes for a subgrid. The whole subgrid size, which includes the ghost cells, is referred to as the *true* size. The size of the area denoted in blue on the figure is referred to as the *logical* size. We define the logical space as the space where useful computations are performed. As we can see on the figure, at the end of the process, only the blue area contains useful information that can be used for the next step.

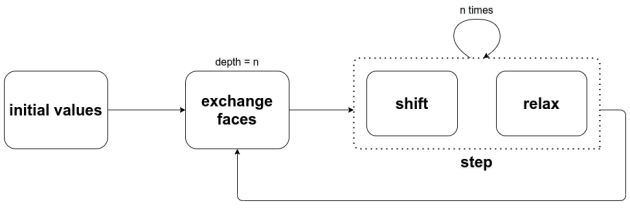


Figure 3.4: Typical execution flow of a stencil solver. It consists of a succession of *exchange* and *step* phases. The *step* phase is often divided into a *shift* (read the neighbors) and a *relax* (compute the new values) phase.

To ensure the coherency of the overall stencil loop, a synchronization phase is needed. We refer to this phase as the *exchange* phase or simply the synchronization depending on the context. In Figure 3.4, we show the basic execution flow of a distributed stencil computation. We first initialize the grid values on each subgrid, then perform successive alternating *exchange* and *step* phases. The important idea is that depending on the overlapping depth n, we can perform n steps after each synchronization. There are, therefore, perfomance implications in the choice of the overlapping depth.

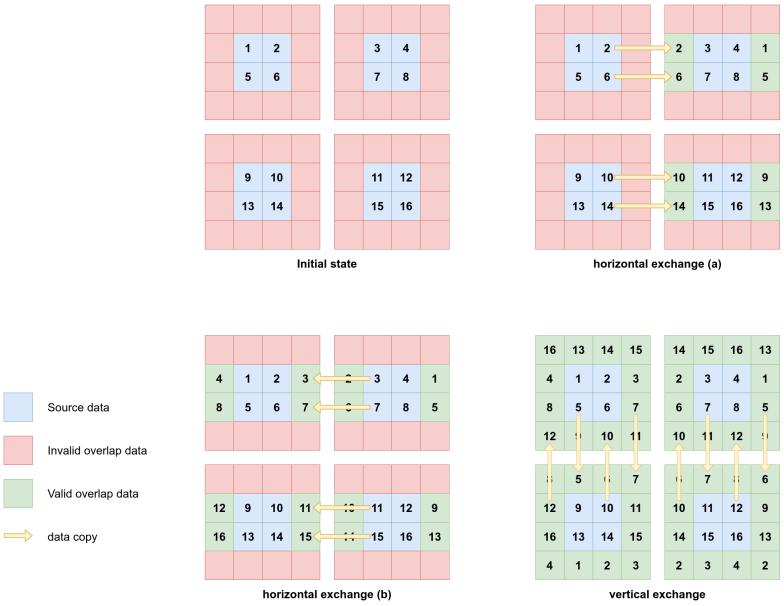


Figure 3.5: This schematic shows how the "faces" can be exchanged between the subgrid in the 2D case, with an overlapping depth of 1. The horizontal copy phase is divided into two parts to better display the process. For the sake of clarity, not all necessary data copies (yellow arrows) are shown. In a real case scenario, the data consist of vectors of floating-point values, rather than integers.

To be more specific, we provide an example for a synchronization between 2×2 subgrids in Figure 12.2. The schematic shows the exchange of "faces" between the subgrids. If we divide the synchronization into two successive phases (horizontal and vertical), we can achieve a full synchronization that includes the corners. The trick is to use a wider frame for the vertical synchronization, which includes the ghost "horizontal" cells. For example, the "7" value is first written to the corresponding ghost cell of the top-left subgrid during the horizontal exchange, then copied to the bottom-left subgrid, resulting in the "7" value being correctly written to the corner of the bottom-left subgrid.

Overall, several tweaks can be made to optimize the process flow of a stencil computation, but the concepts we presented here generally apply to most distributed stencil computations. For instance, the synchronization phase does not necessarily translate into a separated phase in the code. Moreover, it is generally achieve through the use of an interfacial buffer, which contains the data to be exchanged. In principle, only the interfacial buffers should be exchanged between the GPUs, to minimize the amount of data to be transferred. Section 4.5 provides a more detailed example of how interfacial buffers can be used to optimize the synchronization process.

So far, we have discussed the general framework for distributing stencil computations. When performance is a concern, as is the case in most scientific computing applications, new challenges arise and require careful consideration. In the next section, we discuss some of the most common challenges encountered when implementing distributed stencil computations on GPUs.

3.4 Challenges

Stencil algorithms are typically recognized for their high parallelization potential, stemming from their uniform data access patterns. The challenges encountered, however, are frequently similar across various implementations. This section outlines prevalent challenges in deploying stencil computations on GPUs, drawing insights from the stencil implementation detailed in Section 3.2. While the previous discussions have been anchored in a specific example, the challenges highlighted are broadly applicable to a wide array of stencil types.

Kernel Optimization The optimization of stencil kernels is an interesting, yet problem-specific challenge. It is generally accepted that the limiting factor of stencil computations is memory bandwidth. Hence, the discussion on optimization often revolves around making efficient data reuse and data access patterns. We are aware of multiple studies that focus on improving kernels for GPUs [82, 222, 247]. Although we did not make pivotal contributions to this field, we have used the insights from these studies to optimize our kernels.

Data Management Data management is a critical aspect of stencil computations, especially when considering distributed implementations. Numerous studies have been dedicated to optimizing distributed stencil computations [293, 302, 206, 122]. In general, the main focus is to achieve optimal balance between minimization of memory transfers and cost of synchronization. The most common techniques in this regard are temporal blocking and region sharing [124, 194, 188, 248]. Another issue that is closer to the concerns we raise in this thesis is the data management when the GPU memory is exceeded. Midorikawa *et al.* have proposed to use the SSD as a swap space for the GPU memory, make careful data-aware decisions to minimize the impact of the swap, and evaluate the performance of their method on a stencil computation [187]. However, the issue of strategically addressing system memory limitations, particularly when it exceeds capacity, often receives insufficient attention in the literature, likely under the assumption that leveraging additional hardware resources can circumvent the problem.

Coherence and Fault Tolerance In the realm of large-scale distributed computation, an increase in computation size and complexity often leads to a rise in the likelihood of system failures. The complexity of these systems introduces the potential for failures at various stages of computation, which can stem from hardware malfunctions, software bugs, or network disruptions, making fault tolerance a prevalent challenge in distributed computing. The consequences of a single failure can be catastrophic in distributed systems, leading to data corruption, loss of computational progress, and waste of large amounts of computational resources. Ensuring coherence within these systems necessitates conducting operations in the correct sequence to achieve the intended final results, which is paramount for the consistency of computations. However, the manual parallelization of tasks, for instance through the use of frameworks like MPI, not only increases the likelihood for implementation errors but also makes identifying and debugging these errors challenging.

To mitigate these issues, adopting robust software libraries specifically designed for fault tolerance in distributed environments is often recommended. Libraries such as FTI and SCR stand out as common choices for ensuring fault tolerance in large-scale distributed systems [191, 31]. These tools provide a foundational layer of resilience, enabling developers to concentrate on computational goals rather than the underlying complexities of fault tolerance and error recovery.

Consequently, the incorporation of such libraries into the development process of large-scale distributed systems is crucial. In the context of research, where the focus is usually on the development of novel algorithms and methodologies, the use of robust methodologies for fault tolerance is not necessarily a priority. However, it is important to note that the resilience of the system becomes critical in a production environment, where the loss of computational resources can have significant financial implications.

Heterogeneous Hardware Modern supercomputers are often composed of a mix of different hardware. This can include CPUs, GPUs, FPGAs, and other accelerators. The challenge is to make the most of the available hardware, which can be quite different in terms of performance and memory hierarchy. It is critical for some applications, where one type of processor is better for a part of the process, while another type is better for another part [6, 13, 170]. In other applications, where one type of processor is better for the whole process, it can still be relevant to perform heterogeneous computations to make the most of the available

resources. A common example for this is for processes that are faster on the GPU. As most supercomputers include computing node with both CPUs and GPUs, it would be a waste of resources to deliberately not use the CPUs.

Load Balancing Load balancing is a critical aspect of distributed computing and parallel processing. The issue stems from the fact that the computational load is rarely perfectly balanced across all processing units. This can be due to a variety of factors, that range from low-level hardware uncertainties to high-level choices. In the case of stencil computations, the load balancing issue poses a challenge, notably in the framework we described in Section 3.3.

In general, the strategy is to partition the domain into subgrids of equal size, with the assumption that the computational load is uniform across the domain [138]. However, several factors can lead to an imbalance in the computational load, such as local fluid dynamics, boundary conditions, and code optimizations [214]. This is why modern distributed stencil solvers often include a load balancing mechanism that redistributes the workload across the processing units to minimize idle time and maximize computational efficiency [28]. The challenge lies in developing a load balancing mechanism that is both efficient and scalable. For instance, it can be detrimental to transfer a subgrid from one GPU to another, as the cost of the transfer can outweigh the benefits of the load balancing.

3.5 Conclusions

In this chapter, we have presented the main concepts related to stencil computations, with a focus on the implementation of stencil computations on GPUs. We have shown how the GPU architecture can be leveraged to achieve high performance for stencil computations, and how the CUDA programming model can be used to express parallelism in a simple and straightforward way. We have also discussed the challenges of implementing stencil computations on GPUs, in particular the unique challenges of distributed stencil computations.

As we have seen, the literature on stencil computations is vast and varied, with many different approaches and optimizations. These optimizations are often problem-specific, and the choice of optimization depends on the specific problem at hand. While our work does not focus on the optimization aspect of stencil computations, it is important to be aware of the underlying principles to provide relevant contributions to the field. As we have seen throughout this chapter, the challenges associated with distributed stencils are typically solved thanks to the use of an external library, such as MPI or a custom library. The HPC community values runtime execution engines for their role in offering an interface that enables efficient parallel execution within the realm of high-performance computing. In the next chapters, we present the advantages of using such a runtime system for the implementation of distributed stencil. First, in Chapter 4, we present our contributions to the field of efficient stencil computations with the PaRSEC runtime system. Then, in Chapter 5, we present a parallelization approach with the StarPU runtime system.

Chapter 4

High-performance parallelized stencils with PaRSEC

When performance is critical, the classical approach to distributing stencil computations is to rely on a dedicated engine. Among the most popular ones, we can cite OpenLB [137], pyLBM [105], Palabos [152], LB3D [239], StencilFlow [85], and many others. These frameworks are designed to simplify the development of CFD simulations, while providing high performance. In general, they work by performing a domain decomposition, where the global grid is divided into subgrids, which are then distributed to the available processing units (similar to what we describe in Section 3.3). However, to achieve the highest performance in distributed environments, it is often necessary to use a more flexible approach, which can allow for better load balancing, better data locality, etc. In this work, we focus on the development of a high-performance CFD simulations using task-based runtime systems. This chapter focuses on the PaRSEC [46] runtime system, while the next chapter focuses on the StarPU [20] runtime system. Both PaRSEC and StarPU are considered state-of-the-art in the field of HPC, and are used in many research projects and production environments. They rely on different programming models (but are both Task-Based), and have different features and capabilities.

This chapter provides a summary of the task-based approach in Section 4.1, highlighting the benefits of using runtime systems to address the challenges of distributed computing. We then delve into the specifics of the PaRSEC runtime system in Section 4.2, highlighting its key features and capabilities. Subsequent sections build on this foundation, with Section 4.3 explaining the PTG programming model, central to PaRSEC. Our enhancements to this model, designed to optimize stencil computations, are detailed in Section 4.4. Our implementation of a D2Q9 stencil with PaRSEC is outlined in Section 4.5, where we also discuss leveraging the PTG model to achieve high performance. The chapter concludes with a performance evaluation of our implementation on a single A100 GPU in Section 4.6 and discusses the results in Section 4.7.

4.1 Task-Based Runtime Systems

The increasing complexity of modern HPC systems has made the development of efficient applications more challenging. Multiple challenge can arise when aiming to efficiently use the available resources, such as ensuring coherency of the data, managing the memory transfers, handling heterogeneous processing units, etc [131, 189]. Addressing these challenges inevitably leads to a higher complexity in the code, which can make the code harder to maintain and to understand [280, 281]. This is why the HPC community has invested much effort into designing programming models and runtime systems.

One modern way of addressing the problem is to use runtime systems. These frameworks aim to provide a high-level interface to the user, abstracting some of the complexities inherent to distributed computing [267]. They usually rely on a programming model that describes the program in a way that is automatically processable by the runtime system. A powerful class of programming models is the task-based programming model, which allows the user to describe the program as a set of tasks, which are then scheduled by the runtime system. The common denominator of these frameworks is that the basic unit of work is a task. A

task can be seen as a part of the program that can be executed on a processing unit, which can be a CPU core, the whole CPU, a GPU, etc. The runtime system is generally responsible for scheduling the tasks on the available processing units, managing the data transfers, ensuring the coherency of the data, etc.

There are several reasons why using a runtime system can be beneficial. One reason is performance, which is the main reason for our work. The performance gains can be achieved thanks to efficient scheduling from the runtime system, low-level optimizations embedded in the runtime system (e.g., overlapping computation and communication), the ablity to perform heterogeneous computing, or simply the ability to parallelize a process that would be hard to parallelize otherwise. Other reasons include the ability to handle dynamic workloads, the ability to handle failures, or the resilience to changes in the configuration of the system.

The use of task-based parallelization has already been studied in the context of stencil computations [22, 205, 185]. The work of Lima *et al.* [165] appears to be the closest to our work. They provide a comparison of different implementations for a D3Q19 simulation and conclude that the task-based (StarPU) implementation is the fastest. This highlights the potential of the task-based approach, even if their work is limited to a single node. The works of Raut *et al.* [220, 219] are also relevant for our work. Their approach, based on Legion [30] (a task-based runtime system), demonstrated that the task-based approach can compete with the traditional MPI+OpenMP approach. They notably extended their work to multi-GPU systems [218]. Although they assuredly provided a highly scalable distributed implementation, it is not clear whether their approach is easily extendable to schemes where a vectorial kinetic representation is required, as in the LBM. Moreover, we were not able to find any public implementation of their work, which makes it hard to analyze their approach in detail. However, the Legion runtime system appeared to be a good candidate for our work, but we did not further investigate its capabilities due to a lack of time.

In the following sections, we will present the PaRSEC runtime system and explain how its paradigm can be leveraged to implement multi-GPU Lattice-Boltzmann simulations.

4.2 Overview of PaRSEC

PaRSEC is a task-based runtime system that has been created to handle the challenges of distributed hetetogeneous computing. It provides DSLs such as the Parameterized Task Graphs (PTG) model [79] and the Dynamic Task Discovery (DTD) model [117]. These models allow the user to describe the program with a high level of abstraction. PaRSEC has been used in various research projects and has demonstrated high potential in various fields [44, 45, 3].

The development of the PaRSEC runtime system was initiated to address the challenges of efficiently managing memory and ensuring data coherence across a diverse range of processing units. Addressing these challenges, PaRSEC introduces a programming model centered around task flows.

Within this model, the PTG model enables users to describe programs algebraically as sets of tasks along with their dependencies (more on this in Section 4.3). This approach stands in contrast to traditional parallel programming models, which often depend on a fixed (explicit) task graph or adopt a fully dynamic methodology. The PTG model, on the other hand, allows for the deduction of the entire graph without the need for explicit synchronization by every processing unit at any point during execution. This is achieved thanks to compile-time analysis of the algebraic description of the program, which allows the runtime system to make informed decisions about task execution order and resource allocation. One primary limitation of this method is the static nature of the (virtual) graph, which is fully determined at the start of execution.

DTD was introduced in part to address this limitation [117]. This secondary DSL adopts a more dynamic approach, allowing for the dynamic discovery of tasks as computations evolve. DTD is especially beneficial for applications demanding high adaptability. For the applications considered in this work, the PTG model is sufficient, as the task sets are known in advance.

PaRSEC offers several advantages:

- High scalability is offered through its capability to manage a large volume of tasks;
- Low synchronization overhead features prominently, minimizing delays and improving overall efficiency, especially in distributed computing environments where communication costs can be significant;
- Efficient task scheduling is provided by leveraging expressive dependency descriptions, enabling the runtime system to make informed decisions about task execution order and resource allocation;

- High adaptability to various hardware architectures is demonstrated, allowing seamless operation across multicore CPUs, GPUs, and other specialized processors, thereby maximizing performance and resource utilization.

Nevertheless, PaRSEC also has limitations. The complexity involved in expressing algorithms via PTG may pose a steeper learning curve for developers than higher-level programming models. The efficiency of an execution is highly dependent on the task flow description, the quality of which heavily depends on user expertise. Moreover, the static nature of the PTG model can make it challenging to express applications with dynamic task sets. Even if DTD partly addresses these limitations, it still requires a high level of expertise to use effectively, making PaRSEC difficult to use for users outside the HPC community.

Subsequent sections will detail our contributions to the PaRSEC project, focusing on facilitating stencil computations on multi-GPU systems. The forthcoming section will elaborate on the PTG programming model. Section 4.4 will discuss our extensions to the PTG model to more aptly express stencil computations. Sections 4.5 and 4.6 will show a use case involving the implementation of a D2Q9 stencil with PaRSEC and its performance evaluation on a single A100 GPU.

4.3 PTG Programming Model

The PTG model is a high-level programming model that allows the user to describe the program as a set of tasks and their dependencies. The tasks must be divided into task classes, which are the basic unit of description in the PTG model. Then, the user can describe the *in* and *out* data accesses of each task in JDF language, which is the DSL that implements the PTG model. Each of these accesses are described by a *dataflow*, which is a set of rules that describe how the data is accessed. More specifically, the rules must describe where to fetch the data (*in* dependencies) and where to "send" the data (*out* dependencies).

Let us consider a simple example with a *GEMM* (General Matrix Multiply) algorithm. The inputs of the algorithm are two matrices A and B, and the output is the matrix C. To allow distributed computing, we divide the matrices into tiles and refer to them as $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$. To define a *GEMM* task class, we need to provide rules for computing a tile of C.

```
GEMM(m, n) // Name of the task class and its parameters

// Ranges of m and n (tile coordinates)
m = 0 .. matrix_size_m-1
n = 0 .. matrix_size_n-1

// Read data are described with "<-" and write data with "->"
READ dataflow_A <- A(m, ...) // The line (of tiles) m of A
READ dataflow_B <- B(..., n) // The column (of tiles) n of B
WRITE dataflow_C -> C(m, n)

// Kernel for computing C(m, n)
BODY
    float *C_mn = dataflow_C; // The tile C(m, n)
    memset(C_mn, 0, tile_size*tile_size*sizeof(float));

    for(int k = 0; k < matrix_size_k; k++) {
        // Amk and Bkn are not retrievable in this example
        float *A_mk = ...;
        float *B_kn = ...;
        for(int i = 0; i < tile_size; i++) {
            for(int j = 0; j < tile_size; j++) {
                C_mn[i, j] += A_mk[i, k] * B_kn[k, j]
            }
        }
   }
END
```

Code 4.1: Pseudo-(non working) code for the GEMM task class.

The pseudo-code provided in Code 4.1 recalls the basic structure of a GEMM algorithm. It is not valid JDF code, because it does not give the exact order k in which the $A_{i,k}$ and $B_{k,j}$ tiles must be computed. One way to express this is to introduce an iterator k that gives the exact order of the computation.

```
GEMM(m, n, k) // Name of the task class and its parameters

```

```
// Ranges of m and n (tile coordinates)
m = 0 .. matrix_size_m-1
n = 0 .. matrix_size_n-1
k = 0 .. matrix_size_k-1

READ dataflow_A <- A(m, k)
READ dataflow_B <- B(k, n)
// Now, dataflow_C is read and written
RW dataflow_C
    // If first iteration, allocate a new piece of memory, otherwise, use the existing one
    <- (k==0) ? new : dataflow_C GEMM(m, n, k-1)
    // If last iteration, write the result to C, otherwise, sent to the next iteration
    -> (k<matrix_size_k-1) ? dataflow_C GEMM(m, n, k+1) : C(m, n)

// Kernel for updating C(m, n)
BODY
    float *current_C = dataflow_C; // Current tile C(m, n)
    float *A_mk = dataflow_A; // Current tile A(m, k)
    float *B_kn = dataflow_B; // Current tile B(k, n)

    for(int i = 0; i < tile_size; i++) {
        for(int j = 0; j < tile_size; j++) {
            if(k == 0)
                current_C[i, j] = 0;
            current_C[i, j] += A_mk[i, k] * B_kn[k, j]
        }
    }
END
```

Code 4.2: Example JDF code for the GEMM task class.

The updated Code 4.2 is near-valid JDF code. The main difference is that the dataflows have been rewritten so that they correspond to a piece of data (tile). Due to the introduction of the iterator k, the piece of data that will contain the result is now processed $matrix\_size\_k$ times.

```
// Let us focus on the dataflow_C:
RW dataflow_C
    <- (k==0) ? new : dataflow_C GEMM(m, n, k-1)
    -> (k<matrix_size_k-1) ? dataflow_C GEMM(m, n, k+1) : C(m, n)

// Each task class has access to a dataflow_C buffer
// The data are processed in by the runtime system
// The following pseudo-code shows how the JDF code can be translated into C code

void pdeuso_code_get_dataflow_C(void *dataflow_C, int m, int n, int k) {
    // Conditional block deduced from the JDF code
    if(k == 0) {
        // Allocate a new piece of memory
        dataflow_C = new_dataflow_C();
    } else {
        // Get the data from dataflow_C GEMM(m, n, k-1)
        dataflow_C = get_dataflow_C(m, n, k-1);
    }
}

void pseudo_code_send_dataflow_C(void *dataflow_C, int m, int n, int k) {
    // Conditional block deduced from the JDF code
    if(k < matrix_size_k-1) {
        // Send the data to dataflow_C GEMM(m, n, k+1)
        send_dataflow_C(dataflow_C, m, n, k+1);
    } else {
        // Write the result to C(m, n)
        void *C_m_n = get_C(m, n);
        memcpy(C_m_n, dataflow_C, tile_size*tile_size*sizeof(float));
    }
}
```

Code 4.3: Pseudo-code for translating the JDF code for dataflow_C into C code.

The new *dataflow_C* dataflow is more complex and provides insights into how the dataflows are designed in PTG. For instance, Code 4.3 shows pseudo-codes that describe how the JDF code for *dataflow_C* can be translated into C code. Basically, a dataflow must provide at least one of two rules: one for reading and one for writing. Here, `pdeuso_code_get_dataflow_C` shows how the rule for reading the data can be implemented, while `pseudo_code_send_dataflow_C` shows does the same for writing the data. It is possible to have only one of these rules, for instance, if the data is only read or only written. It is important to note

that *dataflows* correspond to the description of the in/out data accesses of a task class and do not represent the actual data, which is handled by the runtime system.

An *in* dependency to a dataflow should always be met with an *out* dependency from the same dataflow. In this case, the dataflow is read and written by the same dataflow (`dataflow_C`). We can see the exact correspondence between the *out* and the *in* rules: `dataflow_C GEMM(m, n, k+1)` and `dataflow_C GEMM(m, n, k-1)`.

The high expressiveness of the PTG model comes with a cost. Here, the introduction of the iterator k added additional constraints to the order of execution and, hence, added avoidable barriers to parallelism. One idea to avoid this is to use the concept of parametrized flows, which is a way to express the dataflows in a more abstract way, as demonstrated in Code 4.4.

```
READ dataflow_A [k = 0 .. matrix_size_k-1] <- A(m, k)
READ dataflow_B [k = 0 .. matrix_size_k-1] <- B(k, n)
```

Code 4.4: Pseudo-code for the A and B dataflows using the idea of parametrized flows.

In the case of GEMM, this approach would not be efficient, because it would require to store an entire line of A and an entire column of B per tile of C, which does not favor data locality. However, we can see that the concept of parametrized flows is a powerful tool to express the dataflows in a more abstract way, and it is especially useful for stencil computations, as we will see in the next section.

4.4 Extending the PTG model for stencil computations

The concept of parametrized flows was developed in response to the intricate dependency patterns commonly observed in stencil computations. Challenges in these computations largely arise from the need for subgrid synchronization, a crucial aspect of the algorithmic framework. Difficulties encountered encompass the requirement for stencil-specific optimizations, the impact of temporal blocking on dependency management, and the pursuit of a formulation that enhances the capacity for generalization. These challenges are showcased in Section 4.5.

Our first enhancement in PaRSEC involves the development of a data structure designed to elegantly represent multi-dimensional data. Prior to this enhancement, PaRSEC primarily utilized data structures suited to matrix-based applications typical of linear algebra, which do not adequately generalize to multi-dimensional contexts. In PaRSEC, the abstraction and management of any data type are handled with the `parsec_data_collection` structure, which encapsulates the data. A `parsec_data_collection` represents a collection of data segments (such as matrix tiles) and a set of access rules. The newly introduced data structure, `parsec_multidimensional_grid`, supports an arbitrary number of dimensions, thereby offering the flexibility and expressiveness required for advanced stencil implementations.

The integration of parametrized flows marks another crucial advancement. To illustrate the challenges posed by the traditional PTG model in representing stencil computations, consider the example of a simple 2D stencil.

```
for(int i = 1; i < width-1; i++) {
    for(int j = 1; j < height-1; j++) {
        float sum = 0;
        for(int k = -1; k <= 1; k++) {
            for(int l = -1; l <= 1; l++) {
                sum += input[i+k][j+l] * gaussian_kernel[k+1][l+1];
            }
        }
        output[i][j] = sum;
    }
}
```

Code 4.5: Pseudo-code for a gaussian blur 2D stencil without parallelization.

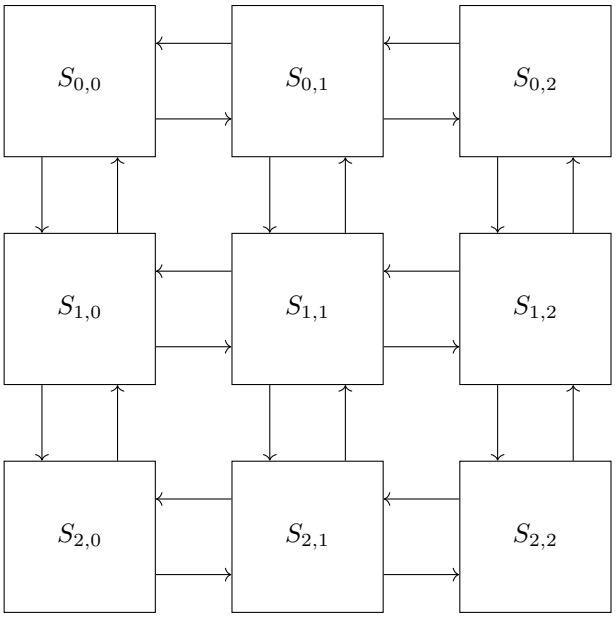


Figure 4.1: Dependency structure of a gaussian blur 2D stencil with 3x3 subgrids.

To parallelize Code 4.5, we divide the grid into $3 \times 3 = 9$ equally-sized subgrids and no overlapping. Figure 4.1 shows the dependency structure of the algorithm under this configuration. A way of expressing this in PTG is to create a task class for each direction.

```
GAUSSIAN_BLUR(x, y) // Name of the task class and its parameters

// Ranges of x and y (tile coordinates)
x = 0 .. width-1 // 0 .. 2 for 3x3 subgrids
y = 0 .. height-1 // 0 .. 2 for 3x3 subgrids

// Rules for getting the neighboring subgrids
READ LEFT_SUBGRID <- (x>0) ? input(x-1, y) : NULL
READ RIGHT_SUBGRID <- (x<width-1) ? input(x+1, y) : NULL
READ UP_SUBGRID <- (y>0) ? input(x, y-1) : NULL
READ DOWN_SUBGRID <- (y<height-1) ? input(x, y+1) : NULL

// Main in/out dataflows
READ OWN_SUBGRID_IN <- input(x, y)
WRITE OWN_SUBGRID_OUT -> output(x, y)
```

Code 4.6: Pseudo-JDF code for the gaussian blur 2D stencil.

Code 4.6, presenting near-valid JDF syntax, illustrates the foundational JDF structure for implementing a Gaussian blur 2D stencil. The issue under consideration is the rule redundancy for acquiring neighboring subgrids. While manageable for a system with four dataflows, this redundancy escalates in higher-dimensional scenarios, potentially leading to errors. Moreover, such redundancy could degrade the scheduling efficiency due to the less discernible dependency pattern. Parametrized flows, as demonstrated in Code 4.7, offer a more succinct expression of the dependency pattern.

```
const int dx[4] = {-1, 1, 0, 0};
const int dy[4] = {0, 0, -1, 1};

// [...]

READ NEIGHBOR_SUBGRID [d = 0 .. 3] <- input(x+dx[d], y+dy[d])
```

Code 4.7: Pseudo-JDF code for the gaussian blur 2D stencil with parametrized flows.

To effectively integrate this code within PaRSEC, alterations to accommodate parametrized flows are imperative. Two methodologies for managing parametrized flows are proposed. Initially, one approach entails analyzing and expanding the parametrized flows at compile time, thus generating a comprehensive set of rules, as depicted in Code 4.8.

```
// Expanded flows
READ NEIGHBOR_SUBGRID_0 <- input(x-dx[0], y-dy[0])
READ NEIGHBOR_SUBGRID_1 <- input(x+dx[1], y+dy[1])
READ NEIGHBOR_SUBGRID_2 <- input(x-dx[2], y-dy[2])
READ NEIGHBOR_SUBGRID_3 <- input(x+dx[3], y+dy[3])

// [...]

// If we want to avoid boilerplate code, we can use a macro:
#define NEIGHBOR_SUBGRID(d) NEIGHBOR_SUBGRID_##d
```

Code 4.8: Pseudo-JDF code for the gaussian blur 2D stencil with expanded parametrized flows.

The primary challenge with the static approach to handling parametrized flows is that it presupposes known bounds at compile time to deduce the number of rules. In contrast, a dynamic approach allows for the expansion of parametrized flows during execution, leveraging runtime information to unfold the parametrized flows. The static method is simpler to implement and potentially more efficient but suffers from a lack of adaptability. On the other hand, the dynamic method offers greater flexibility at the cost of potential efficiency losses and increased complexity in implementation. Implementing the static model requires merely an initial rewriting pass within the PTG compiler, whereas adopting a dynamic model necessitates comprehensive changes to internal operations within PaRSEC.

This research explores an intermediate strategy that expands parametrized flows at execution start. Unlike the static approach, this method performs expansion at runtime, leveraging execution-time information to unfold the parametrized flows. This adjustment necessitates modifications only to the PTG compiler, which is a JFD to C compiler, avoiding extensive alterations to the runtime system itself. The primary challenge lies in ensuring that the generated C code appropriately adjusts the necessary structures at execution start, maintaining overall execution coherence. This introduces substantial technical challenges on which we will not elaborate further in this document.

The outcome of this research is an enhanced version of the PTG compiler capable of managing parametrized flows effectively. In the subsequent section, the application of this new compiler feature to develop a high-performance LBM D2Q9 solver will be discussed.

4.5 Implementation of a D2Q9 stencil with parametrized flows

To assess the potential of the PTG model to express stencil computations, let us consider the implementation of a D2Q9 stencil. We choose a numerical scheme whose dependency structure can be leveraged to achieve efficient subgrid synchronization.

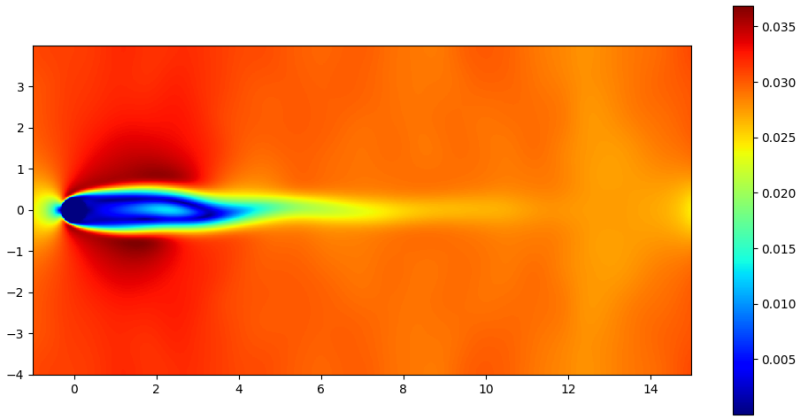
4.5.1 Description of the D2Q9 stencil

The conservative variables of this LBM scheme are the density and the density-weighted velocity: $W = (\rho, \rho u_x, \rho u_y)$. The used velocity set is $(0,0)$, $(\pm 1, 0)$, $(\pm 1, 0)$, and $(\pm 1, \pm 1)$ [38], with corresponding weights: $\frac{4}{9}$ (for the center), $\frac{1}{9}$ (for the 4 "faces"), and $\frac{1}{36}$ (for the 4 "corners") [309]. The rest of the construction is similar or can be extrapolated to 2D from the D3Q27 scheme we describe later in Section 9.3.1 but in two dimensions. In this section, we focus on the dependency structure of the D2Q9 stencil and assume that the numerical process is the result of the application of a stencil on the grid, as described in Code 4.9.

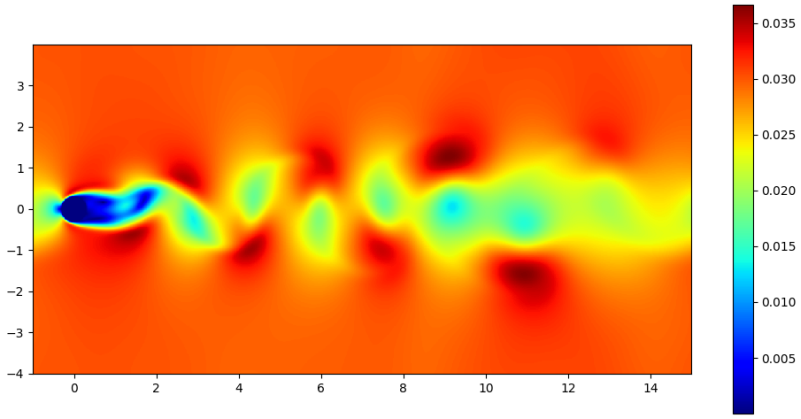
```
const int dir[9][2] = {{0, 0}, {1, 0}, {0, 1}, {-1, 0}, {0, -1}, {1, 1}, {-1, 1}, {-1, -1}, {1,
    -1}};
const float weights[9] = {4.f/9, 1.f/9, 1.f/9, 1.f/9, 1.f/9, 1.f/36, 1.f/36, 1.f/36, 1.f/36};

// Iterate over the physical grid, ignore the borders
for(int i = 1; i < width-1; i++) {
    for(int j = 1; j < height-1; j++) {
        float f[9];

        // Fetch the 9 neighboring cells
        for(int d=0; d<9; d++) {
            int dx = dir[d][0];
            int dy = dir[d][1];
            f[d] = input[i+dx][j+dy][d];
        }
```

```
15
16          phy(f, weights); // Apply the stencil
17
18          // Save the result
19          for(int d=0; d<9; d++)
20              output[i][j][d] = f[d];
21      }
22  }
```

Code 4.9: Pseudo-code for the D2Q9 stencil.



(a) Simulation at $t = 200s$.



(b) Simulation at $t = 400s$.

Figure 4.2: Velocity field of the D2Q9 stencil at $t = 200s$ and $t = 400s$.

Figure 4.2 shows the velocity field of the D2Q9 stencil at $t = 200s$ and $t = 400s$. The sphere blocks the fluid and highly periodic vortices, known as von Kármán vortices, appear after a certain time. The specifics of this simulation are not relevant in this chapter. However, we ensured that the results remain physically plausible (no obvious numerical instability) and that the simulation is stable over time.

As we have explained in Section 3.3, the standard method for parallelizing this kind of stencil is to divide the grid into subgrids and to use interface buffers to synchronize the subgrids. Between each time step, the

edge values of the subgrids are written to the interface buffers. These buffers are then sent to the target location and are used for updating the ghost cells of the neighboring subgrids, hence duplicating the edge values of the neighbors. Using intermediate buffers helps avoiding data transfers, as their size is substantially smaller than the size of the subgrids. In the case of the D2Q9 stencil, it is possible to reduce the size of the interface buffers by leveraging the shape of the stencil.

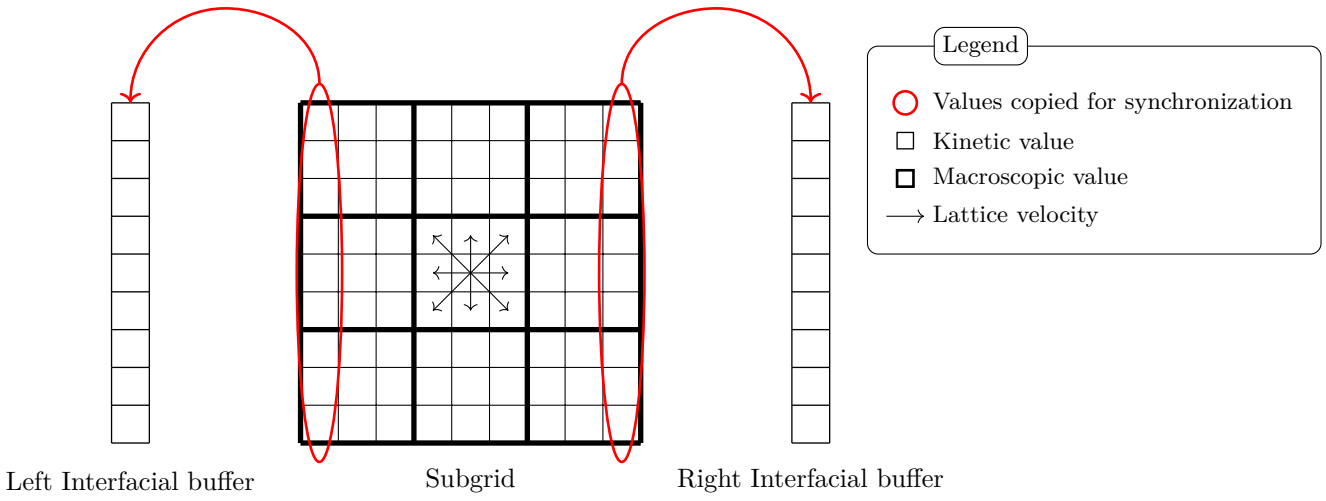


Figure 4.3: Efficient writing to the interface buffers for the D2Q9 stencil on the x-axis for a single time step. Only the outgoing fluid is written to the interface buffers. the same principle applies to the y-axis.

The size of the interface buffers can be reduced by a factor of three by using only the relevant directions, as shown in Figure 4.3. Since the numerical only requires the outgoing fluid of the neighboring cells, only the values whose direction is towards the exterior of the subgrid are written to the interface buffers. Without knowing the lattice structure, all the edge kinetic values would have had to be written to the interface buffers.

While this technique minimizes the size of the interface buffers, it can also be inefficient. In a distributed environment, it can be relevant to perform less communication but more computation. In iterative stencils, it is possible to duplicate computations to achieve better performance with techniques related to temporal blocking. In our framework, this can translate to increasing the depth of the ghost cell area (overlap) and, hence, the size of interface buffers. With this technique, the synchronization can be done less frequently, and the computation can be done more efficiently.

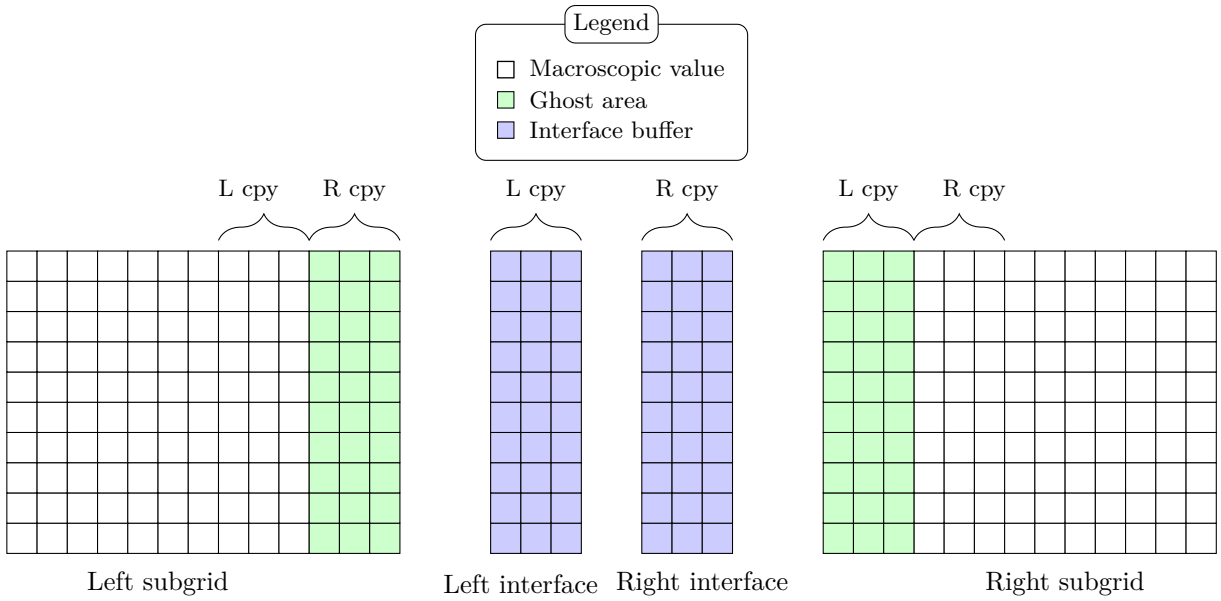


Figure 4.4: Synchronization process for the D2Q9 stencil using a depth of 3 in the ghost cell area. The brackets indicate the regions that should be equal after the synchronization. Namely, *L copy* represents the left-to-right copy, *R copy* the right-to-left copy. The ghost area is only shown for the relevant direction for the sake of clarity, but there are ghost cells in all directions.

Figure 4.4 shows how this principle can be applied to the D2Q9 stencil. The main difference with the previous figure is that all the kinetic values of the neighboring cells are written to the interface buffers, rather than only the outgoing fluid. This is because the cells in the ghost area can now influence each other in all directions. With this type of synchronization, a depth of d allows to perform d iterations of the stencil without synchronization.

We, hence, have two presumably efficient ways of synchronizing the subgrids: the first one is to use the shape of the stencil to reduce the size of the interface buffers, and the second one is to increase the depth of the ghost cell area to reduce the number of synchronizations. In this implementation we adopt a different method depending on the axis. On the y-axis, we use the first method, and on the x-axis, we use the second method. This choice aims to maximize coalesced memory accesses on the GPU. The first method is more efficient for the y-axis because the kinetic data are contiguous in memory for the whole line. The second method allows to achieve better memory efficiency on the GPU. For instance, with a depth of 32, the memory accesses can be grouped by 32, which aligns perfectly with the memory access pattern of the GPU.

This choice introduce a complex synchronization pattern, as the y-axis requires a synchronization every time step, while the x-axis requires a synchronization every 32 time steps. The exchange of the corner values is also not trivial to implement, as the corner values are not easily accessible. In the next section, we will see how this synchronization pattern can be implemented in PaRSEC, leveraging parametrized flows.

4.5.2 Implementation with parametrized flows

To conciliate the y-axis synchronization and the PaRSEC data collection mechanism, we group the kinetic values by their y offset (with regards to the stencil). The kinetic values that go upwards will be grouped together, and so will the kinetic values that go downwards/horizontally. Hence, each subgrid is acutally composed of 3 pieces of data (at the level of the data collection) and we refer to them with the *dy* iterator, that ranges from -1 to 1, or sometimes from 0 to 2 for the sake of clarity.

It is logical to have a corresponding task class for each part of the work. We, hence, have the *fill_grid* task class, which sets the initial condition of the scheme, the *LBM_step* task class, which applies the stencil (a time step), and the *write_horizontal_slices*/*read_horizontal_slices* task classes, which perform the subgrid-to-interface and interface-to-subgrid copies (with the method shown in Figure 4.4). For technical reasons

that we do not detail here, there is a GPU kernel *read_vertical_slices* that does not have a corresponding task class. The vertical synchronization is done directly in the *LBM_step* task class, as it is necessary at each time step. To achieve high performance in distributed environments, we need to ensure that the intermediate buffers can be sent alone to the target location. For this, we interleave task classes for the horizontal and vertical interfacial buffers, which do not perform any computation but allow PaRSEC to treat their dataflows as single pieces of data. This would not work as intented with only the *LBM_step* task class because the engine would transfer all the data required by the neighboring *LBM_step* task classes, including the large kinetic grids. The interleaved task classes are called *vertical_interface* and *horizontal_interface* and represent the state of the buffer at a given synchronization. We also implement two task classes for saving the results to disk, *save_file* and *save_file_reduce*. The *save_file* task class is a CPU task class that writes the data to disk, while the *save_file_reduce* task class is a GPU task class whose purpose is to iterate over the subgrids and write the data to a single (reduced) piece of data.

```
LBM_step(subgrid_x, subgrid_y, step)

subgrid_x = 0 .. subgrid_number_x-1
subgrid_y = 0 .. subgrid_number_y-1
step = 0 .. number_of_steps-1

RW INTERFACE_DOWN
    // Takes the data from the vertical interface at the (step-1)'th synchronization (if exists)
    <- (step!=0)
        ? VERTICAL_INTERFACE vertical_interface(subgrid_x, subgrid_y, 0, step-1)
        : NULL
    // Writes the data to the vertical interface at the step'th synchronization
    -> (step!=number_of_steps-1)
        ? VERTICAL_INTERFACE vertical_interface(subgrid_x, subgrid_y, 1, step)
        : NULL

RW INTERFACE_UP
    // Similar to INTERFACE_DOWN, but targets (subgrid_y+1)%subgrid_number_y instead of subgrid_y

// For each dy offset
READ SUBGRID_FROM[dy = 0..2]
    // If there was a save at the previous step, read the reduced data
    // Note: the first set (initial condition) is always saved
    <- (step%save_interval == 0)
        ? SUBGRID_SAVE_REDUCE save_file_reduce(subgrid_x, subgrid_y, d, step/save_interval)
    // If there was a horizontal sync at the previous step, read the data after the sync
    // Note: we synchronize every overlap_x steps
    <- (step%overlap_x==0)
        ? SUBGRID_HORIZONTAL_WRITE[dy] write_horizontal_slices(subgrid_x, subgrid_y,
                                                               step/overlap_x-1)
    // Else, read from the previous step
    <- SUBGRID_TO[dy] LBM_step(subgrid_x, subgrid_y, step-1)

WRITE SUBGRID_TO[dy = 0..2]
    // Mostly similar to SUBGRID_FROM, not shown for the sake of brevity

BODY [type=CUDA]
    float *subgrid_FROM_DY[3];
    float *subgrid_TO_DY[3];
    for(int dy=0;dy<3;++dy)
    {
        // The parametrized flows SUBGRID_FROM and SUBGRID_TO can be accessed like arrays
        subgrid_FROM_DY[dy] = SUBGRID_FROM[dy];
        subgrid_TO_DY[dy] = SUBGRID_TO[dy];
    }

    LBM_step_call(subgrid_x, subgrid_y,
                  subgrid_FROM_DY, subgrid_TO_DY,
                  INTERFACE_UP, INTERFACE_DOWN,
                  ...); // Call the actual LBM step kernel
END
```

Code 4.10: Pseudo-JDF code with parametrized flows for the LBM_step task class.

Code 4.10 provides insights into how the LBM_step task class is implemented in JDF. SUBGRID_FROM and SUBGRID_TO are parametrized flows that provide the rules for getting and sending the subgrids. They both correspond to 3 dataflows, one for each *dy* offset. Hence, when we enter the body of the task class, we have access to the 3 *in* and *out* kinetic grids. We also have access to the INTERFACE_UP and INTERFACE_DOWN data, which correspond to the vertical buffers using the method shown in Figure 4.3.

Let us focus on the parametrized flow declaration `READ SUBGRID_FROM[dy = 0..2]`. Three *in* dependencies are declared, depending on the time step. These *in* dependencies are read from up to down, meaning that if the first condition is matched, the following ones are not checked. The first one expresses where the *in* data can be found if the data has been saved at the previous step, while the second one indicates where the data can be found if a horizontal synchronization has been performed. The last one is the general case, where the data is read from the previous *LBM_step* task class.

The rest of the task classes are not necessarily relevant to detail the current discussion. The *write_horizontal_slices* and *read_horizontal_slices* task classes interleave the *LBM_step* task class to allow for the synchronization of the horizontal buffers. The *vertical_interface* and *horizontal_interface* task classes only act as intermediary task classes. The *save_file* and *save_file_reduce* task classes have not been designed to be efficient and have only been implemented to test the correctness of the implementation.

Most GPU kernels are also not detailed in this document, as they do not constitute a performance bottleneck. The only kernel that has been optimized is the *LBM_step* kernel, which is the most computationally intensive kernel, as it typically represents about 99% of the execution time. We propose three different implementations of the *LBM_step* kernel in Codes 4.11, 4.12, and 4.13.

```
__global__
void LBM_step_naive(
                    float *subgrid_FROM_DY[3], float *subgrid_TO_DY[3],
                    int x_margin, // depth of the cells we ignore on the x-axis,
                    PRECISION *interface_down, PRECISION *interface_up, // interface buffers
                    )
{
    int cellNum = subgrid_size_x * subgrid_size_y;

    for (int id = blockIdx.x * blockDim.x + threadIdx.x; id < cellNum; id += blockDim.x*gridDim.x)
    {
        int subgrid_true_x = id % subgrid_size_x;
        int subgrid_true_y = id / subgrid_size_x;

        // Tell if the cell necessitates a computation
        bool need_computation =
            subgrid_true_x >= x_margin && subgrid_true_x < subgrid_size_x - x_margin;

        // Read the data from the subgrid
        float f[9];
        for(int d=0; d<9; ++d)
        {
            // dx and dy are the offsets by which we shift the values
            int dy = velocities[d][1]; // -1, 0, 1
            int dx = velocities[d][0]; // -1, 0, 1
            int target_true_x = subgrid_true_x - dx;
            int target_true_y = subgrid_true_y - dy;

            // The ghost cells on the y-axis are handled by the interface buffers
            if(is_in_ghost_area_up(dy, target_true_y))
                f[d] = interface_up[target_true_x+(dx+1)*subgrid_size_x];
            else if(is_in_ghost_area_down(dy, target_true_y))
                f[d] = interface_down[target_true_x+(dx+1)*subgrid_size_x];
            // Else, we read the neighbot value from the subgrid
            else if(need_computation)
                f[d] = subgrid_FROM_DY[dy+1][target_true_y * subgrid_size_x + target_true_x];
        }
        // Now f contains the values of the 9 neighbors

        if(need_computation)
            phy(f); // Compute the new values of f (relaxation, equilibrium, etc.)

        // Write the new values to global memory
        for(int d=0; d<9; ++d)
        {
            int dy = velocities[d][1];

            // Update the current subgrid
            if(need_computation)
                SUBGRID_TO_DY[dy+1][subgrid_true_y * subgrid_size_x + subgrid_true_x] = f[d];

            // If on the edge, update the interface buffers
            if(is_in_edge_area_up(dy, subgrid_true_y))
                interface_up[subgrid_true_x+(1+dx)*subgrid_size_x] = f[d];
            if(is_in_edge_area_down(dy, subgrid_true_y))
                interface_down[subgrid_true_x+(1+dx)*subgrid_size_x] = f[d];
        }
```

```
    }
}
```

Code 4.11: Pseudo-CUDA code for the naive LBM_step kernel.

```
__global__
void LBM_step_optimized(...)
{
    int dimX = subgrid_size_x - 2*x_margin;
    int dimY = subgrid_size_y;
    int cellNum = dimX * dimY;

    for (int id = blockIdx.x * blockDim.x + threadIdx.x; id < cellNum; id += blockDim.x*gridDim.x)
    {
        int rx = id % dimX;
        int ry = id / dimX;
        int subgrid_true_x = x_margin + rx;
        int subgrid_true_y = ry;

        // We do not need to check if the cell is in the computation area

        // The rest of the kernel is the same as the naive kernel
    }
}
```

Code 4.12: Pseudo-CUDA code for the optimized LBM_step kernel.

```
__global__
void LBM_step_per_line(...)
{
    int x_min = x_margin;
    int x_max = subgrid_size_x - x_margin - 1;
    int y_min = 0;
    int y_max = subgrid_size_y - 1;

    for(int true_y=y_min+blockIdx.x;true_y<=y_max;true_y+=gridDim.x)
    {
        for(int true_x=x_min+threadIdx.x;true_x<=x_max;true_x+=blockDim.x)
        {
            // The rest of the kernel is similar to the naive kernel (but no check for the
     computation area)
        }
    }
}
```

Code 4.13: Pseudo-CUDA code for the per-line LBM_step kernel.

These codes differ in the way the CUDA grid is organized. Since we have a margin on the x-axis (due to the ghost area), no computations is needed for the cells in this area. The naive kernel iterates on the whole grid regardless of the margin and ignores the cells that do not need to be computed. The optimized kernel only iterates on the cells that need to be computed. The per-line kernel iterates so that one line along the x-axis is computed by one CUDA block. It is not clear what approach is the best. The naive kernel ensures perfectly coalesced accesses, while the other two lower idle thread time. In the next section, we perform various experiments, one of which is to determines the most efficient version.

4.6 Performance Evaluation

4.6.1 Benchmarking the CUDA kernels

In this first study, we aim to determine the most efficient configuration for running the D2Q9 LBM stencil. There are two major unknowns: the chosen depth of the overlap on the x-axis and the implementation for the *LBM_step* kernel. To assess the impact of these parameters, we run the D2Q9 stencil for 4500 iterations on a 20736×10368 (single precision) grid, with an A100 GPU. We make the overlap vary from 1 to 32 and thest the naive, the adjusted grid, and the per-line implementations of the *LBM_step* kernel. The block configurations have been fine-tuned for each kernel on an overlap value of 10. The shown results are the average of 16 runs.

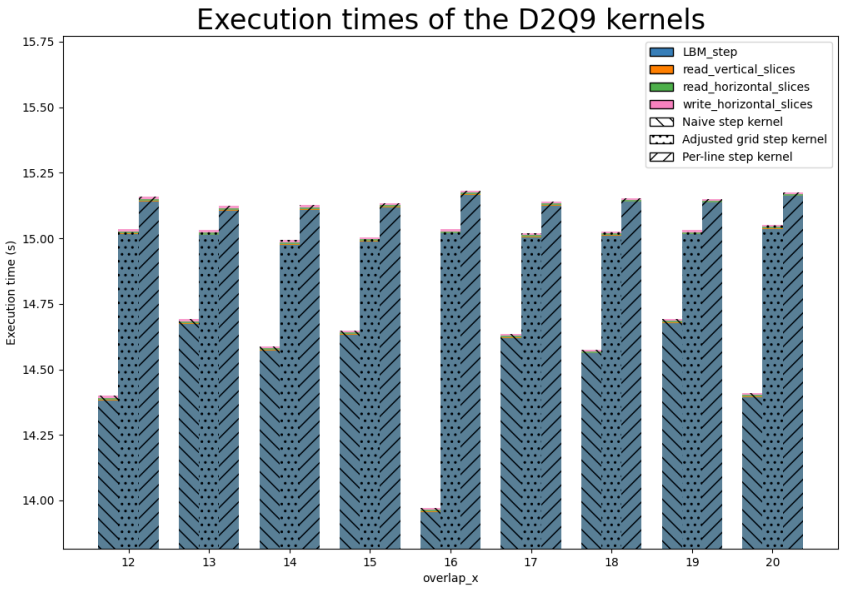


Figure 4.5: Comparison of the execution times of the CUDA kernels for the D2Q9 stencil under different configurations. The *overlap_x* is the depth of the ghost cell area over the x-axis. The different *LBM_step* kernels are denoted by different hatch patterns, from left to right: the naive kernel (described in Code 4.11, the adjusted grid kernel (described in Code 4.12), and the per-line kernel (described in Code 4.13).

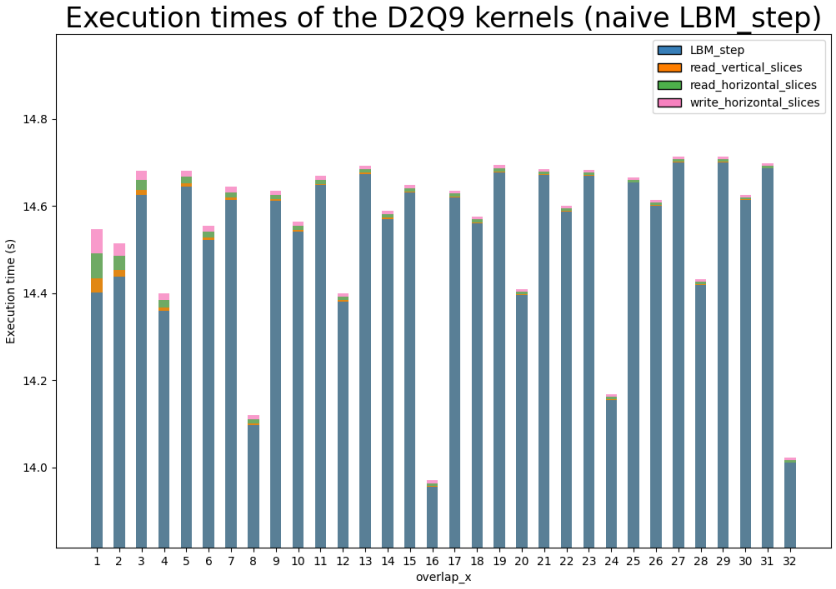


Figure 4.6: Comparison of the execution times of the CUDA kernels for the D2Q9 stencil with different overlaps on the x-axis. Only the naive *LBM_step* kernel is considered.

Figure 4.5 presents the execution times for various *LBM_step* kernels with overlap values ranging from 12 to 20. This analysis reveals that the naive kernel outperforms its counterparts across the tested overlap

values, establishing itself as the fastest. However, it is also observed that the performance of the naive kernel is highly sensitive to changes in the overlap value.

To delve deeper into this phenomenon, Figure 4.6 specifically examines the naive *LBM_step* kernel, extending the range of overlap values from 1 to 32. On this new scale, it appears clearly that the best performance is achieved when the overlap size aligns with powers of two. The best performance is achieved for an overlap of 16 and 32, followed by 8 and 24, and then 4, 10, 20, and 28. The reason for this behavior likely lies in the memory access pattern of the naive kernel. The naive *LBM_step* kernel is written in such a way that all threads are perfectly coalesced for memory accesses and threads that target the ghost cells are simply not used. When the overlap is 16, the first and last warps will use exactly half of their threads (as long as the subgrid size is a multiple of 32) and the CUDA grid will be perfectly aligned with the subgrid. Although it is not clear whether these performance peaks are due to better balancing of the scheduler or to a hardware feature, it is not surprising to see that such improvements are not possible or less efficient for other overlap values.

Overall, it is observed that the *LBM_step* kernel is by far the most influential factor in the overall performance of the D2Q9 stencil. The other kernels, while still having a slight impact for low overlap values, become negligible as the overlap increases. This is expected, as the synchronization mechanism on the x-axis has been designed to optimize data accesses for the GPU.

The conclusion of this first experiment is that optimal performance is reached with the naive *LBM_step* kernel and an overlap of 16. For the moment, only the GPU time of the kernels has been measured. The next step is to measure the time of the entire D2Q9 stencil, including the potential overhead of PaRSEC. In the next section, we will see how the global execution performs with or without parametrized flows.

4.6.2 Comparison with standard PaRSEC dataflows

In this section, we want to assess the computational cost of using parametrized flows. In the tested D2Q9 application, all the parametrized dataflows depend on the y-offset of the stencil, which is known at compile time. This makes it possible to unroll the parametrized flows by hand, by duplicating them and modifying the correponsing values, as shown in Code 4.6. The dependencies (of the flows) can also be unrolled by hand, by adding a rule in the condition for each possible value of the parameter, as demonstrated in Code 4.14.

```
RW SUBGRID_SAVE_REDUCE
    // If first save, retrieve from initial condition
    <- (s==0)
        ? SUBGRID_D FillGrid(subgrid_x, subgrid_y, dy)
    // Else, retrieve from the result of LBM_STEP
    <- (true)
        ? SUBGRID_TO[dy] LBM_STEP(subgrid_x, subgrid_y, (s)*save_interval-1)
    // Send to the next LBM_STEP
    -> (s*save_interval < number_of_steps)
        ? SUBGRID_FROM[dy] LBM_STEP(subgrid_x, subgrid_y, s*save_interval)

// Can be unrolled to

RW SUBGRID_SAVE_REDUCE
    <- (s==0)
        ? SUBGRID_D FillGrid(subgrid_x, subgrid_y, dy)
        <- (true && dy==0)
            ? SUBGRID_TO_0 LBM_STEP(subgrid_x, subgrid_y, (s)*save_interval-1)
        <- (true && dy==1)
            ? SUBGRID_TO_1 LBM_STEP(subgrid_x, subgrid_y, (s)*save_interval-1)
        <- (true && dy==2)
            ? SUBGRID_TO_2 LBM_STEP(subgrid_x, subgrid_y, (s)*save_interval-1)
    // Send to the next LBM_STEP
    -> (s*save_interval < number_of_steps && dy==0)
        ? SUBGRID_FROM_0 LBM_STEP(subgrid_x, subgrid_y, s*save_interval)
    -> (s*save_interval < number_of_steps && dy==1)
        ? SUBGRID_FROM_1 LBM_STEP(subgrid_x, subgrid_y, s*save_interval)
    -> (s*save_interval < number_of_steps && dy==2)
        ? SUBGRID_FROM_2 LBM_STEP(subgrid_x, subgrid_y, s*save_interval)
```

Code 4.14: Example of unrolling the dependencies of a parametrized flow. The *dy* parameters ranges from 0 to 2 for the sake of the example.

This lets us compare both versions of the D2Q9 stencil, one with parametrized flows and the other with unrolled flows. For the version with unrolled flows, PaRSEC is compiled with disabled parametrized flows

and is, hence, close to the standard version of PaRSEC. Both versions are compiled with the highest level of optimization and the same compiler flags. The shown results are the average of 256 runs. The rest of the parameters are the same as in the previous experiment.

	GPU kernels	initialization	whole run	PaRSEC overhead
Parametrized	14.5793 s	0.0226 s	16.3202 s	11.94%
Non-parametrized	14.5786 s	0.0225 s	16.3187 s	11.94%

Table 4.1: Execution times for the parametrized and non-parametrized versions of the code.

Table 4.1 presents the results of this experiment. The first column shows the execution time of the GPU kernels, the second one shows the execution time of the initialization time, and the third one shows the total execution time. The initialization time is the time taken to create the PaRSEC context, the task classes, and the dataflows. The PaRSEC overhead is computed as the relative difference between the total execution time and the execution times of the GPU kernels.

The initialization time is expected to be greater for the parametrized flows version, as the generated code adds more work for the rewiring of the data structure. However, this experiment shows that the overhead of the initialization in the parametrized flows is not noticeable. Otherwise, the other times are similar for both versions, which shows that the changes made to the PTG compiler do not significantly impact the performance of PaRSEC.

4.7 Discussion

In this work, we have introduced the possibility of using parametrized flows in PaRSEC. We have demonstrated that this feature can be used to implement a D2Q9 stencil with results similar to what can be achieved with standard PaRSEC dataflows. The current state of the implementation does not handle execution on multiple nodes, which is a serious, but not definitive, limitation.

The main advantage of using parametrized flows is that it allows for a more compact and readable code. It is also a bit more flexible, as the number of dataflows can be deduced at initialization time, rather than at compile time with standard PaRSEC dataflows. With parametrized flows, it is theoretically possible to implement a generic JDF stencil distribution code that would work for different stencil shapes and number of dimensions. We have seen that the impact on the performance of using parametrized flows is negligible in practice, which is a good sign for the future of this feature.

Some limitations of parametrized flows are due to the use of PaRSEC as a runtime system. There is a practical maximum number of dataflows that can be created for a given task class. This maximum number depends on the overall dependency structure of the program and is caused by the use of internal masks of size 32.

Overall, this work is a first step towards a more flexible and expressive DSL. We have shown that it is possible to implement a high-performance stencil with parametrized flows in PaRSEC. Future work will focus on fixing the limitations of the current implementation and on evaluating the performance of PaRSEC on multiple nodes.

This concludes our contribution to the field of stencil computations using the PaRSEC framework. In the next section, we present the rest of our contribution to this field, with a focus on the StarPU runtime system.

Chapter 5

Efficient ditributed stencils on StarPU

In the previous chapter, we have presented our contributions to the efficient parallelization of stencil-based algorithms using PaRSEC, a task-based runtime system. We refer to Section 4.1 of the aforementioned chapter for a general introduction to task-based runtime systems and the justification as to why they are relevant for the parallelization of stencil-based algorithms. In this chapter, we present our work on the parallelization of stencil-based algorithms using StarPU, another task-based runtime system.

In this second work, we use the StarPU runtime system [20] to parallelize stencil-based algorithms. We use a more generic approach than in the previous work with PaRSEC and, hence, make as few assumptions as possible about the shape of the stencil. The main goal of this work is to create a generic framework for efficient task-based parallelization of stencils using StarPU.

StarPU is a task-based runtime system that aims to provide a unified interface for parallelizing applications on heterogeneous architectures. It provides support for multiple technologies, such as OpenCL, CUDA, and MPI. While StarPU shares some similarities with PaRSEC, it also has some key differences, including a different programming model. Instead of using an algebraic description of the dependencies between tasks, StarPU uses a dynamic approach where the dependencies are resolved at runtime. It has demonstrated high potential in various fields [10, 11, 64]. In this section, we present the main features of interest of StarPU, and how they can be leveraged to implement multi-GPU Lattice-Boltzmann simulations.

In Section 5.1 and Section 5.2, we present the StarPU programming model and how it can be leveraged to parallelize stencil-based algorithms. Then, in Section 5.3, we present preliminary results we have obtained on the D2Q7 scheme we presented in Section 3.2. Finally, in Section 5.4, we present the latest results we have obtained on a larger simulation.

5.1 Programming Model

Initially, StarPU employed a method where programmers explicitly defined a task graph as a Directed Acyclic Graph (DAG), specifying both the tasks and their dependencies. This direct approach, however, was not the most user-friendly, prompting a shift towards a more abstract and dynamic method. The improvement lies in the automatic deduction of the task graph from the data accesses declared at task submission, based on the principle of sequential consistency order [145].

Adopting the Sequential Task Flow (STF) paradigm [9], StarPU significantly streamlined this process. In STF, tasks are sequentially submitted to the runtime system without the need for programmers to explicitly define dependencies among them. It becomes the responsibility of the runtime system to ensure execution coherence by analyzing the data accesses of each submitted task. This evolution simplifies task management and enhances user experience by abstracting the complexity of dependency management.

```
for (k = 0; k < NT; k++) {
    POTRF (A[k][k]);
    for (m = k+1; m < NT; m++) {
        TRSM (A[k][k], A[m][k]);
    }
    for (n = k+1; n < NT; n++) {
        SYRK (A[n][k], A[n][n]);
        for (m = n+1; m < NT; m++) {
            GEMM (A[m][k], A[n][k], A[m][n]);
        }
    }
}
```

Code 5.1: Pseudo-code for generating the Cholesky factorization graph.

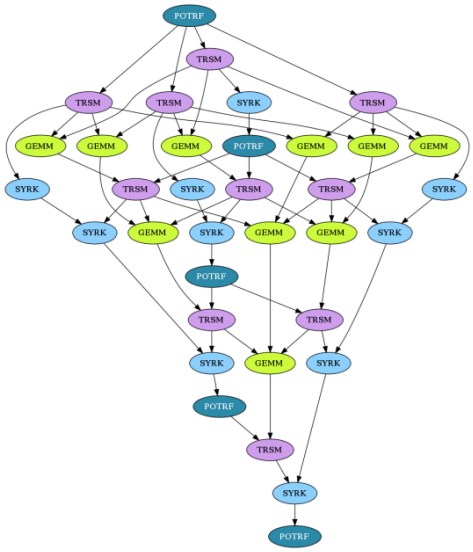


Figure 5.1: Task graph for the Cholesky factorization algorithm.

Figure 5.1 and Code 5.1, as introduced by Thibault [265], illustrate a tiled Cholesky factorization algorithm and its associated task graph. The algorithm is coded sequentially, comprising a series of operations such as `POTRF`, `TRSM`, `SYRK`, and `GEMM`. These operations interact with data segments (tiles) and their access types (e.g., read, write, read-write), forming the dependency relations among tasks. Such dependencies are representable via a DAG, with nodes signifying tasks and edges denoting task dependencies. It is essential to note that such DAGs are rarely explicitly constructed by the runtime system, as such structures become inefficient with increasing number of tasks.

The Cholesky factorization example underscores the advantages of the STF paradigm. It simplifies the programming process by allowing a straightforward algorithmic description without the need to manually outline dependencies, enhancing the ease of programming significantly. Compared to more verbose models like PTG, where describing dependencies may introduce errors and excessive boilerplate code, the STF paradigm stands out for its simplicity. The balance between the effort required to specify dependencies and the potential performance benefits varies by application and runtime system. However, numerous applications utilizing StarPU have reported leading performance across diverse domains [12, 10, 42, 143, 11, 178, 15, 257, 18, 52, 65, 14], indicating an effective compromise between programming convenience and performance. Therefore, opting for STF in parallelizing stencil-based algorithms appears to be a judicious choice. In the forthcoming sections, we will delve into our methodology for distributing stencil-based algorithms using StarPU.

5.2 Distributed stencil-based algorithms with StarPU

Our strategy for parallelizing stencil-based algorithms with StarPU aligns with the principles outlined in Section 3.3 and Chapter 4. The domain is segmented into subgrids to facilitate distribution across computational resources. Synchronization of these subgrids, essential for updating ghost cells, is managed through an intermediary buffer, optimizing inter-node communication efficiency.

In StarPU, explicit data management is achieved via *handles*, abstract data constructs that encapsulate metadata about the data, such as its memory location or size. These handles are independent from the actual data and are used to declare data accesses at task submission.

Our method involves utilizing two handles for each subgrid, designated for reading and writing, respectively. These "read" and "write" handles are alternated with each simulation time step.

The management of intermediate buffers can be performed in different ways. One approach, depicted in Code 5.2, is to allocate a temporary buffer for each of the necessary "sends". This approach is convenient as it matches the way the interfaces should be exchanges, but could lead to an increased overhead due to continuously allocating and deallocating memory handles. Moreover, it can result in lowered parallelism if the tasks are not inserted properly. Another approach is to use a fixed number of buffers, which are allocated at the beginning of the simulation and released at the end. The number of buffers must be carefully chosen to prevent deadlocks or resource starvation. To be sure that no starvation occurs, we can assign two buffers per subgrid interfaces, a subgrid interface being a shared face between two subgrids in the 3D case. One buffer is associated to the A to B direction and the other to the B to A direction, A and B being the two subgrids. Code 5.3 shows how the task insertions for the synchronization can be done with a fixed number of buffers.

```
for (t = 0; t < NB_STEPS; t++) {

    // Perform a time step on each subgrid
    for (i = 0; i < num_subgrids; i++) {
        // Perform the LBM step
        insert_task(
                step, // Task type
                READ, subgrid_handles[i%2][i], // Input data
                WRITE, subgrid_handles[(i+1)%2][i] // Output data
                );
    }

    // Send the bordering cells to the neighbors if it is not the last time step
    if(t < NB_STEPS-1) {
        for (i = 0; i < num_subgrids; i++) {
            // List of directions in which there is an exchange
            const directions = [(1,0), (-1,0), (0,1), (0,-1)];
            for (d in directions) {
                // Get the entry in subgrid_handles corresponding to the neighbor for the next
    ((i+1)%2) time step
                neighbor = get_neighbor(i, d, (i+1)%2);

                buffer = new_starpu_handle(); // "allocate" a temporary buffer

                // Copy the bordering cells from subgrid i to the temporary buffer
                insert_task(
                        subgrid_to_slice, // Task type
                        READ, subgrid_handles[(i+1)%2][i], // Input: the current subgrid
                        WRITE, buffer // Output: the temporary buffer
                        );

                // Write the buffered values to the neighbor
                insert_task(
                        slice_to_subgrid, // Task type
                        READ, buffer, // Input: the temporary buffer
                        RW, neighbor // Output: the neighbor
                        );

                relieve_starpu_handle(buffer); // "free" the temporary buffer
            }
        }
    }
}
```

Code 5.2: Pseudo-code for the insertion of tasks using temporary buffers. The LBM step is first performed on each subgrid, followed by the exchange of bordering cells with neighboring subgrids. This method offers little parallelism, because the synchronization imposes a strong sequential constraint, as `subgrid_to_slice` can only be called if the read-write of the `slice_to_subgrid` (neighbor) has been completed.

```
slice_handles[SLICE_NUM] = initialize_all_interfaces();

for (t = 0; t < NB_STEPS; t++) {
    // For each subgrid
    for (i = 0; i < num_subgrids; i++) {

        // If it is not the first time step, we update the ghost cells from the buffers
        if(t > 0) {
            const directions = [(1,0), (-1,0), (0,1), (0,-1)];
            for (d in directions) {
                insert_task(
                        slice_to_subgrid, // Task type
```

```
                        READ, slice_handles[get_slice_index(t, i, d)], // Input: the slice
                        RW, subgrid_handles[i%2][i] // The subgrid to update
                        );
            }
        }

        // Now that the ghost cells are up-to-date, we can perform the LBM step
        insert_task(
                step, // Task type
                READ, subgrid_handles[i%2][i], // Input data
                WRITE, subgrid_handles[(i+1)%2][i] // Output data
                );

        // If it is not the last time step, we fill the buffers
        if(t < NB_STEPS-1) {
            const directions = [(1,0), (-1,0), (0,1), (0,-1)];
            for (d in directions) {
                insert_task(
                        subgrid_to_slice, // Task type
                        READ, subgrid_handles[(i+1)%2][i], // Input: the current subgrid
                        WRITE, slice_handles[get_slice_index(t+1, i, d)] // Output: the slice
                        );
            }
        }
    }
}
```

Code 5.3: Pseudo-code for the insertion of tasks using a fixed amount of buffers. The execution is more streamlined, at the cost of more memory usage, as the slice buffers must remain allocated during the entire simulation.

To achieve distribution across multiple nodes, we use the StarPU MPI backend. This backend works by wrapping the MPI calls in StarPU calls, allowing for a seamless integration of MPI communication in the task-based programming model. It is based on a user-defined repartition of the handles across the nodes. Based on this repartition, StarPU can at all times determine which node is responsible for a given handle and perform the necessary data transfers and execute the tasks at an appropriate time to ensure data consistency. Concretely, The distributed version of our code differs in the following ways:

- The `starpu_variable_data_register` calls, which are used to register the data to StarPU, are replaced by `starpu_mpi_data_register`;
- The `starpu_mpi_data_register` function requires a rank parameter to specify the MPI rank that will be responsible for the data. We set the ranks so that the subgrids are grouped in contiguous clusters of approximately the same size;
- The `starpu_task_insert` calls are replaced by `starpu_mpi_task_insert`, which will silently handle the MPI communication.

With this methodology, we can expect satisfactory performance on a distributed memory system. The fact that the subgrids are assigned once and for all to a node ensures that they are never moved through the network, which would be detrimental to performance. In the next section, we conduct preliminary experiments to validate our approach. The goal is to test whether the proposed method is suitable for the parallelization of stencil-based algorithms.

5.3 Validation of the method

To evaluate the performance of this approach, we conducted experiments on a D2Q7 scheme. This scheme, described in Section 3.2.2, is convenient thanks to its order 2 convergence, which allows us to be near-certain that the implementation is correct when we observe the same convergence rate. The reference implementation is *Patapon* [29], whose implementation is close to the one we present in Section 3.2.3. These preliminary results have been presented at the Compas 2022 conference [96].

5.3.1 Methodology

The goal of the test case is to compare the performance of our StarPU implementation (*TB-LBM*; Task-Based Lattice-Boltzmann Method) with the reference implementation *Patapon*. There are several differences between the two implementations:

- *Patapon* uses OpenCL, while *TB-LBM* uses CUDA;
- *Patapon* can only use 1 GPU, whereas *TB-LBM* can distribute the work on multiple processors;
- *TB-LBM* always use the StarPU backend and the subgrid system, which creates an overhead.

We present the results for 3 configurations: 1 GPU, 2 GPUs, and 2 GPUs + 1 CPU. For the 1-GPU configuration, we set a single subgrid of the same size as the global grid and an overlapping depth of 1. For the other 2-GPU configurations, we divide the space into $4 \times 4 = 16$ subgrids and use an overlap of 8.

We use different grid dimensions: *small* (1024), *medium* (2048), *large* (4096) and *huge* (8192). The data of the *huge* test case does not fit into the memory of one of the tested GPU and cannot be simulated by *Patapon*. TB-LBM with 1 GPU could execute this test case if the grid was split into smaller subgrids, but we impose that there is only one subgrid in this benchmark. We adapt the number of performed time steps depending on the grid size. The *huge* case performs 64 time steps, the *large* one performs 256, the *medium* 1024, and the *small* 4096 (we multiply by 4 between each case). This allows us to expect the number of computations to be the same between the different grid sizes.

The following parameters are common to all the test cases:

- Hardware: the experiments were performed on a single node with two Intel(R) Xeon(R) CPU E5-2683 v4 at 2.10GHz (32 cores in total). The node also has two NVidia P100 GPUs, each with 16GB of memory;
- Software: we use the 11 February 2022 commit of the master branch of StarPU, GCC 9.3, and CUDA 11.2. Our application has been compiled with the following flags *-03* with GCC and *-O3 -arch=sm_60* with nvcc. We use the StarPU scheduler DMDA (HEFT).

5.3.2 Analysis

Figure 5.2 shows the execution times for the different configurations and grid sizes. We observe that the performance between the different configurations varies greatly depending on the grid size. For the *medium* and the *large* grid sizes, going from 1 GPU to 2 GPUs speedups the execution by about 2. It is an argument in favor of our strategy because it means that we do not lose excessive performance with our decomposition. For a *small* grid size, however, the TB-LBM with 1 GPU is the fastest configuration of the four. This is a surprising result that is likely due to the small size of the data that makes the face exchanges very efficient.

Patapon is relatively consistent in terms of execution time throughout the possible grid sizes. TB-LBM is more fluctuant, which can be explained by the fact that we use a system of subgrids and slice exchanges that induces memory transfers and scheduling choices that we do not control, whereas *Patapon* has a linear transferless execution. The difference between *Patapon* and TB-LBM also depends on the grid size:

- For the *small* size, TB-LBM is always faster than *Patapon*.
- For the *medium* size, *Patapon* is always faster.
- For the *large* size, *Patapon* is faster than the 1-GPU configuration but slower than the two 2-GPU configurations.

In theory, we would expect that the *step* kernel of *Patapon* is slower or comparable to that of 1-GPU TB-LBM because OpenCL kernels are generally slower than CUDA ones. In this experiment, the *medium* and *large* test cases do not behave as expected. There are 3 main differences between *Patapon* and TB-LBM that can explain this. The first one is the *step* kernel that is slightly different. The second difference is the use of StarPU itself which can lead to inefficient scheduling choices. In our case, we observed that only the DMDA scheduler provides satisfactory results. Finally, the last and more likely difference that could explain

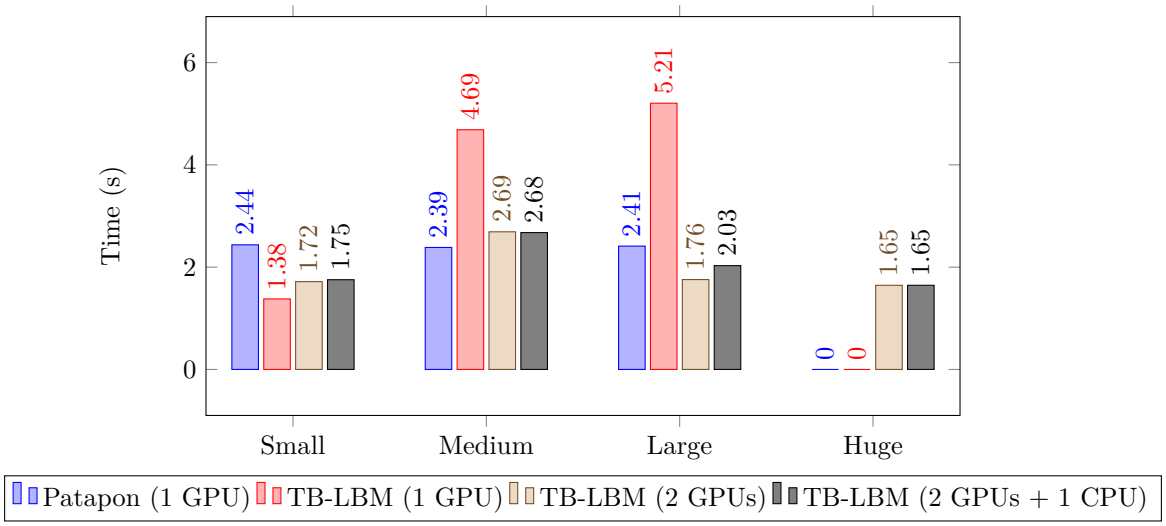


Figure 5.2: Execution times for the different grid sizes (*small*, *medium*, *large* and *huge*) and three hardware setup alternatives. *Patapon* is a state-of-the-art software that is only able to run on 1 GPU. TB-LBM implements the subgrid mechanism and is able to run on multicore hybrid configurations.

that TB-LBM is sometimes slower is the regular use of a synchronization mechanism to keep the halo of each subgrid coherent. In TB-LBM 1-GPU, there is only one subgrid but the halo of this subgrid still needs to be synchronized between the opposite sides which induces a substantial amount of additional read/writes. In *Patapon*, there is no such synchronization as the *step* kernel accesses the neighbors with a modulo operation. For the 2-GPU and the 2-GPU + CPU configurations, the difference with *Patapon* is less noticeable. On the other hand, adding the CPU processor does not appear to help the execution.

Finally, the result of interest is the *huge* test case. Since the data do not fit into a single GPU, we measure the ability of TB-LBM to distribute the algorithm. The test cases are designed in such a way that the amount of operation stays the same. We can, therefore, extrapolate that the computational throughput of *Patapon* would theoretically allow an execution time of ≈ 2.4s in the *huge* test case which corresponds to a speedup of approximately 31%. This demonstrates the relevance of using StarPU and, by extension, the task-based method for performing LBM simulations on a large scale.

When these experiments were initially conducted, the StarPU MPI backend was not fully operational, preventing us from testing our implementation on multiple nodes. However, the results from these early tests were promising and convinced us that completing the MPI backend was a worthwhile endeavor. Since then, we have successfully implemented and launched the MPI backend. In the next section, we will evaluate the overall performance of this now operational MPI implementation by conducting a larger distributed simulation.

5.4 Evaluation on a larger simulation

In this section, we aim to go further in the evaluation of the TB-LBM implementation. To achieve this, we run a D3Q27 stencil, which is the same we present later in Chapter 9. Hence, the results we present here can be fairly compared with those of the aforementioned chapter. We refer to Section 9.3.1 for the details of the numerical scheme.

5.4.1 Methodology

Technical details

Unless stated otherwise, the physical parameters are the same as in Chapter 9. The implementation is adapted to match the StarPU programming model. The `LBM step` and subgrid synchronization kernels are

encapsulated in StarPU tasks. The synchronization is is performed thanks to two tasks: `subgrid_to_slice` and `slice_to_subgrid`. The slices are of size $N_1 \times N_2 \times 9$, where N_1 and N_2 are the dimensions of the subgrid along the two axes perpendicular to the direction of the exchange. We limit the number of exchanged velocities to 9 using the same argument as in Figure 4.3.

Each node is permanently assigned a specific set of subgrids. The subgrids are distributed in such a way that each node is responsible for a contiguous set of subgrids. The configuration of computational resources, including the number of nodes, the quantity of GPUs per node, and the scheduling strategy employed, are dynamically determined based on the experimental requirements.

Hardware setup

The experiments are conducted on the sirocco07-13 nodes of the PlaFRIM platform. Each node is equipped with two Intel Xeon E5-2683 v4 CPUs, each with 16 cores, and two Nvidia P100 GPUs. The nodes are interconnected via an OmniPath 100 Gbit/s network. The GPUs are connected to the CPUs via PCI Express, and the network is based on OpenFabrics.

5.4.2 Assessing the best scheduling strategy

The scheduling is a crucial aspect of the performance of a task-based runtime system. The impact of the scheduling strategy on the performance of the simulation depends on several factors, such as the hardware setup, the dependency structure of the tasks, the granularity of the tasks, etc. StarPU offers several scheduling strategies, each with its own strengths and weaknesses. Presumably, in our case, the efficiency of the overall simulation will heavily benefit from achieving good data locality. This is because the entire design of the application is based on the premise that most of the data transfers are the exchanged interfaces A bad choice of scheduling can easily lead to a situation where the subgrids are moved from one GPU to another on each time step, which would be detrimental to performance. In StarPU, three scheduling strategies are thought to handle this aspect particularly well: *dmda*, *dmdas*, and *Heteroprio*. The *Lws* strategy is also theoretically designed to encourage data locality, but it is generally less efficient than the other three strategies. Here, we focus on evaluating different scheduling strategies to identify those that could be most relevant for enhancing performance across various setups.

Figure 5.3 shows the execution times of the D3Q27 scheme with the four scheduling strategies on a 4-node setup, each with two GPUs. The box plots represent the execution times for 32 runs of the simulation. To ensure the grid size is large enough to warrant distribution across multiple GPUs, the total logical grid size is $384 \times 1536 \times 384$, accounting for a total grid size of 23.328 GB (if stored as a single array). This grid is divided into $3 \times 12 \times 3$ subgrids, aiming to achieve enough granularity to distribute the work across the available computational resources effectively.

In our analysis, we observed that the *dmda* and *dmdas* scheduling strategies consistently outperformed the *Heteroprio* approach in terms of execution times. Conversely, the *Lws* strategy exhibited inferior performance, which aligned with our expectations due to its known limitations in handling the granularity of the tasks and dependencies effectively.

Although *Heteroprio* performed slightly worse in our initial tests, we believe it should be able to outperform *dmda* and *dmdas* with the appropriate tuning, in particular by incorporating more spatial-awareness. The main reason for this belief is that *Heteroprio* works by assigning priorities to task types. In our case, since we did not make a distinction between tasks within a task type (i.e., all *step* tasks have the same priority), the scheduler does not have the necessary information to make the best decisions. Since the different in execution times between the different scheduling strategies appears relatively small, it can be argued that with proper tuning, *Heteroprio* could be the most efficient strategy. Consequently, throughout the rest of this work, we continue to employ *Heteroprio* and refine this strategy in subsequent experiments, aiming to explore its adaptability and improve its efficiency within our simulation framework.

5.4.3 Performance evaluation

In this section, we evaluate the performance of the proposed implementation. We first assess the bottlenecks of the implementation and then conduct strong and weak scaling experiments to evaluate the capacity of

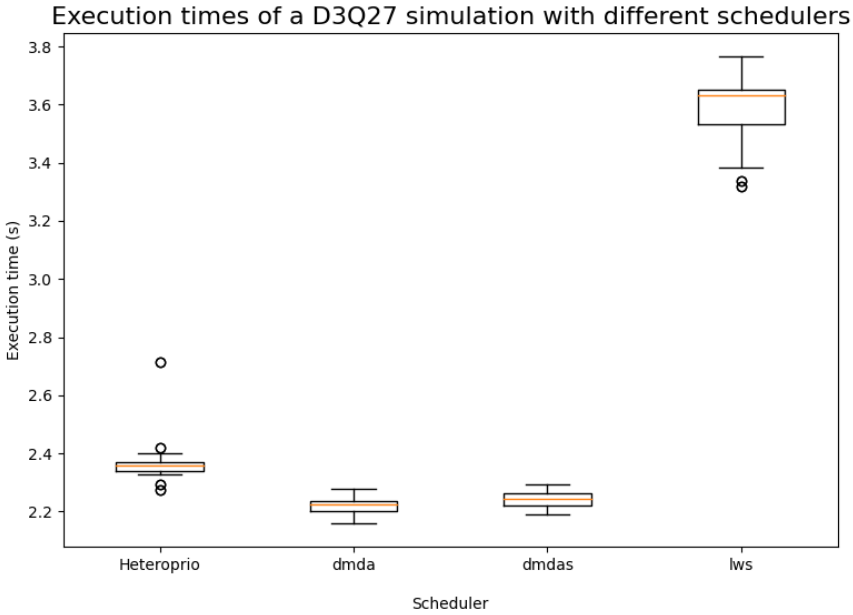


Figure 5.3: Execution times of the D3Q27 scheme on 4 nodes and 2 GPU per node with different scheduling strategies. The total grid is divided into $3 \times 12 \times 3$ subgrids. The work is distributed across four nodes, each equipped with two P100 GPUs.

the implementation to parallelize the simulation. These results aim to provide insights into the efficiency of the implementation and identify potential areas for improvement.

Assessing the bottlenecks

In this first experiment, we aim to identify the main bottlenecks of the implementation. For this, we set a configuration which should be close to real use cases, but with no attempt to fine-tune the parameters. The total grid is divided into $2 \times 16 \times 1$ subgrids and we perform 32 time steps. We run the simulation on 2 nodes, each equipped with 2 GPUs.

The execution with one GPU per node lasted 4.57 seconds, whereas the execution with two GPUs per node lasted 4.13 seconds. The proportion of time dedicated to performing actual computation kernels is approximately 27% for the single-GPU configuration and 15% for the dual-GPU configuration. These results indicate unexpectedly high non-kernel execution times. Additionally, the minimal performance improvement between the single-GPU and dual-GPU configurations is unexpected. Ideally, a speedup close to 2 would be anticipated; however, the observed speedup is only 1.11. This suggests an implementation issue, which we aim to diagnose by analyzing the execution traces.

Figures 5.4 shows the execution traces when limited to using a single GPU per node, while Figure 5.5 shows the trace with no constraint on the number of GPUs used. These traces are generated thanks to the FxT [80] tool, which is embedded in StarPU. Then, the ViTE tool [75] is used to visualize the traces.

These traces highlight the communication bottlenecks of the implementation. Let us distinguish between two types of communication bottlenecks: the inter-node MPI communications and the intra-node communications. The intra-node communications only occur in the dual-GPU configuration (Figure 5.5), where data can be exchanged between the two GPUs of a node. On the traces, we can see that the purple region (intra-node communications) accounts for a significant portion of the execution time, which explain the limited speedup between the single-GPU and dual-GPU configurations. Most of these purple bands are associated with a single data transfer (white arrow). Further analyses reveal that these costly data transfers are attributed to whole subgrid exchanges between the GPUs. These exchanges can occur if the scheduler decides to move a subgrid from one GPU to another, typically to avoid a load imbalance.

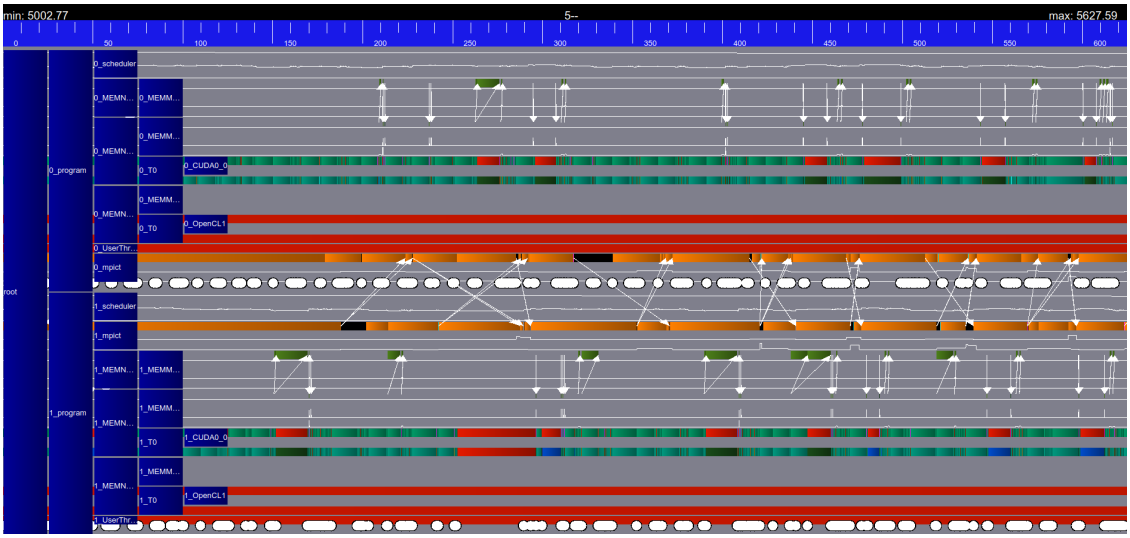


Figure 5.4: Execution trace of the D3Q27 scheme on 2 nodes, each with a single GPU. The time frame has been chosen arbitrarily and is 500 ms long. Red sections indicate worker idle time, orange sections show MPI communications, and green sections represent kernel execution. White arrows illustrate data transfer trajectories.

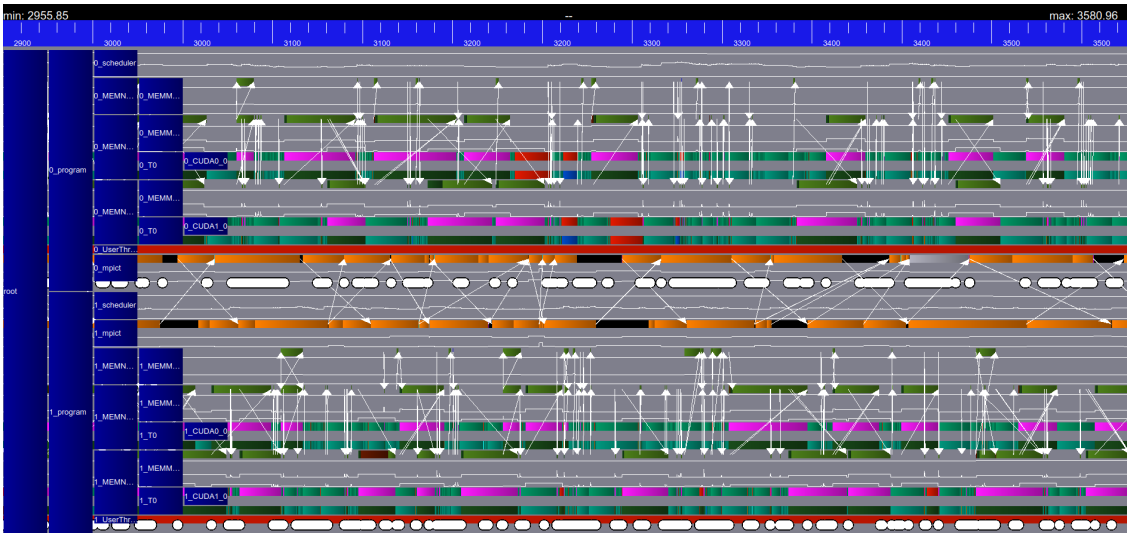


Figure 5.5: Execution trace of the D3Q27 scheme on 2 nodes, each with 2 GPUs. The time frame has been chosen arbitrarily and is 500 ms long. Red sections indicate worker idle time, orange sections show MPI communications, green sections represent kernel execution, and purple sections highlight intra-node communications (here, inter-GPU data transfers). White arrows illustrate data transfer trajectories.

For the moment being, we do not have a satisfactory solution to this issue. A possible solution would be to create or improve a scheduler that would discourage the movement of large data in such a situation. Here, we have a case where it is usually better to keep a GPU in starvation (idle) than to steal a subgrid from another GPU. The best we have been able to do is to quantify the exact number of subgrid "swaps" and tune the *Heteroprio* scheduler in order to minimize them. Thanks to this tuning, we were able to conclude that the default locality strategy of Heteroprio is actually the best one in that regard. It should be noted that subgrids could be affected *a priori* to a GPU, but this would incur a less dynamic execution, which is not the direction we want to take.

On the other hand, the inter-node MPI communications are a bottleneck that appears in both configurations. This was unexpected, as the subgrid system was designed to minimize the amount of data exchanged between nodes. By analyzing the exact number of initiated communications, we can quantify the amount of data exchanged. Over the execution, a total amount of 558 MB of data was exchanged between the nodes (with both configurations, as the communications are identical). For executions of more than 4 seconds, this equates to a bandwidth of 139.5 MB/s. This is too low for the 10 Gbit/s Ethernet network. A latency of approximately 4 seconds $\div 744$ calls = 5.4 ms per call would also appear to be too high.

Running a minimalist MPI benchmark on the same nodes revealed that the observed latency is indeed around 5 ms. Hence, the problem is likely due to a configuration issue. For our purposes, this latency is acceptable, as we are able to achieve near-optimal computation/communication overlap with P100 GPUs. However, this issue will need to be addressed in future work to ensure the implementation properly scales to larger systems and more powerful GPUs.

In the two following experiments, use the same subgrid configuration, but distribute the MPI ranks differently. In this experiment, the MPI ranks were distributed across the Y-axis: low ranks are on the first node and high ranks on the second node. With aditional tuning, we realized that performance could be slightly improved by distributing the ranks across the X-axis. This leads to each rank being responsible for a "slice" of the subgrids. The possible reason for this improvement is that the dependency releases are more evenly distributed across the subgrids, leading to a less impactful scheduling decision. For instance, with 2 nodes, if the ranks are distributed along the Y-axis, which has 16 subgrids, only the 1^{st}, 8^{th}, 9^{th}, and 16^{th} subgrids will include useful computations for the inter-rank synchronization, and the scheduler might not be able to make the best decision. On the other hand, if the ranks are distributed along the X-axis, all the ranks can potentially be involved in the synchronization, leading to a more balanced dependency structure. In the following experiments, we will show the performance we obtain using the X-axis distribution.

Strong scaling

To evaluate the capacity of the implementation to parallelize the simulation, we perform a strong scaling experiment. Here, we limit the number of used GPU per node to 1, to avoid the intra-node communication bottleneck. We use a constant $192 \times 768 \times 192$ (total) grid size, which would require 2.916 GB of memory if it were stored as a single array. This grid is divided into $N \times 16 \times 1$ subgrids, where N is the number of used nodes. Accounting for the fact that we use two subgrids (one for the reads and one for the writes), the ghost cells, and slices for the synchronization, the total theoretical memory usage ranges from 6.179 GB to 6.426 GB depending on the number of nodes, which is well within the memory capacity of a single P100 GPU. To ensure a substantial workload, we set the number of iterations to 48.

Figure 5.6 shows the execution times of the simulation for different numbers of nodes. The colored parts of the bars represent the cumulated execution time of the kernels, while the white parts represent the rest: either idle time or StarPU overhead (e.g., data transfers, scheduling). The presented results are the average of 32 runs. Since the subgrid partitioning is fine-tuned for 4 nodes, the execution times for 1 and 2 nodes should not be considered optimal. However, the busy time (i.e., the time spent executing the kernels) lets us extrapolate potential performance for the 1 and 2 node configurations.

The results show a decreasing trend in the busy time with the number of nodes, which is a positive sign of good scalability. It is reasonable to expect that with further tuning, the 1 and 2 node configurations would exhibit lower idle times (the overhead typically increases with the number of nodes). We can, hence, focus our analysis on the busy time. The speedup on busy time observed between the 1 and 4 node configurations is approximately 2.45, which is less than what we would expect in an ideal scenario (close to 4). One of the reasons for this discrepancy is the increased work introduced by the synchronization tasks `subgrid_to_slice` and `slice_to_subgrid`. This is due to the increased number of interfaces that need to be synchronized as the number of subgrids increases. In theory, if the synchronization kernels took the same time as in the 1-node configuration, the speedup would be of approximately 3.16. The other reason is the latency of the MPI communication, which becomes a bottleneck when the problem size is small relative to the entire system.

Finally, the 4-node configuration shows an average overhead of StarPU is 62.6%, which is relatively high. An analysis of the traces shows that the execution is mostly bottlenecked by the MPI communication, and in particular the latency of the communication. Considering the chosen implementation, it is not surprising that the efficiency of the execution decreases for small problem sizes. For such problems, this issue could be

Figure 5.6: Strong scaling of the StarPU implementation of the D3Q27 scheme. The total grid size is fixed and the number of nodes is increased.

circumvented by using temporal blocking, which would minimize the number of MPI communications and increase the amount of available tasks (and hence parallelism) at each time step.

To assess the performance in a context where the problem size is large enough to warrant distribution across multiple nodes, we conduct a weak scaling experiment.

Weak scaling

Since we aim to simulate larger problems, the most relevant metric for us is the weak scaling. This time, we make the total grid size increase with the number of nodes. The D3Q27 scheme is bounded by memory accesses, so it is reasonable to define the workload as the number of grid points multiplied by the number of iterations. The configurations are, hence, set so that $N_x \times N_y \times N_z \times \text{it\_num} \div N \approx \text{constant}$, where N is the number of nodes, N_x, N_y, and N_z are the sizes of the global grid along the three axes, and it_num is the number of iterations. Table 5.1 shows the exact configurations used for this experiment. The rest of the parameters are the same as in the strong scaling experiment.

N	N_x	N_y	N_z	it_num	$N_x \times N_y \times N_z \times \text{it\_num}$	$N_x \times N_y \times N_z \times \text{it\_num} \div N$
1	256	1024	256	54	3.69×10^9	3.69×10^9
2	320	1280	320	55	7.21×10^9	3.60×10^9
4	384	1536	384	64	1.45×10^{10}	3.63×10^9

Table 5.1: Configurations used for the weak scaling experiment.

Figure 5.7 presents the total execution times for the simulation across different node counts, in a format similar to previous experiments. The results, averaged from 32 runs, are depicted with colored bars showing the cumulative execution times of the kernels and white portions indicating idle time or StarPU overhead such as data transfers and scheduling. As with earlier experiments, the subgrid partitioning was optimized for 4 nodes, rendering the performance at 1 and 2 nodes suboptimal. The subgrid partitioning is the same as in the previous experipent.

We can see that the kernel execution time is almost constant for the different configurations, indicating

Figure 5.7: Weak scaling of the StarPU implementation of the D3Q27 scheme. The total grid size increases with the number of nodes.

a properly balanced workload. However, the non-kernel execution time decreases with the number of nodes, which is an artifact of the chosen subgrid partitioning. Let us then focus on the execution of the configuration with 4 nodes. We observe an average of 4.12 seconds for the kernel execution time and 0.95 seconds for the idle execution time, which gives an overhead of 23.2%. It is an acceptable overhead for a task-based runtime system, especially considering the fact that no load balancing is performed.

A more objective measure of the performance of this configuration is the theoretical number of memory accesses performed per second, also referred to as the processing speed. It correponds to the minimum number of memory accesses (read or write) required to perform the simulation. This measure, whose formula is provided later in Equation 9.9, lets us compare scenarios with and without subgids, different grid sizes, etc. For the configuration with 4 nodes, we obtain a theoretical number of memory accesses of $588,6$ GB per second.

To assess whether this processing speed is justified in terms of computational throughput, we compare it to the processing speed of a case with no subgrids and a single GPU. We run 32 simulations with the same parameters as the 1-node configuration and obtain a processing speed of $216,8$ GB per second. If we bring this processing speed by node, we obtain 147.2 GB/s/node for the 4-node configuration and 216.8 GB/s/node for the 1-node configuration. Hence, we observe a loss of efficiency of 32.1% when scaling to 4 nodes, which is close to the observed overhead (idle time) of 23.2%.

Overall, we can conclude that the observed performance of our tasked-based LBM solver is satisfactory, even for distributed simulations. It should be noted that even if we had observed a lower performance, the approach would still be relevant for larger simulations, as the memory requirements would prevent the use of a single GPU. In the next section, we will begin a discussion on the use of StarPU for LBM simulations and the obervations we made during the development of the solver.

5.5 Conclusion

In this chapter, we have presented our contribution oriented towards the efficient use of StarPU for parallel stencil computations. We have presented a generic approach for parallelizing stencil-based algorithms. Our work led to the development of a task-based LBM solver that is both versatile and efficient. Contrary to our

PaRSEC implementation, this solver is able to run on multiple nodes and can serve as an anchor for future comparisons. Our implementation is competitive compared to a state-of-the-art implementation and can be executed on heterogeneous architectures.

However, it should be acknowledged that the latest results obtained on a large D3Q27 simulation are not as good as expected. The main reason for this appears to be the latency of the MPI communication. We initially expected this latency to be low enough to allow for an efficient overlap of computation and communication. However, the high processing speed of the GPUs makes the workload vanish too quickly, leading to a high idle time if the pratitioning is not correctly tuned. This issue is exacerbated with more powerful GPUs, as the processing speed increases, rendering the proposed approach inefficient. Our solution to this problem would be to use a dynamic partitioning of the subgrids, which would allow us to adapt the workload to the processing speed of the GPUs. This partitioning would allow to use different approaches (e.g. temporal blocking, compression) on different parts of the domain. Multiple technical challenges took more time to overcome than expected, leaving this solution unexplored at the time of writing. However, future works will focus on this next logical step.

Numerous problems can arise when using task-based runtime systems. Throughout this work, we have encountered several issues that have required us to delve into the inner workings of the runtime system to understand the problem and find a solution. One such issue was the proper tuning of Heteroprio, the scheduler we used in our experiments. As this scheduler works with user-defined priorities, it is crucial to set these priorities correctly to ensure the best performance. At an early stage of the project, it was not clear how to set these priorities, and we had to experiment with different values to find the best configuration. This led to the idea of improving the scheduler by making it infer the priorities automatically. This required a substantial amount of work and is the subject of the next chapter.

Chapter 6

Improving the Heteroprio Scheduler of StarPU

In the preceding chapters, we explored the advantages of employing a task-based runtime system. These systems offer a range of benefits, including enhanced performance through optimized resource utilization and simplified development via high-level interfaces that abstract some complexities associated with distributed computing. However, these interfaces can also mask the complexity of the underlying system, potentially impacting the performance of applications. Task scheduling, the focus of this chapter, is a critical aspect in this context. This chapter is based on a paper we published in the journal PeerJ Computer Science [98].

In StarPU, task scheduling is managed dynamically by the scheduler at runtime. Operating within the Task-Flow model, the scheduler relies on partial graph information for its decisions, as it does not have complete knowledge of the dependency graph at the time of scheduling. The primary role of the scheduler is to determine the next task for execution and the processing unit it should run on, decisions that can significantly influence application performance.

Heteroprio [11], one of the default scheduling strategies in StarPU designed for heterogeneous machines, has shown notable performance improvements in various applications [13, 171] Yet, its effective use requires manual priority assignment to different task types within applications, often necessitating extensive benchmarking or accurate programmer intuition about scheduling needs. This requirement for additional programming effort means Heteroprio does not qualify as a fully automated scheduler. Moreover, its dependence on static priorities limits its adaptability during execution, which is a drawback in scenarios where flexibility is crucial.

This study proposes a method for automatically computing efficient priorities for Heteroprio, with a primary focus on automation. Achieving high performance serves as a secondary objective. We introduce heuristics that provide a fitness score for each task type/processing unit combination, allowing for the deduction of priorities by sorting processing units and task types by descending score. The contributions of this study include:

- describing various heuristics leading to efficient priorities;
- defining a new methodology for automatically configuring the Heteroprio scheduler according to these priorities;
- evaluating our approach across a wide array of graphs using emulated executions;
- validating our concept in StarPU by running existing task-based scientific applications with our new automatic scheduler.

These contributions have resulted in a new version of Heteroprio in StarPU, referred to as AutoHeteroprio, which qualifies as a fully automatic scheduler, unlike the semi-automatic nature of Heteroprio. Furthermore, we demonstrate that using the fully automatic version does not lead to significant slowdowns and may sometimes facilitate speedups.

The paper is organized as follows: Section 6.1 provides the background and prerequisites, defining the task scheduling problem, presenting related works, introducing Heteroprio, and formalizing the problem

targeted in our study. Section 6.2 presents the heuristics and implementation details. Finally, Section 6.3 discusses the performance evaluation of our approach.

6.1 Background

6.1.1 Scheduling problem

The primary goal in the task graph scheduling problem is to minimize the overall program finish time, also known as makespan. This finish time is influenced by the order of task execution and their allocation to a specific processor type [140]. Variations in the finish time objective exist. For instance, some studies focus on reducing the mean finish time (MFT), referred to as the mean time of a system or the mean flow time, which represents the average finish time of all tasks executed [59, 156]. The MFT metric aims to minimize the memory required for storing incomplete tasks. Other research targets improvements in metrics such as energy consumption [308]. Despite these variations, the overall finish time remains the most commonly utilized metric in scheduling for measuring performance, which is why it is the metric adopted for performance measurement in this work.

Related work

The quest for an optimal schedule in heterogeneous computing is recognized as NP-complete [58], prompting researchers to develop various strategies for efficient execution. Scheduling can be categorized as either static, where decisions are made before execution, or dynamic, with decisions made during the execution of applications. The spectrum between these categories includes hybrid approaches that integrate elements of both static and dynamic scheduling [88].

Yu-Kwong Kwok and Ahmad [301] detail a static scheduling approach for distributing workload across fully connected multiprocessors, employing the dynamic critical-path scheduling algorithm that computes a critical path for task sequencing. Despite the potential of static scheduling, the preference has shifted towards dynamic scheduling, especially since static methods may not fully capture complex dependencies representable by a Directed Acyclic Graph (DAG).

Topcuoglu, Hariri, and Wu [271] introduce the Heterogeneous Earliest-Finish-Time (HEFT) and Critical Path on a Processor (CPOP) algorithms. HEFT optimizes task scheduling by minimizing the earliest finish time, whereas CPOP calculates critical paths for each processor, considering communication costs in its scheduling decisions. The requirement of HEFT to analyze the entire task graph introduces overhead, particularly for larger graphs.

Khan [130] proposes the Constrained Earliest Finish Time (CEFT) algorithm, incorporating constrained critical paths (CCPs) that represent windows of ready tasks, offering improvements over HEFT but encountering similar bottlenecks.

Jiang, Shao, and Guo [123] explore Tuple-Based Chemical Reaction Optimization for scheduling, producing results comparable to those of HEFT. Choi et al. [69] propose dynamic scheduling based on historical Estimated-Execution-Time (EET) for each task, aiming to optimize processor allocation, though sometimes deviating to avoid work starvation.

Xu et al. [296] develop an efficient genetic algorithm for heterogeneous scheduling, achieving performance comparable to that of HEFT and CPOP. However, the effectiveness of genetic-based schedulers is often contingent on processing large-sized DAGs or multiple iterations.

Wen, Wang, and O'Boyle [290] focus on calculating relative CPU and GPU speedups for task prioritization, effective in scenarios with minimal data transfers but limited in complex scheduling contexts.

Luo et al. [173] utilize a graph convolutional network and reinforcement learning for scheduling optimization, demonstrating efficiency in simulations but with uncertainties regarding real-time application feasibility.

Comprehensive surveys of classical scheduling strategies are provided by Maurya and Tripathi [181], and Beaumont et al. [32], offering insights into the performance and complexities of various algorithms.

Heteroprio overview

The Heteroprio scheduler has been developed for optimizing the fast multipole method and is implemented in StarPU [49, 11, 20]. StarPU is designed such that the scheduler is a distinct component that a user can change or customize. StarPU schedulers rely on two mechanisms known as push-task and pop-task. The push-task is called when a task becomes ready (i.e. when all its dependencies are satisfied). The workers indirectly provoke this call at the end of the execution of a task, if it does allow a new task to be executed. A worker calls the pop function when it fetches a task. This happens either because it has just finished executing a task or after it has been idle for a certain amount of time. Thus, in StarPU, the behavior of a scheduler can be summarized by its push and pop mechanisms.

Heteroprio uses multiple lists of buckets. Each bucket is a first in, first out (FIFO) queue of tasks. When a task becomes available, it is pushed to a bucket. The target bucket is set by the user when submitting the task. There is typically one bucket per task type but the user can choose to group the tasks as they wish. Besides, each architecture has a priority list that represents the order in which the corresponding workers access the buckets. When a worker becomes available, it iterates over the buckets using the priority list and picks a task from the first non-empty bucket it finds. Therefore, these lists define which tasks are favored by a particular architecture. The user must fill them before the beginning of the parallel execution. Figure 6.1 schematizes how the workers select their tasks in Heteroprio. For the sake of simplicity, the CPU and GPU priorities are mirrored, but this is not necessarily the case: we can apply any permutation to the priority list of a processor type.

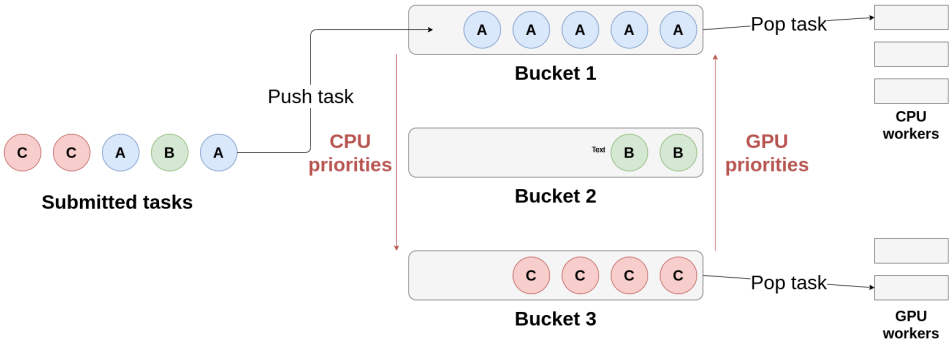


Figure 6.1: Schematic of the principle of Heteroprio. The CPU workers iterate on buckets 1, then 2, and finally 3. The GPU workers iterate the other way around in this example.

We provide a detailed example of an execution with Heteroprio in the appendix 11.1.1. In 2019, an enhancement has been brought to Heteroprio to take into account the data locality [50]. The original version treats all workers of the same type exactly equally, which completely discards memory management and can lead to massive and sometimes avoidable data movement. In the new version of Heteroprio, known as LaHeteroprio, workers select their tasks not only depending on their position in the FIFO list of the buckets but also depending on their memory affinity with the tasks. The affinity is computed thanks to multiple heuristics that the user can choose.

6.1.2 Formalization

General scheduling problem

The scheduling problem is usually defined as follows. Let us consider an application that has a matching DAG referred to as $G = (V, E)$, where V are the v nodes and E are the edges. Each node represents a task, and each edge represents a dependency between two tasks. We define Q as the set of q processors and W as the computation cost matrix. The nodes (tasks) are referred to as v_i, where i can range from 1 to v. The processors are referred to as p_j, where j can range from 1 to q. This computation cost matrix is of size $v \times q$

and $w_{i,j}$ represents the cost of executing task v_i on processor p_j. The cost can be any metric that we seek to minimize. In our case, it is the execution time of a task.

To take data transfers into account, we can add the following definitions. The *Data* matrix represents the required data transfers. $Data_{i,k}$ is the amount of data that needs to be transferred from the processor that executes v_i to the processor that executes v_k. The B matrix defines the transfer rates between processors: $B_{i,k}$ is the transfer rate between p_i and p_k. The L vector represents the communication startup cost of each processor. Hence, the model allows us to define the communication cost of one edge (i, k):

$$c_{i,k} = L_m + \frac{Data_{i,k}}{B_{m,n}} \ , \tag{6.1}$$

where m and n represent, respectively, the chosen processor for v_i and v_k.

To provide a formal definition of the makespan, we introduce the Actual Start Time, and the Actual Finish Time (AST, and AFT). The AFT of a task v_i is defined by $AFT(v_i) = AST(v_i) + w_{i,j}$ (where p_j is the chosen processor for task v_i). The AST of a task v_i is defined as follows:

$$AST(v_i) = \max_{v_j \in pred(v_i)}(AFT(v_j) + c_{j,i}) \ , \tag{6.2}$$

where $pred(v_i)$ is the set of predecessors of v_i. This formula expresses that the task v_i starts as soon as possible, but after all the transfers have been completed. The memory transfers can be ignored by removing the $c_{j,i}$ term.

The *schedule length* (or *makespan*) is defined as the finish time of the last task:

$$makespan = \max_{v_i \in V}(AFT(v_i)) \ . \tag{6.3}$$

We define this makespan as our objective and aim to minimize it. The formalization we provide in this section is general and applicable to most scheduling situations. In the next section, we define additional notations and constraints that relate to the use of Heteroprio.

Heteroprio automatic configuration problem

In this section, we present additional definitions that are needed for the specific Heteroprio scheduling problem. We define a set of b buckets referred to as b_i, where i can range from 1 to b. The concept of bucket is explained in 6.1.1. A solution is defined by a matrix S, where $S_{i,j}$ is the priority of task v_i on processor p_j. When a task is affected to a processor p_j, it has to be the one with the highest priority in the S matrix for p_j over all the tasks that are ready to be executed. We assume that a single bucket is assigned to each type of task. As explained in section 6.1.1, this is not necessarily the case. The number of task types is expected to be significantly smaller than the total number of tasks. Thanks to this, our algorithms have complexity tied to q or b (rather than v) and run fast in practice. This can be illustrated by comparing the possible Heteroprio schedules against all the possible schedules. Consider a graph of 32 tasks with no edges (no dependencies), two different types (A and B), and one processor. As only the execution order of the tasks can change, there are $\binom{32}{1} = 32! \approx 2.63 \cdot 10^{35}$ possible schedules. The scheduling decisions that Heteroprio can take depend on the matrix S, which has only 2 possible configurations in this case: one where A has the highest priority and one where it is B. In every situation, Heteroprio has always exactly $(b!)^q$ possible schedules, where b is the number of different task types, which is assumed to equal the number of buckets. As Heteroprio is designed to handle two processor types, we can simplify some notations. If $arch$ refers to the CPU, $\overline{arch}$ refers to the GPU and vice versa. It should be noted, however, that the heuristics have been generalized to more than 2 processing unit types. Additionally, w_i^{arch} refers to the estimated cost of executing v_i on processor $arch$. Finally, the presented model does not take into account memory transfers, as they are only to a small degree taken into account in Heteroprio.

6.2 Heuristics for automatic configuration

In this section, we first detail the metrics that we use as a basis for our heuristics (section 6.2.1). The heuristics are described in a second step in section 6.2.2.

6.2.1 Relevant metrics

We recall that we do not try to obtain priorities for each task but for each type of task. Consequently, when the number of predecessors, the number of successors, or the execution time are required, the average of all tasks of the same type is used. We also emphasize that these metrics are not the heuristics, but rather the values that are fed to the heuristics. These only aim at giving a quantitative input to the heuristics.

CPU-GPU execution time difference. The CPU-GPU execution time difference can be expressed either as a relative or an absolute difference.

We use the following notations when referring to these metrics:

$$diff_{arch}(v_i) = \overline{w_i^{arch}} - w_i^{arch} \ , \tag{6.4}$$

$$rel\_diff_{arch}(v_i) = \frac{\overline{w_i^{arch}}}{w_i^{arch}} \ , \tag{6.5}$$

where w_i^{arch} is the cost of v_i on $arch$.

The idea of using these metrics is to be able to favor the most efficient architecture. Although the two metrics aim at measuring the same effect, they are not equivalent, as explained in the following example.

Task / Worker	A	B
CPU	100s	1s
GPU	130s	10s
Relative difference (w_i^{GPU}/w_i^{CPU})	$\times$1.3	$\times$10
Absolute difference ($w_i^{GPU} - w_i^{CPU}$)	30s	9s

Table 6.1: Example of relative and absolute costs for tasks of two types and two types of processors.

Let us consider the costs of two tasks on two architectures of Table 6.1. The question is which task type should a CPU worker favor. Here, we consider that both types of processors can execute tasks of types A and B. The relative difference would suggest executing B is a better choice, as its relative difference is higher (the CPU is 10 times faster). However, the absolute difference would suggest that A is a better choice, as it saves 30 seconds instead of 9 seconds.

The absolute and relative differences can, therefore, induce different scheduling choices.

Normalized out-degree (NOD). The normalized out-degree formula [166] is given by:

$$NOD(v_i) = \sum_{v_j \in succ(v_i)} \frac{1}{ID(v_j))} \ , \tag{6.6}$$

where $ID(v_j))$ is the inner degree of task v_j (i.e., its number of predecessors). This metric gives an indication about how many tasks can be expected to be released. In this view, it would mean that releasing $\frac{1}{ID(v_j))}$ of a task v_j is as if it is partially released, at a proportion of $\frac{1}{ID(v_j))}$. For example, releasing 2 tasks at a "ratio" of $\frac{1}{2}$ can be viewed as being equivalent to releasing 10 tasks at a ratio of $\frac{1}{10}$. This obscures the combinatorial nature of task-based execution but is a useful tool for guiding heuristics.

However, the NOD does not take into account the type of the tasks that will be released, which is critical in some cases. For example, in a case where we lack GPU jobs (starvation), the released GPU work is more beneficial than the released CPU work.

Normalized released time (NRT). We introduce the normalized released time (NRT). This metric is derived from the NOD and given by:

$$NRT_{arch}(v_i) = \sum_{v_j \in succ(v_i)} \frac{P_{exec}(v_j, arch) \cdot w_j^{arch}}{ID(v_j))} \ , \tag{6.7}$$

where $P_{exec}(v_j, arch)$ is the probability that v_j is executed on architecture $arch$. This probability is not known during an execution. We instead measure the processor execution proportion of each task type during the execution and use this proportion as an approximation of the probability in our formula.

This formula is more refined than the first NOD formula for two reasons. Firstly, it takes into account the cost of the potential released successors. It is presumably better to release N tasks with a cost of 10 seconds, than N tasks of 1 second because it may release a higher workload. Secondly, CPU and GPU execution times are differentiated. This difference is crucial in a heterogeneous system. Having an NRT formula for both CPU and GPU gives information about where the released work is likely to be executed.

Useful released time (URT). We extend the normalized released time to define the useful released time given by:

$$URT(v_i) = NRT_{CPU}(v_i) \cdot IDLE(CPU) + NRT_{GPU}(v_i) \cdot IDLE(GPU) \ , \tag{6.8}$$

where $IDLE(arch)$ is the idle proportion of $arch$ workers over all the execution. The URT represents how much useful time will be released after a task has finished its execution. The *useful time* is defined as the amount of released work that could help feeding the starving processors. This *useful released time* is estimated by scaling the released work (NRT) of each architecture to the idle proportion of the corresponding architecture. It is implied that the idle proportion is a relevant way of quantifying how much a processor is starving.

6.2.2 Heuristics for task prioritizing

In this section, we present six heuristics: PRWS, PURWS, offset model, softplus model, interpolation model, and NOD-time combination.[1].

Parallel released work per second. In a typical scenario, tasks with high NOD scores should be encouraged to be executed as soon as possible, since they tend to release new tasks in the long run. In both theoretical and practical scenarios, however, using NOD alone as a score does not produce efficient priorities. Indeed, a task can have numerous successors (high NOD) but of low cost. If the costs of the successors are low, the newly released workload will also be low.

To take this effect into account, we introduce a new variable that is designed to give information about the quality of the released tasks. The idea is to keep a high degree of parallelism. This variable is the sum of the execution times of the successors of a task on their best architecture. With this variable we create the formula for the PRWS heuristic:

$$PRWS_{arch}(v_i) = \frac{NOD(v_i)}{w_i^{arch}} \cdot \Big(\sum_{v_j \in succ(v_i)} \min_{arch \in Q} (w_j^{arch}) \Big) + diff_{arch}(v_i) \tag{6.9}$$

Dividing by the cost of the task lets us measure the "releasing speed" (the released work comes at the cost of executing v_i). Adding $diff_{arch}(v_i)$ to the sum helps favoring the best architecture. To improve the work balance between the CPUs and the GPUs, the URT metric can be used instead of the NOD. The Equation 6.9 becomes:

$$PURWS_{arch}(v_i) = \frac{URT(v_i)}{w_i^{arch}} \cdot \Big(\sum_{v_j \in succ(v_i)} \min_{arch \in Q} (w_j^{arch}) \Big) + diff_{arch}(v_i) \ . \tag{6.10}$$

Offset model. The offset model has a score that is defined by the following formula:

$$offset\_model_{arch}(v_i) = (URT(v_i) + \alpha) \cdot (diff_{arch}(v_i) + \beta) \ . \tag{6.11}$$

In this model, the score is computed by multiplying $URT(v_i)$ and $diff_{arch}(v_i)$. α and β are two hyperparameters that control the displacement for each of the two values. For example, if $\alpha = 0$ and $\beta = 0$, then tasks that have a URT of 0 and those that have a $diff$ of 0 would have the same score (0), implying that

[1] Other heuristics are presented in a research report [95]

they are equivalent in terms of criticality. The default values for α and β are 1.3 and 1. This model has the downside of requiring two hyperparameters. Moreover, it is unable to distinguish between tasks when their $diff$ equals $-\beta$, even if their URT are different.

Softplus model. The softplus model is given by the formula:

$$softplus\_model_{arch}(v_i) = (1 + URT(v_i)) \cdot ln(1 + e^{diff_{arch}(v_i)}) \tag{6.12}$$

The idea of this model is comparable to that of the offset model but uses the $softplus$ function ($softplus(x) = ln(1 + e^x)$). In contrary to the offset model, we multiply by $softplus(diff_{arch}(v_i))$ rather than by $diff_{arch}(v_i)$ directly. The $softplus$ mostly changes the behavior of the heuristic when the $diff$ is negative or around zero. This tends to negate the impact of $diff$ when it tends towards zero.

Interpolation model. The interpolation model combines the two previous models. When the URT approaches zero, it tends towards the offset model. It behaves more like the softplus model as the URT grows. It is given by:

$$\begin{aligned} &interpolation\_model_{arch}(v_i) = \\ &rpg(URT(v_i)) \cdot (1 + URT(v_i)) \cdot (1 + arch(v_i)) \\ &+(1 - rpg(URT(v_i))) \cdot ((-log(1 + exp(-archDiff)))) \ , \end{aligned} \tag{6.13}$$

where the interpolation is defined by the rpg function as follows:

$$rpg(x) = \begin{cases} 1 & \text{if } x \geq 1 \\ \sqrt{x} \cdot \sqrt{2 - x} & \text{otherwise} \end{cases} \tag{6.14}$$

This model aims at improving the two previous ones. We assume that the offset model gives particularly good priorities when URT is low and conversely for the softplus model. The idea is to perform an interpolation between the two models depending on the URT value and is controlled by the rpg function. $rpg(URT(v_i)) \in [0, 1]$ because the URT is always positive. When $URT(v_i) = 0$, the interpolation model behaves like the offset model (with $\alpha = 1$ and $\beta = 1$). When $URT(v_i) \geq 1$, it behaves like the softplus model, but without the $(1 + URT(v_i))$ term.

NOD-time combination. The NOD-time combination (NTC) heuristic is defined by the following formula:

$$NTC_{arch}(v_i) = diff_{arch} + \alpha \cdot NOD(v_i) \cdot e^{-\beta \cdot max\_rel\_diff^2} \tag{6.15}$$

where

$$max\_rel\_diff = max(rel\_diff_{arch}(v_i), 1/rel\_diff_{arch}(v_i)) \tag{6.16}$$

This equation needs two hyperparameters α and β. This heuristic aims at diminishing the importance of the NOD as the relative cost difference increases. α controls the importance of the NOD, compared to that of the $diff$, while β controls the range in which the NOD is taken into account. If $rel\_diff_{arch}(v_i)$ is too high, the exponential is negated and the score equals $diff_{arch}$. The default value of α and β are 0.3 and 0.5.

6.2.3 Notes concerning the implementation in StarPU

Cost normalization. If all the costs of the nodes of a DAG are scaled by a factor α, the heuristics should give the same priorities. This would not be the case if we directly input the raw task costs. We, therefore, choose to normalize the costs of the task types.

Normalizing a set of heterogeneous costs is not straightforward. We propose the following normalization formula:

$$z_{i,j} = v \cdot \frac{w_{i,j}}{\sum_{0 \leq i < v} \min\limits_{0 \leq j < q} (w_{i,j})} \ , \tag{6.17}$$

where $z_{i,j}$ is the normalized cost.

This formula normalizes the costs so that the average cost of a task on its best architecture equals 1. This method relies on the assumption that tasks are usually executed on their best architecture. This assumption, however, is disputable in some scenarios.

Execution time prediction. The heuristics presented in this study rely on the execution times of the tasks. We consider that every task of a certain type has the same execution time. In practice, however, tasks of the same type can have radically different costs. Since the tasks have not been computed at the time they are pushed in the scheduler, we need to estimate their duration in real-time. We choose to approximate the cost of a task group by taking the average effective execution time of previous tasks of the corresponding type. If a task has never been executed on an architecture, we have no precise estimation of its execution time. We, therefore, implement two behaviors:

- the estimation is set to a default value of 100000 seconds (default behavior);
- if an estimation exists on another processor, we take the fastest estimation, else we take 100000 seconds.

This solution is imperfect, in particular when their execution times are dispersed. In this case, the scores given by the heuristics may translate into inefficient priorities. We assume that in most cases, taking the average execution time is sufficient for generating reliable priorities.

Task-graph. In this model, we consider that the applications are converted into a task graph which is a DAG. Most memory access types (READ, WRITE, READ-WRITE) can be translated in a dependency in a DAG. Some accesses, however, cannot be transcribed in terms of direct static dependencies. For example, StarPU has a memory access type known as STARPU_COMMUTE which is used when several contiguous (READ-)WRITE accesses can be performed in any order but not at the same time. A simplistic use case of this would be when the tasks increment a shared counter. This access mode has been used in mathematical applications, e.g., for an optimized discontinuous Galerkin solver or the fast multipole method [8, 52]. For this type of access, we can reason in terms of availability rather than dependency: 1) if no task is commuting on the data, any task can take the memory node, and 2) if one task is commuting, the memory node is blocked. Thus, the heuristics cannot use all the information they have on applications that use these relatively uncommon memory access types. In practice, in the presented heuristics, these accesses are treated as write accesses.

Heteroprio automatic configuration. In our implementation, we update the priority lists in the scheduler only when a task is pushed in the scheduler. More precisely, the priorities are updated the first time a task is pushed (the first time the scheduler discovers a new type of task), and then every n^{th} pushed task. This choice avoids updating the priorities too often and should, therefore, help reduce the scheduling overhead.

6.3 Performance study

6.3.1 Evaluation based on emulated executions

We create a simple simulator for running a fake StarPU execution. As input, it takes the fake DAG of an application, the costs of the tasks, and the priority lists. It then simulates an execution with the Heteroprio scheduler based on our model (see section 6.1.2). As output, it gives the theoretical execution time of the whole fake application. This theoretical execution time does not include data transfers.

It can be viewed as a black box where we input priorities and obtain an execution time as output. We, therefore, choose this tool as a base for elaborating our experimental protocol. This protocol aims at generating a score for a heuristic based on how well it performs in multiple scenarios. It has two purposes. Firstly, it provides a fast way to check how successful a heuristic is. Secondly, it provides an additional argument for our work if the heuristics perform as well in the protocol as in real applications.

Graph generation

To be able to evaluate our heuristics, we generate a dataset of 32 graphs with diversity in the number of task types, the costs of the tasks, and the graph shape. To generate a graph, we generate tasks while filling a pipeline of workers [2]. We affect each task to its best worker. Consequently, at the end of the generation

[2]The DAG generating code is publicly available [51]

process, we know the scheduling that minimizes the makespan and have a lower bound for a hardware configuration that corresponds to the pipeline. We also generate a predecessor matrix P randomly. This predecessor matrix is of size $v \times v$ and $P_{i,j}$. It represents the average number of predecessors of tasks of type i that are of type j. Our graph generation method uses this predecessor matrix as input and adjusts the predecessors of the newly created tasks so that they match the values of the matrix.

The generator needs the following parameters:

- a seed for the generation of random numbers
- the final amount of tasks
- a list of task types, with their associated CPU and GPU costs and their expected proportion in the pool of tasks
- a number of CPU and GPU workers
- a predecessor matrix

Table 6.2 gives details about the generated datasets.

data index	CPU number	GPU number	CPU/GPU	close CPU-GPU task proportion	far CPU-GPU task proportion	task with numerous predecessors proportion	average predecessor number	max predecessor number	task without successor proportion	task with numerous successors proportion	max successor number	average CPU-GPU diff (relative)
0	4	14	0.286	0.549	0.336	0.502	4.038	7	0.423	0.243	41	0.580
1	13	8	1.625	0.377	0.623	0.000	2.101	3	0.467	0.084	105	0.905
2	11	12	0.917	0.687	0.135	0.000	2.526	3	0.781	0.033	540	0.855
3	2	7	0.286	0.191	0.302	0.211	2.529	4	0.388	0.104	142	1.089
4	13	15	0.867	0.508	0.233	0.007	1.289	5	0.575	0.057	102	1.605
5	9	7	1.286	0.276	0.573	0.141	2.301	4	0.360	0.127	546	1.154
6	13	3	4.333	0.314	0.026	0.000	2.396	3	0.339	0.118	62	0.715
7	11	1	11.000	0.226	0.531	0.000	1.491	3	0.496	0.062	307	1.036
8	9	12	0.750	0.418	0.582	0.000	1.675	3	0.255	0.070	22	0.187
9	12	1	12.000	0.405	0.043	0.367	2.927	4	0.176	0.180	57	0.388
10	2	9	0.222	0.167	0.777	0.000	0.995	1	0.301	0.005	7	3.791
11	13	10	1.300	0.232	0.529	0.000	1.384	3	0.507	0.083	24	1.720
12	4	6	0.667	0.286	0.530	0.000	1.325	3	0.658	0.060	103	1.179
13	4	11	0.364	0.018	0.497	0.000	1.462	3	0.563	0.031	72	1.484
14	8	1	8.000	0.498	0.468	0.000	1.850	2	0.459	0.088	52	1.756
15	3	8	0.375	0.112	0.888	0.000	2.686	3	0.570	0.072	111	4.153
16	15	3	5.000	0.294	0.126	0.000	1.347	2	0.466	0.094	20	1.243
17	10	1	10.000	0.452	0.514	0.000	2.258	3	0.766	0.064	548	0.228
18	7	3	2.333	0.160	0.565	0.139	1.679	4	0.432	0.094	54	1.793
19	9	14	0.643	0.269	0.725	0.000	1.817	3	0.334	0.084	108	2.238
20	8	11	0.727	0.386	0.392	0.000	1.859	3	0.294	0.093	25	0.850
21	8	8	1.000	0.527	0.324	0.323	2.655	5	0.386	0.083	439	0.917
22	15	9	1.667	0.350	0.650	0.126	2.268	4	0.281	0.107	147	2.050
23	14	4	3.500	0.008	0.973	0.000	1.288	3	0.228	0.022	116	12.786
24	1	2	0.500	0.115	0.175	0.133	1.934	5	0.327	0.115	18	0.881
25	9	11	0.818	0.278	0.278	0.000	2.030	3	0.275	0.119	13	0.626
26	4	14	0.286	0.299	0.512	0.166	1.884	4	0.372	0.111	34	0.771
27	15	1	15.000	0.453	0.417	0.000	1.551	2	0.685	0.090	55	0.253
28	9	3	3.000	0.635	0.266	0.099	1.474	5	0.477	0.066	131	0.187
29	15	8	1.875	0.288	0.539	0.396	3.558	6	0.186	0.264	50	1.534
30	10	10	1.000	0.612	0.000	0.169	2.552	7	0.368	0.197	28	0.434
31	12	13	0.923	0.395	0.605	0.000	1.516	3	0.482	0.094	17	2.245

Table 6.2: Details of the randomly-generated graph dataset. CPU-GPU close tasks are the tasks that have less than +20% between the two processor costs and conversely for far CPU-GPU costs. Here, "numerous" means 5 or more.

Protocol

We run fake executions on the 32 generated graphs for each heuristic. We compare the obtained makespans to the makespans obtained with control priorities and provide a slowdown for each heuristic. The control priorities are obtained with an iterative optimization algorithm. The algorithm begins with random CPU and GPU priorities. It then performs multiple iterations, alternating between CPU and GPU. At every iteration, all the possible priority permutations for the current architecture (CPU/GPU) are tested and the fastest permutation is kept. In the case of a tie, the fastest priorities are chosen randomly among the equally-ranked bests. These control priorities aim at giving anchor points for computing the slowdowns of the heuristics.

Results

Test case \ Heuristic	Offset	Softplus	Interpolation	PURWS	PRWS	NTC
0	1.119	1.120	1.119	1.285	1.285	1.135
1	1.001	1.049	1.001	1.062	1.062	1.001
2	1.045	1.031	1.063	1.170	1.264	1.209
3	1.096	1.154	1.032	1.178	1.208	1.241
4	1.117	1.104	1.138	1.159	1.119	1.110
5	1.052	1.048	1.007	1.194	1.165	1.062
6	1.318	1.280	1.361	1.182	1.061	1.452
7	1.436	1.536	1.530	1.752	1.056	1.019
8	1.144	1.078	1.076	1.046	1.017	1.025
9	1.029	1.042	1.029	1.017	1.017	1.023
10	1.329	1.329	1.329	1.000	1.000	1.048
11	1.010	1.010	1.010	1.128	1.160	1.010
12	0.992	1.009	1.035	1.038	1.047	0.990
13	1.026	1.126	1.069	1.183	1.183	1.126
14	1.014	1.003	1.014	1.034	1.034	1.014
15	1.010	1.010	1.010	1.321	1.281	1.283
16	1.020	1.297	1.297	1.020	1.020	1.020
17	1.052	1.019	1.058	1.050	1.050	1.026
18	1.193	1.054	1.040	1.366	1.304	1.193
19	1.163	1.487	1.163	1.224	1.279	1.354
20	1.000	1.000	1.268	1.254	1.254	1.163
21	1.156	1.251	1.156	1.474	1.351	1.158
22	1.134	1.118	1.134	1.143	1.197	1.065
23	0.999	1.126	0.999	1.002	1.126	1.154
24	1.007	1.055	1.020	1.191	1.154	1.043
25	1.042	1.068	1.042	1.037	1.037	1.055
26	1.124	1.063	1.076	1.075	1.084	1.070
27	1.028	1.028	1.013	1.002	1.002	1.019
28	1.034	1.018	1.114	1.065	1.077	1.034
29	1.092	1.082	1.083	1.159	1.159	1.009
30	1.014	1.014	1.014	1.075	1.051	0.992
31	1.106	1.106	1.106	1.118	1.118	1.118

Table 6.3: Slowdown obtained on emulated executions by comparing the estimated lower-bound against Heteroprio-based executions using the different heuristics. The lower bound is estimated with an iterative optimization algorithm.

The results of our emulated simulations are available in Table 6.3. We show the slowdown of the 6 heuristics we present in this paper (compared to the control priorities): PRWS, PURWS, offset model, softplus model, and interpolation model. We see that some slowdowns are lower than 1. This means that some heuristics find better priorities than the control priorities, which have been found with an iterative optimization algorithm. In general, we observe that the slowdown ranges between +0% and +20%. In most test cases, the best of the 6 heuristics usually has a slowdown of less than +10%. There are some exceptions such as cases number 0, 4, 19, 21, 22, or 31. From these simulated executions, we expect the choice of heuristic to have a significant impact.

6.3.2 Evaluation on real applications

Configuration

Hardware. We carry out our experiments on three configurations. Each one has a different GPU model. In this paper, we use the model name of the GPUs for referring to the associated configuration:

- **K40M** is composed of 2 Dodeca-cores Haswell Intel Xeon E5-2680 v3 2.5 GHz, and 4 K40m GPUs (4.29 TeraFLOPS per GPU). We use 7 CUDA streams per GPU;
- **P100** is composed of 2 Hexadeca-core Broadwell Intel Xeon E5-2683 v4 2.1 GHz, and 2 P100 GPUs (8.07 TeraFLOPS per GPU). We use 16 CUDA streams per GPU;
- **V100** is composed of 2 Hexadeca-core Skylake Intel Xeon Gold 6142 2.6 GHz, and 2 V100 GPUs (14.0 TeraFLOPS per GPU). We use 16 CUDA streams per GPU.

Software. We select four applications that are already parallelized with StarPU to evaluate our scheduler:

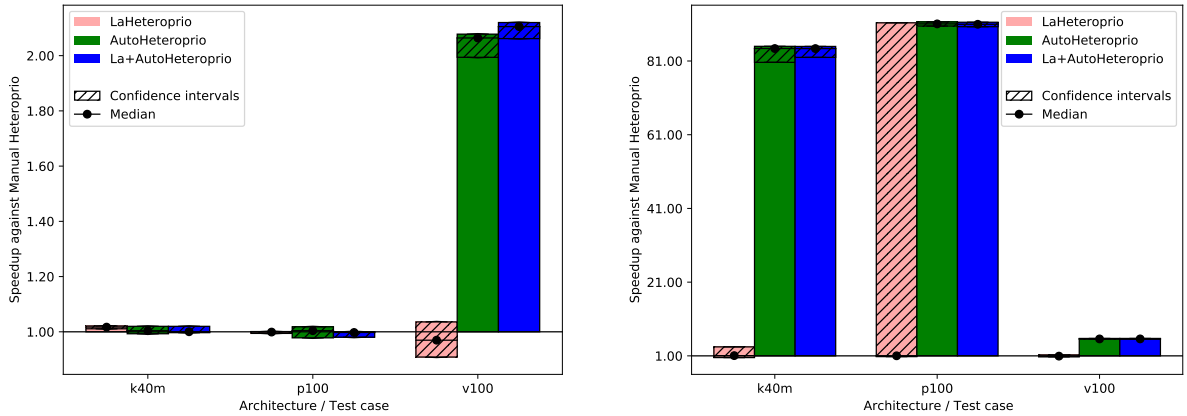
- **ScalFMM** [10] is an application that implements the fast multipole method (FMM). The FMM algorithm computes the n-body interactions between the particles directly and across a tree mapped over the simulation box. We use it with two test-cases based on the *testBlockedRotationCuda* program. The first one runs with the default parameters and 10 million particles. The other one runs with a block size of 2000, a tree height of 7, and 60 million particles;
- **QrMUMPS** computes the QR factorization of sparse matrices [12] using the multifrontal method [91]. When it was extended to heterogeneous architectures in 2016 by Florent LOPEZ [170], Heteroprio was the fastest scheduler of StarPU for this application. In our experiment, we choose to measure the factorization time of the TF16 matrix [266], from the JGD_Forest dataset;
- **Chameleon** is a library for dense linear algebra operations that supports heterogeneous architectures [7]. We select the same operations as the ones considered by the authors for the benchmarks presented in their user guide: a Matrix Multiplication (GEMM), a QR factorization (QRM), and a Cholesky factorization (POTRF). We use a block size of 1600 and a matrix size of 40000 for the Matrix Multiplication and the QR factorization. For the Cholesky factorization, we use a matrix size of 50000;
- **PaStiX** is a library which provides a high performance solver for sparse linear systems [113, 142]. We consider two stages of the example program named 'simple': the LU factorization and the solve step. The program generates a Laplacian matrix. We choose a matrix size of 100^3.

For a given set of parameters (scheduler, hardware configuration, etc.), each application is run 32 times. All these applications can be configured to use StarPU and, therefore, the task-based model. The codelets (low-level kernels) are encapsulated into tasks that are submitted to StarPU. The four applications have CPU and CUDA kernels and at least one task that has both a CPU and a CUDA implementation. For the latter hybrid tasks, the scheduler is responsible for making the proper processor type choice. Finally, the tested applications are all written in C, except for QrMUMPS which is written in Fortran. To make the applications usable for our tests, we change parts of them. We update QrMUMPS and ScalFMM so that they use performance models, which are needed by our automatic strategy but also by most schedulers. Additionally, we create new static priorities for the Heteroprio scheduler in Chameleon and PaStiX. The

methodology for setting these priorities is detailed in the appendix 11.1.2. Unless otherwise indicated, the execution times are the median value of the 32 corresponding runs. All schedulers that need a calibration run (which sets up the performance models) use an extra run that is not included in the final results.

Comparison between manual and automatic priorities

In this study, we compare the performance of four versions of the scheduler: Heteroprio, LaHeteroprio, AutoHeteroprio, and LaAutoHeteroprio (AutoHeteroprio with LA enabled). We use Heteroprio as the reference value and provide the speedups of the three other versions. For AutoHeteroprio and LaAutoHeteroprio, we provide the data of the best heuristic, i.e. the heuristic whose average execution time is the lowest. The median execution time of Heteroprio (the reference) is divided by each individual execution time for obtaining speedups. By doing so, we obtain a set of speedups for each case, rather than a single value. This lets us display a median and two limits of a confidence interval. For this confidence interval, we exclude the 5% highest and 5% lowest values.



(a) (b)

Figure 6.2: Speedups of LaHeteroprio, AutoHeteroprio, and LaAutoHeteroprio against Heteroprio in the two ScalFMM test cases. We study two test cases: (a) 10 million particles and (b) 60 million particles. The hatched area represents the interval of confidence of the 32 corresponding runs.

Figure 6.2 shows the results for ScalFMM. In the first test case (10 million particles), all the versions are comparable on the p100 and k40m architecture. In the v100 case, AutoHeteroprio and LaAutoHeteroprio are about 2 times faster than normal Heteroprio. In the second test case (60 million particles), AutoHeteroprio and LaAutoHeteroprio are more than 80 times faster on the p100 and k40m architectures and about 5 times faster on the v100 architecture. LaHeteroprio (respectively, LaAutoHeteroprio) does not show such a high difference to Heteroprio (respectively, AutoHeteroprio) in this scenario. The reason for this is that data transfers are hard to avoid in this application because only two task types have a GPU implementation. Their data must be transferred back to the main memory to be used by tasks on the CPU.

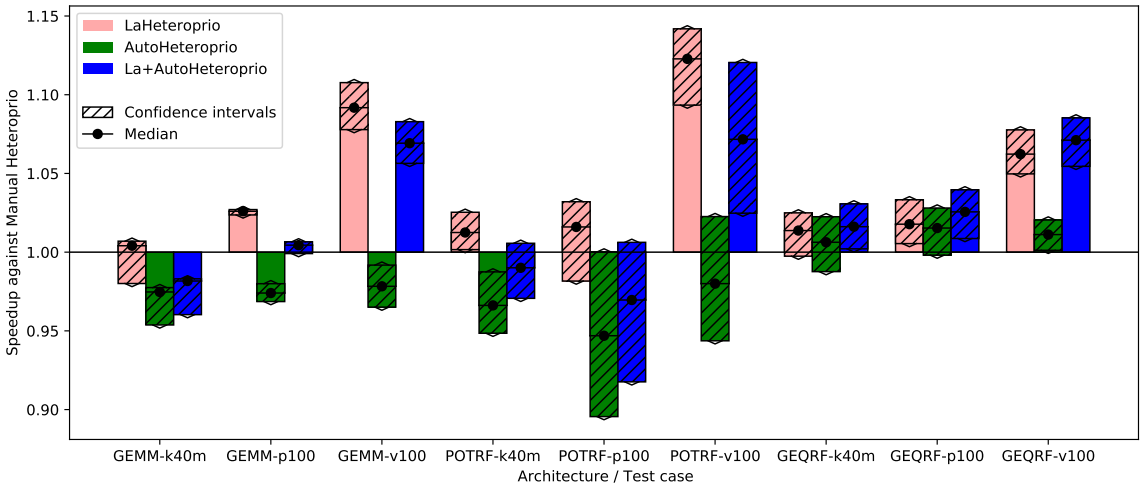
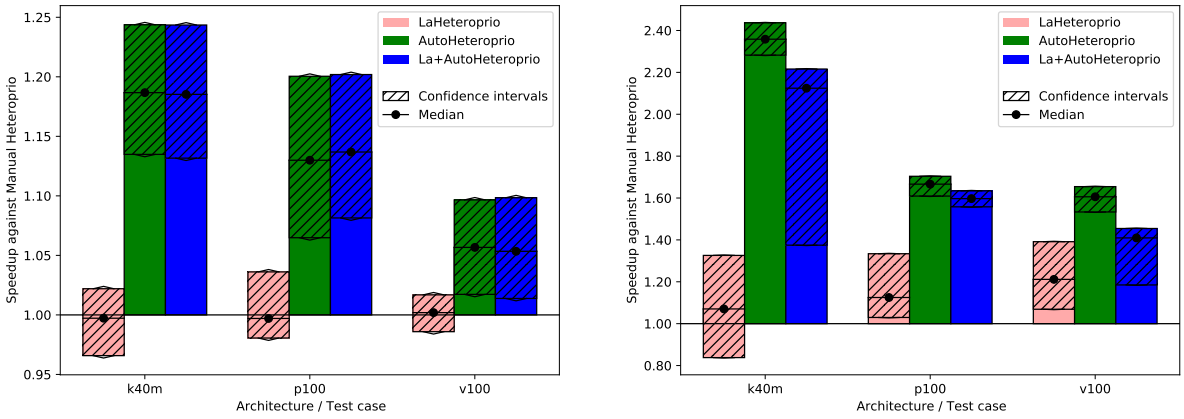


Figure 6.3: Speedups of LaHeteroprio, AutoHeteroprio, and LaAutoHeteroprio against Heteroprio on Chameleon test cases (GEMM, POTRF, and GEQRF). P100, V100 and K40M relate to the hardware configuration. The hatched area represents the interval of confidence of the 32 corresponding runs.

Figure 6.3 shows the results for Chameleon. In this application, automatic priorities are systematically slower than their manual counterparts. Indeed, AutoHeteroprio generally has a speedup of less than 1 and LaAutoHeteroprio is usually worse than LaHeteroprio. Furthermore, LaHeteroprio and LaAutoHeteroprio tend to be faster, which suggests that locality has greater importance in Chameleon than in ScalFMM.

We explain the lack of performance of automatic versions by a lack of precision in the execution time estimations of the tasks. This leads to an inefficient choice of priorities. The execution time estimations of the tasks are biased because AutoHeteroprio averages the execution time of a task type. Yet, in Chameleon, the data size has an important impact on the execution times of the tasks. This breaks our initial premise which is that each task within a bucket has the same execution time.

We provide the results for the QR Factorization from QrMUMPS and on the LU Factorization from PaStiX in Figures 6.4a and 6.4b, respectively. In both cases, AutoHeteroprio shows a significant increase in performance on all configurations. In the QR-MUMPS test, AutoHeteroprio reaches more than +18% speedup. In the LU factorization, it goes past x2.3 speedup on the k40m architecture. It appears that the dynamic change of the priorities at runtime of the automatic Heteroprio is an advantage in both applications (to evaluate these changes, we manually export the priorities during the executions).



(a) (b)

Figure 6.4: Speedups of LaHeteroprio, AutoHeteroprio, and LaAutoHeteroprio against Heteroprio in the Qr factorization (QrMumps) and the LU factorization (PaStiX). (a) Qr factorization (QrMUMPS). (b) LU factorization (PaStiX). The hatched area represents the interval of confidence of the 32 corresponding runs.

Overall, we have multiple observations. It appears that using automatic priorities does not always harm performance. In some cases, it can even increase them. Automatic priorities are only slower in the case of the GEMM and POTRF test cases in Chameleon. In some cases, the speedups of the automatic priorities become particularly high when run on a new architecture (e.g. Figure 6.2). This demonstrates the ability of automatic priories to adapt to the current architecture. Manual priorities, on the other hand, can hardly be efficient on multiple different architectures.

Comparison with other schedulers

In this section, we compare Heteroprio with other schedulers available in StarPU:

- the *Eager* scheduler uses a central task queue from which all workers retrieve tasks concurrently. There is no decision on the task distribution. The worker picks the first task that is compatible with their PU;
- the *LWS* (Locality Work Stealing) scheduler uses one queue per worker. When a task becomes ready, it is stored in the queue of the worker that released it. When the queue of a worker is empty, the worker tries to steal tasks from the queues of other workers;
- the *Random* scheduler randomly assigns the tasks to compatible workers;
- the *DM* (deque model) scheduler uses a HEFT-like strategy. It tries to minimize the makespan by using a look-ahead strategy;
- the *DMDA* (deque model data aware) follows the principle of DM but adds the data transfer costs;
- the *DMDAS* (deque model data aware) acts as the DMDA scheduler but lets the user affect priorities to the tasks. Since this scheduler needs user-defined priorities, we discard DMDAS from the results when the application does not define custom priorities.

For the sake of conciseness, by default we only display the results for the best between Heteroprio (respectively AutoHeteroprio) and LaHeteroprio (respectively LaAutoHeteroprio). When the difference between the LA and the non-LA version is noticeable, we display the 4 versions. For the automatic Heteroprio versions (AutoHeteroprio and LaHeteroprio), we aggregate all the data of every heuristic designed in AutoHeteroprio. Since there are 28 different heuristics in AutoHeteroprio and 32 runs for each one, the data for the automatic configuration consists of $28 \times 32 = 896$ runs, while other the other shown data consist of 32 runs.

Figure 6.5 shows the execution times of the solve step in PaStiX with different schedulers on the p100 configuration (the results for the v100 and k40m configurations are comparable).

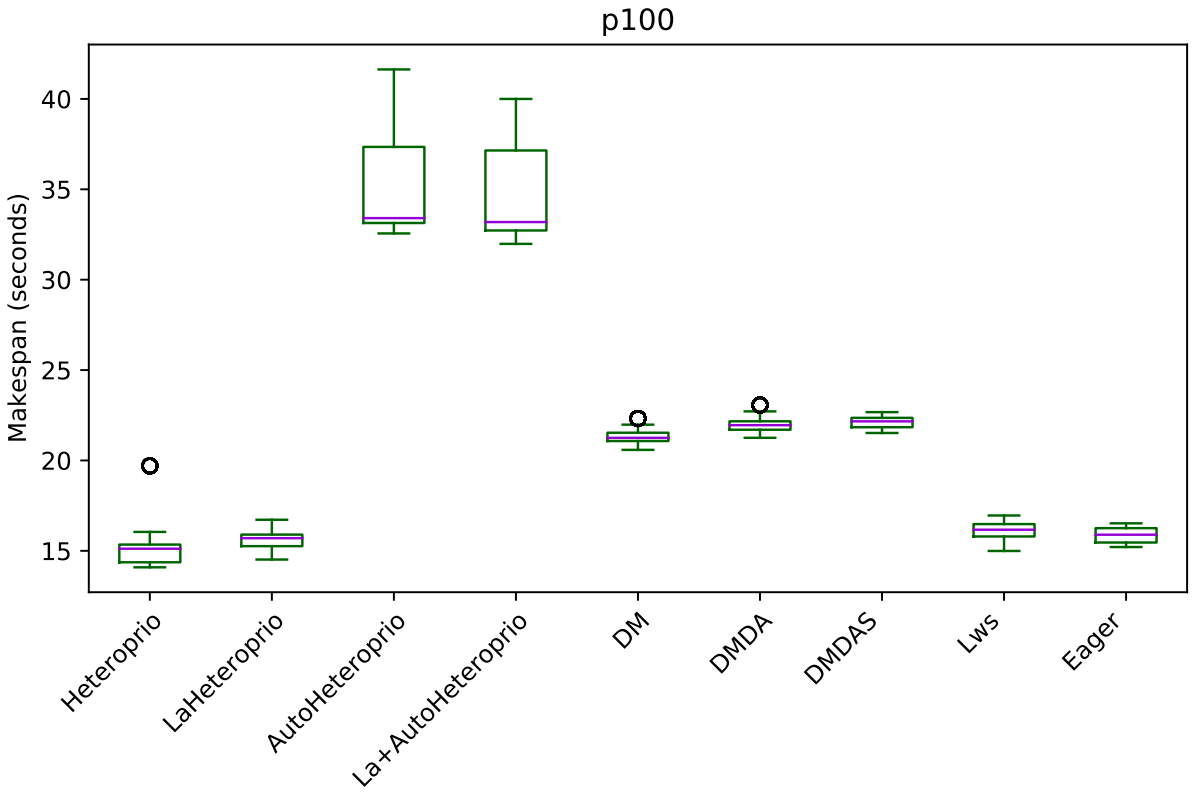


Figure 6.5: Execution times of the PaStiX solve step for different schedulers on the P100 configuration. The boxes show the distribution of the 32 makespans (896 for AutoHeteroprio and LaHeteroprio) for each scheduler.

We can group the schedulers into three performance categories (sorted from slowest to fastest):

- AutoHeteroprio and LaAutoHeteroprio
- DM, DMDA, and DMDAS
- basic Heteroprio, LaHeteroprio, LWS, and Eager

To explain this result, let us explain the task structure of this application. There are only two types of tasks with average execution times of 95 and 120 microseconds. These execution times are relatively short for a runtime system like StarPU. Indeed, the overhead of StarPU is relatively high, as it has been designed to handle large amounts of data. In particular, the use of a scheduler is only relevant when the expected gained time is greater than the overhead of the scheduler. In this test case, it appears that the scheduling decision has less importance than in other applications, as lightweight schedulers tend to perform better. It confirms that Heteroprio and LaHeteroprio have a low overhead. Their overhead is comparable to those of LWS and Eager. This test also points out that AutoHeteroprio and AutoLaHeteroprio have a significant overhead. For these, the overhead is higher than that of DM, DMDA, and DMDAS.

We compare the schedulers for the QrMUMPS test case in Figure 6.6. AutoHeteroprio performs better than manual Heteroprio, which is already better or as good as other schedulers, depending on the configuration.

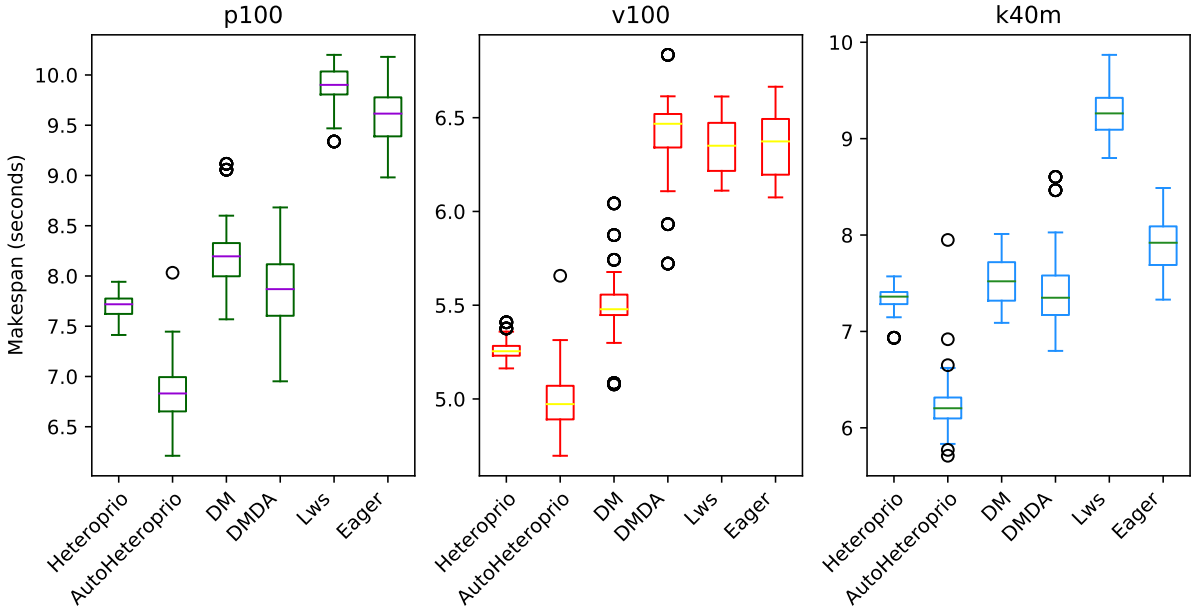


Figure 6.6: Execution times for QrMumps for different schedulers on the three configurations. The boxes show the distribution of the 32 makespans (896 for AutoHeteroprio) for each case.

Figure 6.7 presents the results for the Matrix multiplication in Chameleon, on the k40m and the p100 configurations. The V100 has been left out as the results are similar to the P100 configuration. We observe that AutoHeteroprio is faster and more reliable than schedulers like LWS or random but less efficient than the DM schedulers. The results for the Cholesky factorization that we present in Figure 6.8, are similar. In this configuration, AutoHeteroprio is closer to the performances of DM. Manual Heteroprio performs almost as well as DM.

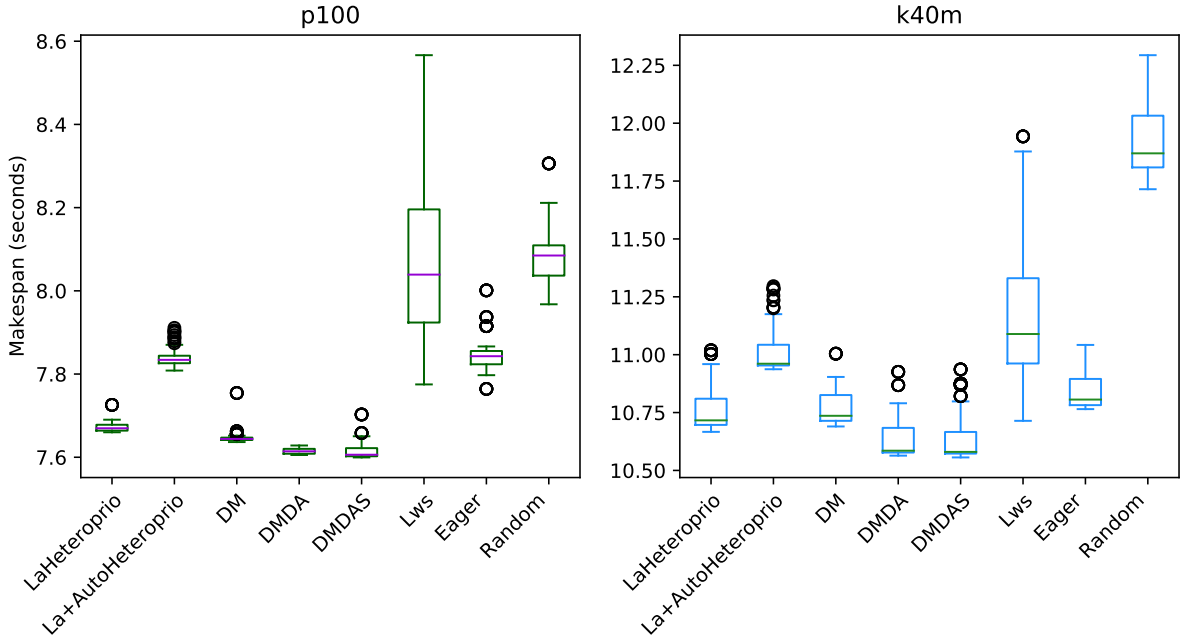


Figure 6.7: Execution times for Chameleon GEMM for different schedulers on two configurations. The boxes show the distribution of the 32 makespans (896 for LaAutoHeteroprio) for each case.

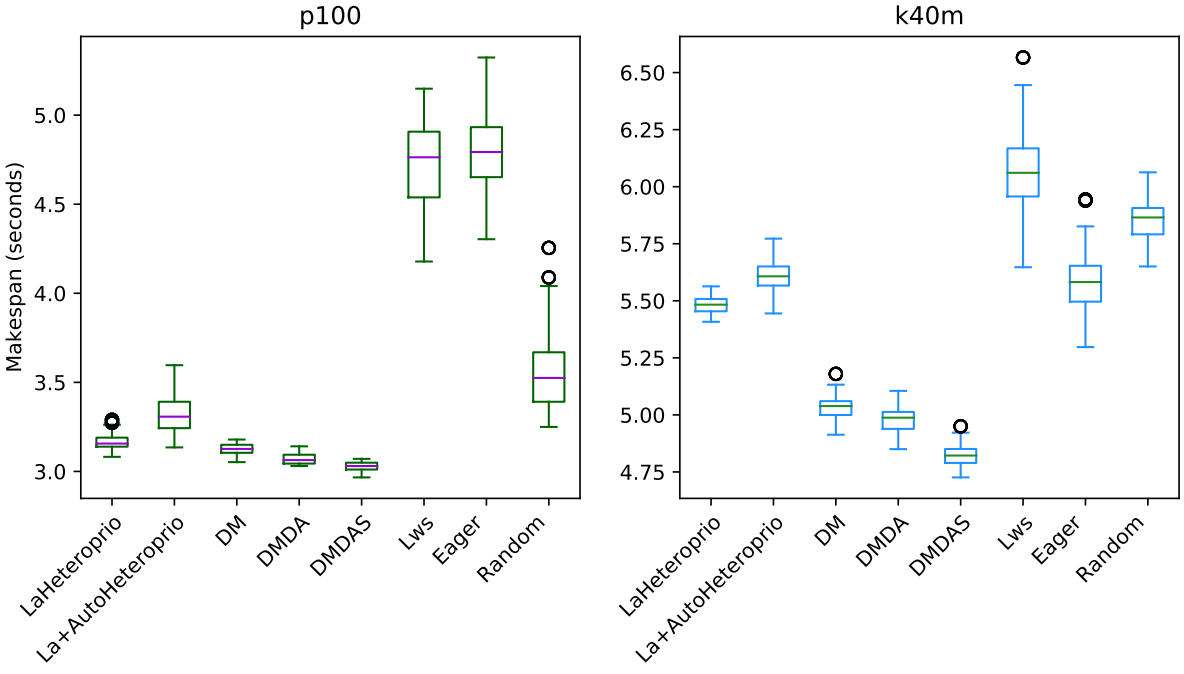


Figure 6.8: Execution times for Chameleon Cholesky factorization for different schedulers on the p100 and the k40m configuration. The boxes show the distribution of the 32 makespans (896 for LaAutoHeteroprio) for each scheduler.

We present the results for the Chameleon QR Factorization in Figure 6.9. In the p100 configuration

(and the v100 configuration which is comparable), both Heteroprio versions perform comparably to the DM scheduler. In the k40m configuration, the performance of both versions is low. Heteroprio only seems to do better than the random scheduler. The Eager scheduler outmatches DM schedulers.

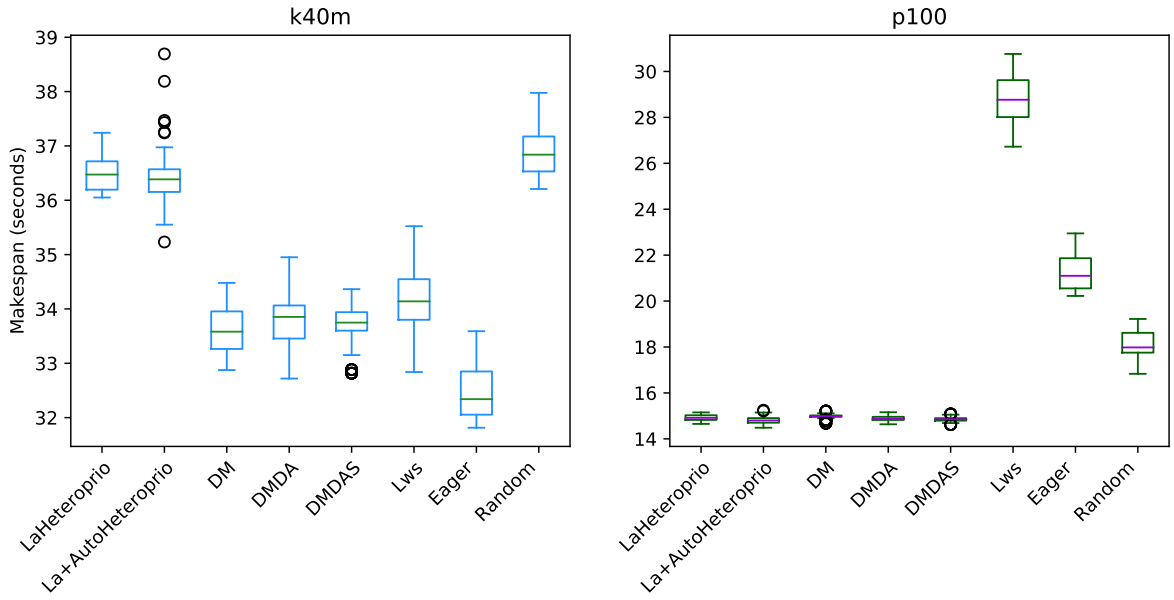


Figure 6.9: Execution times for Chameleon QR factorization for different schedulers on the p100 and the k40m configuration. The boxes show the distribution of the 32 makespans (896 for LaAutoHeteroprio) for each scheduler.

In the case of factorization with PaStiX (Figure 6.10), AutoHeteroprio performs well on the p100 configuration. In contrast, on the k40m configuration, DMDAS, DMDA, and LWS schedulers perform better. With the v100 configuration, the results of AutoHeteroprio are only better than the ones of Heteroprio.

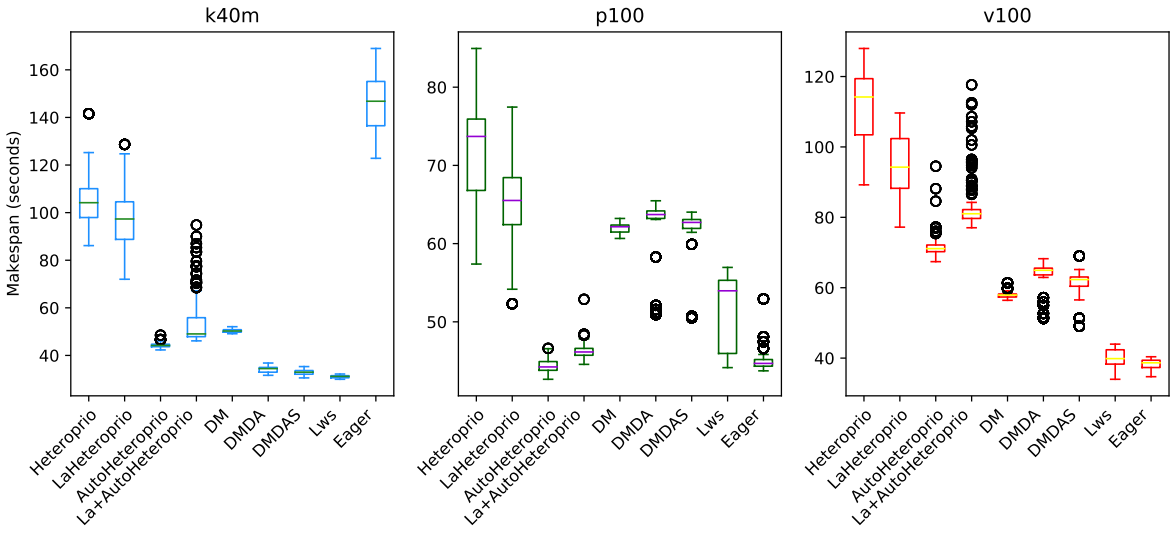


Figure 6.10: Execution times for PaStiX factorization for different schedulers on the three hardware configurations (k40m, p100 and v100). The boxes show the distribution of the 32 makespans (896 for AutoHeteroprio and LaAutoHeteroprio) for each scheduler.

The results of the ScalFMM tests cases are shown in Figures 12.3 and 12.4. These are represented using a logarithmic scale because of the high differences between the execution times of the schedulers. We can see that AutoHeteroprio performs well on this application. It is comparable and sometimes better than schedulers of the DM family. Note that the DM and DMDA schedulers can use more than one calibration run. This presumably explains their uppermost bullets in the figures. AutoHeteroprio only needs one calibration run before achieving its best performance.

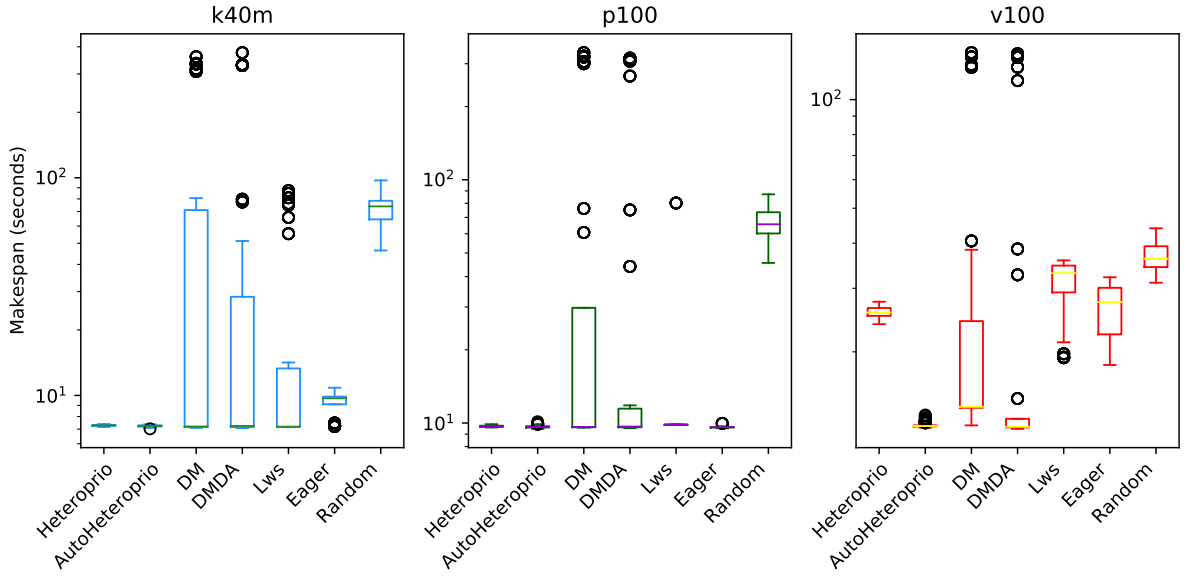


Figure 6.11: Execution times for the first ScalFMM test case on the three hardware configurations (k40m, p100 and v100). The scale of the Y-axis is logarithmic. The boxes show the distribution of the 32 makespans (896 for AutoHeteroprio) for each scheduler.

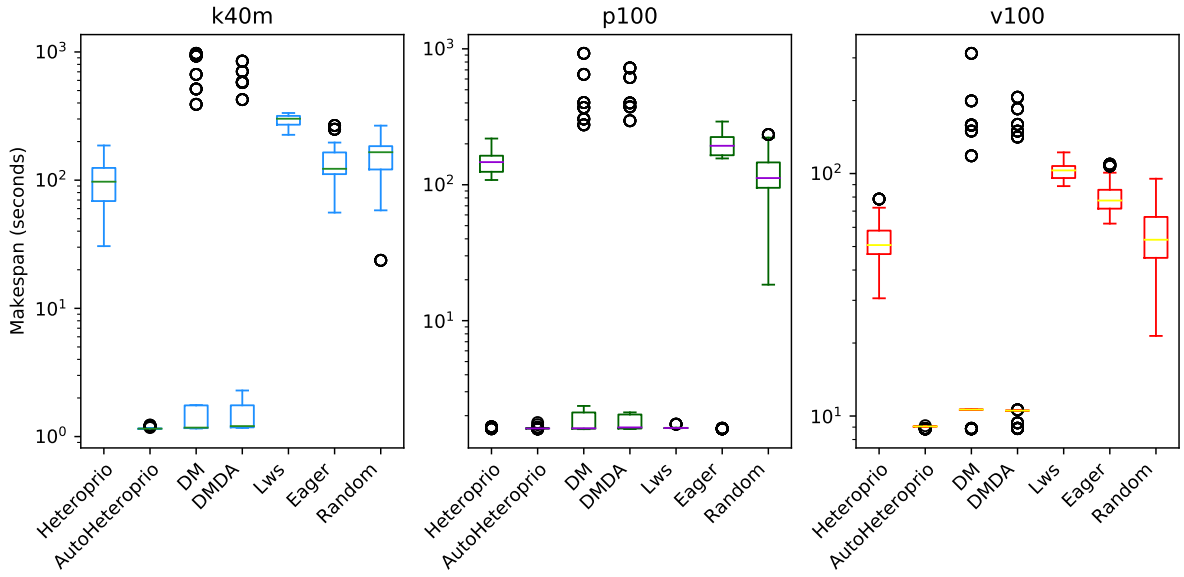


Figure 6.12: Execution times for the second ScalFMM test case on the three hardware configurations (k40m, p100 and v100). The scale of the Y-axis is logarithmic. The boxes show the distribution of the 32 makespans (896 for AutoHeteroprio) for each scheduler.

This second study gives an overview of the performance of different applications with various schedulers in StarPU. With these results we can estimate the impact of the choice of scheduler on the overall execution time and evaluate the competitiveness of Heteroprio with manual or automatic priorities. In general, AutoHeteroprio offers satisfying results compared to its competitors. When it does not, it is usually in cases where the Heteroprio (manual) version is already slow. The only cases where AutoHeteroprio does not achieve acceptable performance when compared to Heteroprio are the Chameleon GEMM and the PaStiX solve step. Moreover, AutoHeteroprio does improve the performance of Heteroprio significantly in other cases such as in QrMumps, PaStiX factorization, and some ScalFMM configurations. Therefore, this study suggests that AutoHeteroprio is a competitive scheduler for a runtime system like StarPU. In addition to this, it is fully automatic, contrary to some of its competitors (Heteroprio, LaHeteroprio, and DMDAS).

Comparison of different heuristics in AutoHeteroprio

In AutoHeteroprio, the priority lists are computed thanks to heuristics. In section 6.3.2, we show the performance of the best heuristic over all the 28 measured executions, while in section 6.3.2 we show the aggregated performance of the 28 heuristics. In this section we seek to measure the impact of the choice of heuristic. We compute the average execution time of each heuristic and compare it against the average execution time of all heuristics. We establish the results shown in Table 6.4, which are the maximum and minimum differences observed across all the 28 heuristics on each application. While it appears that the relative difference is relatively low, typically around 1%, it is always less than 5%, with the largest difference being in the POTRF test case. In the latter case, the slowest heuristic is nearly 10% slower than the fastest.

Application	FMM	Chameleon POTRF	Chameleon GEMM	Chameleon GEQRF	PaStiX	QrMUMPS
longest time	+3%	+5%	< +1%	< +1%	+1.5%	+1%
shortest time	> -1%	-4%	> -1%	> -1%	-1%	-1%

Table 6.4: Longest and shortest relative time observed between heuristics across all test-cases.

We provide the average relative differences between heuristics for the Cholesky factorization in Chameleon

(POTRF) in Figure 6.13. This is the application in which the choice of heuristic has the most impact.

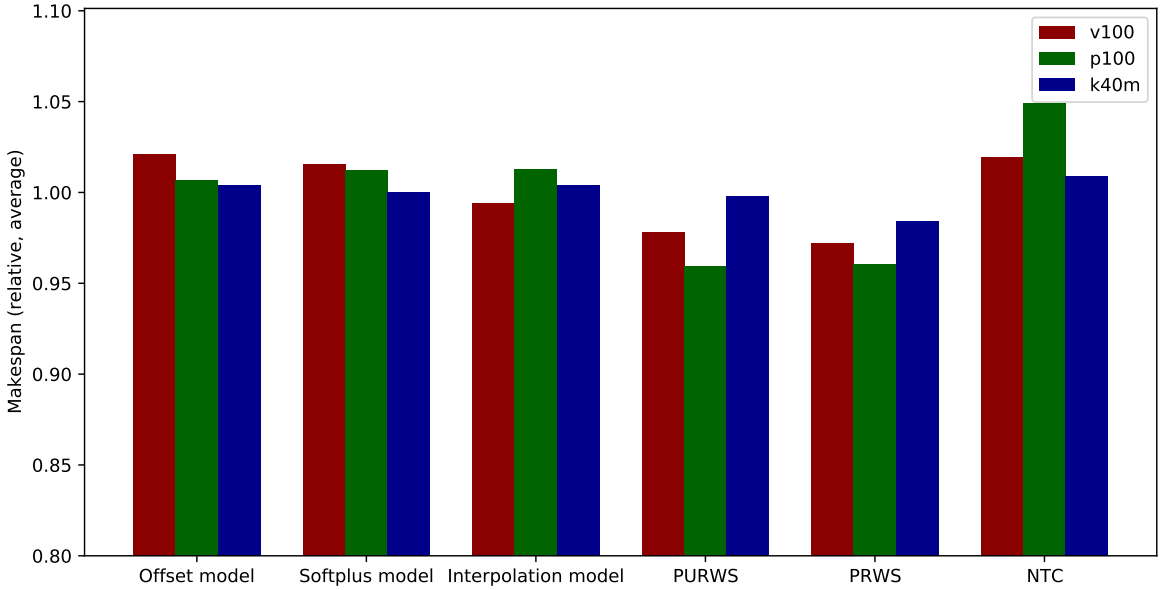


Figure 6.13: Relative difference between 6 heuristics in the case of the Cholesky factorization (Chameleon POTRF).

We observe that the heuristics PRWS and PURWS are the ones that give the best execution times, while the NTC (NOD Time Combination) heuristic is the one leading to the worst execution times for this application.

This study suggests that the choice of heuristic typically has an impact of less than 1% on the resulting execution time. The highest impact we measure is less than 10% slowdown between the fastest and the slowest heuristic in the POTRF test case. The impact of the choice of heuristic is, therefore, limited compared to the one of the scheduler. In practice, this implies that application developers can rapidly assess the performance of Heteroprio on their application only by testing one heuristic (typically with a $\pm 1\%$ makespan confidence interval). Additionally, once a user determines that Heteroprio is efficient for their application, they can further fine-tune the scheduler by benchmarking different heuristics and choosing the best one.

6.3.3 Evaluation on a stencil application

The results of the previous section suggest that AutoHeteroprio is a competitive scheduler for StarPU. This scheduler has been designed with a generic approach, as we used randomly generated graphs to create the heuristics. We have shown that this approach effectively leads to efficient priorities in already existing applications. In this section, we provide insights into the ability of AutoHeteroprio to find the correct priorities on the D3Q27 stencil application we use in Section 5.4.

Given that the scheduling challenge varies greatly depending on the subgrid partitioning and that we have already provided a comparison of different schedulers in Section 5.4.2, we only focus on the priorities that AutoHeteroprio finds for our stencil application. We run the application 28 times, each time with a different heuristic. The chose grid layout is $2 \times 8 \times 2$, and the experiment is run with 2 nodes, each with 2 P100 GPUs. The used priorities are printed every 32 times a task is pushed to the scheduler and a script is used to extract and count them. This methodological choice allows to capture the potential evolution of the priorities during the execution of the application.

Table 6.5 shows the proportion of the different priorities found by AutoHeteroprio. Since the application is bottlenecked by the inter-node communication, the found priorities do not appear to significantly impact the execution time. However, it is still interesting to analyze the priorities found by AutoHeteroprio and relate them to the properties of the codelets.

Priorities	proportion
`i2s - s2i - step`	57.8%
`s2i - i2s - step`	19.4%
`s2i - step - i2s`	19.0%
`step - s2i - i2s`	3.8%

Table 6.5: Proportion of the different priorities found by AutoHeteroprio in our stencil application. `s2i` and `i2s` correspond to the `subgrid_to_interface` and `interfaces_to_subgrid` codelets, respectively.

Codelet	average NOD	average URT
`step`	0.54	0.56
`subgrid_to_interface`	2.36	0.93
`interface_to_subgrid`	2.54	1.12

Table 6.6: Codelet-specific data for the D3Q27 simulation.

Table 6.6 provides the average NOD and URT, which are the two most interesting metrics for this case. Since we only have GPU codelets, the $diff$ (time difference between the execution on this architecture and the execution on the fastest architecture) is always 0. The NOD (formula 6.6) estimates the number of new tasks that could be released after the execution of a task, while the URT (formula 6.8) estimates the (normalized) amount of working time that could be added to the processing unit.

Ranking the codelets by either their NOD or URT always gives the same order: `interface_to_subgrid - subgrid_to_interface - step`, which is the most common priority found by AutoHeteroprio. The configuration `subgrid_to_interface - interface_to_subgrid - step` are the second most common priorities found by AutoHeteroprio. These second priorities also make sense, as `subgrid_to_interface` and `interface_to_subgrid` are very close in terms of NOD and URT. As the metrics are updated regularly, a swap between `subgrid_to_interface` and `interface_to_subgrid` is likely to occur simply due to the variance in the measurements. The third priority set, `subgrid_to_interface - step - interface_to_subgrid`, appears to be associated with heuristics that value having a low absolute execution time. Finally, the last priority set, `step - subgrid_to_interface - interface_to_subgrid`, is simply an edge case for a heuristic (not described in this work for the sake of conciseness) where having a $diff$ of 0 sets all the scores to 0 and make the order arbitrary.

With only 3 task types and a single architecture, this experiment helps us understand the behavior of AutoHeteroprio on a simple application. We see that the found priorities usually align with the goals we would expect from a scheduler. The tasks that are the most critical (high NOD or URT) are often prioritized first.

In the next section, we will conclude this chapter and discuss the potential improvements that could be made to AutoHeteroprio.

6.4 Discussion

This study has demonstrated the evolution of the Heteroprio scheduling paradigm from semi-automatic to fully automatic, eliminating the need for user intervention in the decision-making process of the scheduler. Through the development and validation of specific heuristics within an execution simulator, this transition has proven to be effective. These heuristics have delivered performance levels comparable to, or in some cases superior to, the Heteroprio framework in real-world applications.

At the outset of this chapter, the primary goal was articulated as establishing a fully automatic scheduler, with a secondary goal of ensuring high performance. The introduction of AutoHeteroprio marks the achievement of the first goal by presenting a system that operates autonomously. In terms of performance, AutoHeteroprio has demonstrated efficiency, matching or exceeding the performance of the semi-automatic version in the majority of scenarios. However, it is important to note that in specific cases, such as with the stencil application, AutoHeteroprio does not reach the performance levels of *dmda* or *dmdas*. This discrepancy can be attributed to the coarse granularity of tasks within the application. It is also observed

that AutoHeteroprio encounters performance limitations under circumstances where Heteroprio similarly struggles. These limitations are inherent to the design principles of Heteroprio and are not addressed by the current scope of this work.

Expanding on the methodologies employed in this study, a limitation in AutoHeteroprio is its reliance on multiple heuristics for the computation of priorities. The findings indicate that the selection of a specific heuristic does influence the performance of the scheduler, albeit with a limited impact. The necessity to choose a heuristic—and potentially adjust its hyperparameters based on application-specific performance needs—poses a challenge. Currently, a default heuristic is employed, with the expectation that users may switch it as required. An improvement could involve dynamically selecting the heuristic, allowing it to adapt during the execution of the application based on the performance metrics of the scheduler. However, this approach introduces complexities in defining effective runtime performance evaluation metrics and criteria for selecting the optimal heuristic.

Another potential area for enhancement is the clustering of tasks. In the current version of AutoHeteroprio, tasks are clustered by task type, which is not a true StarPU concept. The differentiation of tasks relies on their names, which works well in practice but is not a robust approach. A more sophisticated approach could incorporate statistical techniques for clustering, accommodating both the qualitative (e.g., the task name, the availability of a CPU/GPU implementation) and quantitative (e.g., the expected execution time, the NOD) data associated with tasks. For this purpose, several suitable clustering algorithms have been identified, such as the K-Prototypes algorithm [119], TwoStep Cluster Analysis [249], and hierarchical clustering algorithms based on the Gower distance [283]. Whether this strategy would enhance performance, considering the potential overhead from per-task clustering, is not clear.

Further contemplation leads to the idea of applying heuristics on a per-task basis rather than per type. Currently, tasks are aggregated by "type" due to the bucket mechanism of Heteroprio. A more granular approach would allow metrics to directly reflect individual tasks, not merely an average across a bucket. This shift would necessitate a fundamental reevaluation of the underlying scheduling mechanism, which is currently not optimized for distinguishing between individual tasks. Hayfa *et al.* are currently working on a scheduler that considers each task individually [261].

Finally, there is potential for developing a dual scheduling mode that responds to the runtime context. This idea stems from observations that Heteroprio shows rapid performance when tasks have low execution times, case in which the overhead from scheduling is critical. In contrast, when task execution times are high, the impact of scheduling overhead is less significant, allowing for more time to be dedicated to informed decision-making. A dual mode, alternating between Heteroprio and HEFT (DAG analysis) based on the context, could enhance scheduling efficiency. The criteria for selecting the mode could include factors such as the presence of starved workers, execution time of tasks, or the number of tasks in the queue. While this method holds promise for improving performance, implementing it properly requires significant effort. Integrating Heteroprio and dmda (a StarPU scheduler based on HEFT) involves dealing with a range of technical details aimed at enhancing their effectiveness, which may not always be compatible.

This chapter concludes our contributions to the field of task-based runtime systems. In Chapter 3, we have introduced the challenges associated with stencil applications and opened on the potential of task-based runtime systems to address these challenges. Then, in Chapter 4 and Chapter 5, we have presented our contributions to the integration of stencil-specific concepts into state-of-the-art task-based runtime systems. In PaRSEC, we have introduced the concept of parametrized flows, which lets us perform subgrid-level temporal blocking in an elegant manner. In StarPU, we have designed a robust and efficient stencil solver that leverages the task-based model of the runtime system. Finally, in this chapter, we have presented AutoHeteroprio, a fully automatic scheduler that extends the Heteroprio scheduler of StarPU and achieves high performance in real-world applications. These contributions are the result of a collaborative effort between us and the communities of PaRSEC and StarPU, whom we thank for their support and feedback.

This thesis focuses on achieving high performance in memory-constrained environments. While the path of using runtime systems appeared promising to achieve efficient data management, we believe that the current state of the art in task-based runtime systems is not yet mature enough to address such challenges. The reason for this is that the memory model of these runtime systems is logically optimized for performance in systems with abundant memory. Hence, few flexibility is offered to the user to control the data management. Futures works will be directed towards implementing a new memory model for StarPU, which will aim to better guide scheduling decisions based on the memory constraints of the system.

However, the rest of this thesis focuses on the development of efficient compression methods without concerns for integration into runtime systems. The goal is to provide solid proof of concepts that can be integrated into runtime systems in the future. In the next chapter, we present the discrete wavelet transform, a powerful tool for data compression, and explain how it can be tuned for efficient data compression in the context of fluid simulations.

Chapter 7

Designing wavelets for CFD simulations

7.1 Introduction to the Discrete Wavelet Transform

The Discrete Wavelet Transform (DWT) stands out as a powerful tool for signal processing and analysis. This technique breaks down a signal into coefficients, allowing to reveal its unique aspects across different scales and positions. These coefficients capture in-depth time and frequency details, providing a comprehensive representation of the signal.

The DWT forms the backbone of our approach to data compression. It enables the effective analysis and modification of multi-dimensional data, offering a flexible framework for achieving high compression ratios. The evolution of wavelet compression traces back to the works of Haar and Gabor, transitioning from concepts of continuous wavelets to the DWT. Grossmann and Morlet introduced the term "wavelet" in the 1980s [106], marking the beginning of a significant shift in how signals could be processed. Subsequent contributions by Daubechies, such as [83] and [84], have solidified wavelet theory, culminating in its application in standards like JPEG2000 for image compression [251]. The comprehensive book by Mallat [255] offers an extensive review of wavelet research, providing insights into the theoretical and practical aspects of wavelet analysis.

Grasping specific properties of wavelets, such as symmetry, vanishing moments, compact support, and mass conservation, helps understand the design of the wavelets utilized in this research. Symmetry in wavelets facilitates image reconstruction by ensuring that the error pattern is symmetric, which is beneficial for visual perception. Vanishing moments enable wavelets to disregard specific data trends, beneficial for noise reduction and data compression. Mass preservation is critical in ensuring that the total mass or information content of the signal remains unchanged post-transform, a necessity for accurate simulations. We provide some additional details on these properties in the Appendix 11.2, but the literature referenced earlier, particularly the works by Daubechies and Mallat, provides in-depth insights into these properties.

A well-known concept in wavelet theory is Multiresolution Analysis (MRA) [110, 4, 193]. It provides a robust framework for efficient signal processing and analysis, enabling the decomposition of signals at different resolution levels. The idea of MRA is to decompose a dicretized signal into a sequence of approximations at different resolution levels, each capturing the characteristics of the signal at a specific scale. This process can be reversible, allowing for the reconstruction of the original signal from the approximations.

Figure 7.1 illustrates the general principle of MRA using low-pass and high-pass filters. Each level of discrete wavelet transform corresponds to a pair of low-pass and high-pass filters. The low-pass filter obtains the approximation coefficients $a_{j,k}$, while the high-pass filter captures the detail coefficients $d_{j,k}$. The detail coefficient capture the disrepancy between the "expected" signal and the actual signal. The notation $a_{j,k}$ and $d_{j,k}$ denote the coefficients at the jth level of decomposition. The approximation coefficients serve as input for the next decomposition level, and the process repeats until the desired level of decomposition is achieved. If the transform is reversible, the original discretized signal can be reconstructed using inverse filters specially designed to combine the approximation and detail coefficients from each level.

To visually show the concept of MRA, refer to Figure 7.2, which depicts the process using a Gaussian-

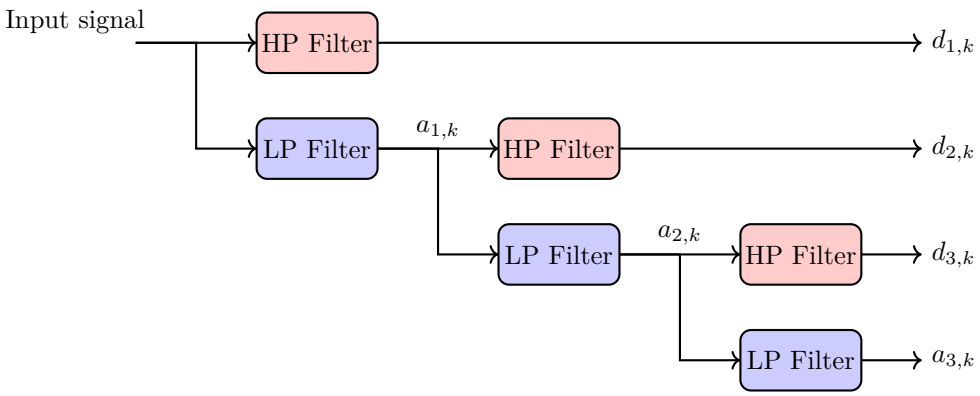


Figure 7.1: General principle of Multiresolution Analysis using LP and HP Filters. The notation $a_{j,k}$ and $d_{j,k}$ denote the coefficients at the jth level of decomposition.

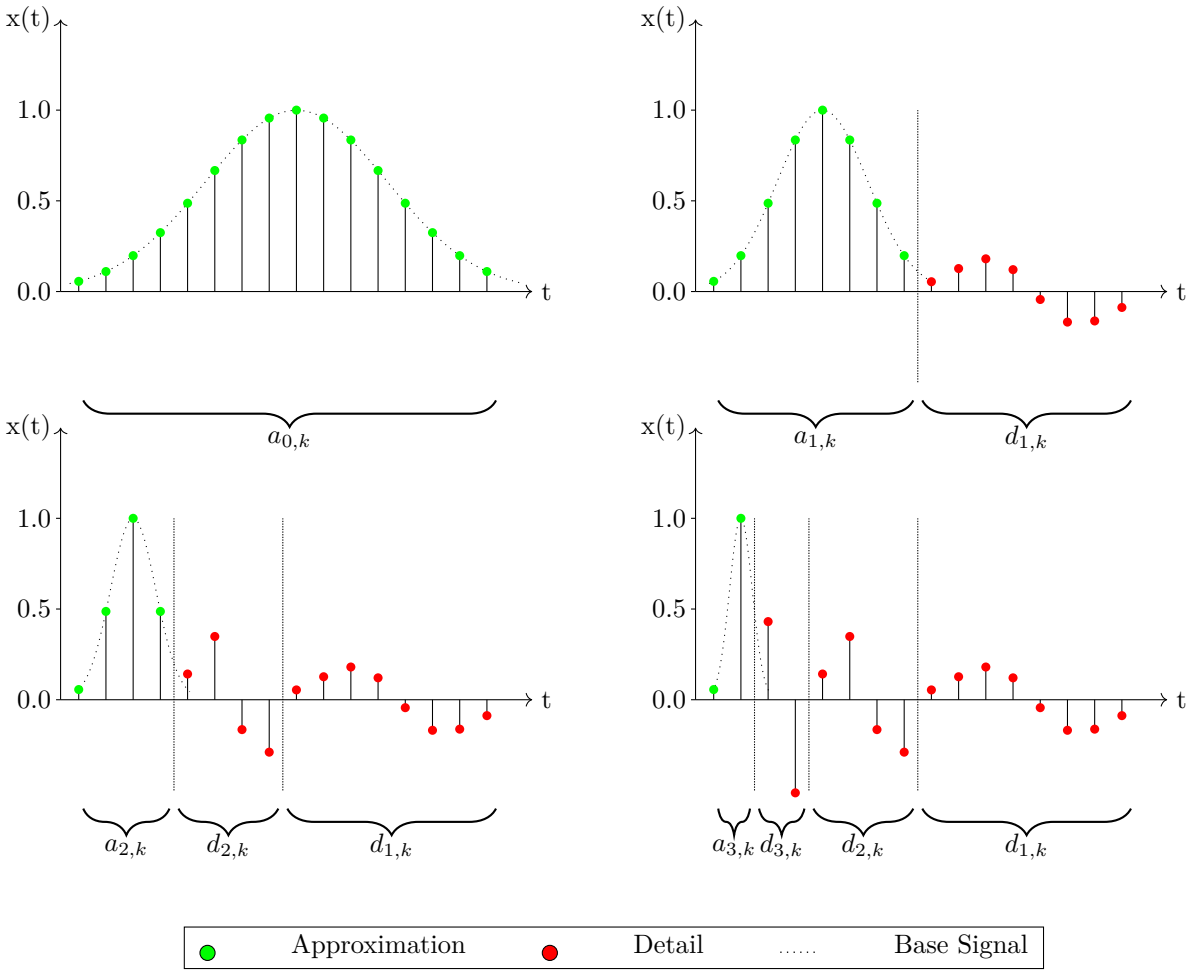


Figure 7.2: Multiresolution Analysis using a Gaussian function for signal decomposition. The continuous signal (dotted line) is represented by 16 sampled points. The green approximation coefficients capture the low-frequency components, while the red details represent the discrepancies between the approximation and the analyzed signal. The notation $a_{j,k}$ and $d_{j,k}$ indicates the coefficients at the jth level of decomposition. For example, $d_{1,k}$ corresponds to the detail coefficients after one level of wavelet transform. After each decomposition level, the signal is downsampled by a factor of two.

like function as the original signal. The green dots represent the original signal obtained by sampling the continuous original signal at equally spaced intervals. The red dots represent the detail coefficients, highlighting discrepancies between the approximation coefficients and the exact values of the original signal. In this example, a variation of Haar wavelets is employed, where $a_{j,k}$ coefficients remain identical to those of the previous level, and $d_{j,k}$ coefficients are calculated as the difference between the analyzed signal and the $a_{j,k}$ coefficients. Note that these wavelets are designed for clarity and pedagogical purposes, and may not be efficient or practical for real-world applications.

Overall, MRA offers a structured approach for breaking down signals into layers at various scales, enhancing signal processing and analysis efficiency. It operates under the premise that signals display consistency or regularity across these scales. When signals vary greatly at different scales, MRA may not be as effective. However, the required regularity is often seen in CFD simulations, making MRA a suitable tool for data compression in this context. The selection of wavelets is also crucial, as it determines the properties of the compression scheme which can be critical for the accuracy of the simulation. The forthcoming section will explore wavelet design, focusing on attributes that render them ideal for CFD simulations.

7.2 Designing Wavelets for CFD Simulations

7.2.1 Challenges

Designing wavelets for CFD simulations focuses on achieving efficient data compression while preserving important aspects of the signal. As discussed in Section 3.3, the global grid is typically divided into subgrids, each managed by a separate processor, which influences the application of wavelet transforms. Typically, wavelet transforms anticipate a certain size for these subgrids, which introduces constraints on the possible global grid size. Adjusting the sizes and numbers of these subgrids, however, often provides a satisfactory solution to these constraints. A significant challenge arises from the fact that signals within these subgrids do not exhibit periodic patterns, contrary to the usual expectations for wavelet transforms. Considering wavelets that account for neighboring values could offer a solution, but such values are not always accessible. Focusing on ensuring vanishing moments across the analyzed interval, especially at non-periodic boundaries, presents another strategy to address the challenges of non-periodicity. In practice, maintaining constant boundary values during the transform facilitates the synchronization of subgrids, although the specifics of this synchronization depend on the chosen computational scheme and its implementation. This method simplifies the process, though it is not a strict requirement.

An important condition for our application is to ensure that the compression scheme is conservative, meaning that the mass of the reconstructed signal is equal to the mass of the original signal. We ensure this property by verifying that the details wear no mass and refer to it as mass conservation. Ensuring a conservative scheme is identified as essential in most numerical schemes [118]. However, simply preserving mass does not guarantee the accuracy of simulations; the induced error of the compression must also be managed effectively [73]. Achieving mass conservation at the global grid level is, hence, imperative. It is trivial to show that preserving the mass within a subgrid is a sufficient but not necessary condition to achive mass conservation at the global level.

The final goal is to achieve high compression ratios. As we have explained in the previous section, the MRA framework produces a set of approximations that are close to the original signal, and a set of details that capture the discrepancies between the approximations and the original signal. The details are expected to be small in most cases, which reduces the entropy of the data. This can be exploited to achieve effective compression, as we will see in Section 7.3.3. The implication on the design choices is that we must ensure that the details are as small as possible. This is often achieved in practice by ensuring filtering of polynomial trends up to a given order, which is a direct consequence of the vanishing moments property of the wavelets.

In the next sections, we present two approaches: the first one is based on the LGT5/3 and CDF9/7 wavelet constructions, while the second one, which we will refer to as lifter Haar wavelets, is derived from the Haar framework. We will first present them in the 1D case, and extend them to the multi-dimensional case in Section 7.3.1.

7.2.2 Notations and Definitions

Here, we assume that the signal is defined on the interval $[0, 1]$ for the sake of simplicity. The extension to other intervals is straightforward, and we will not discuss it here. Let f, defined on $[0, 1]$, be the non-periodic signal on which we want to apply the DWT and J be the sampling scale. It is important to note that our formulation differs from the standard wavelet transform, which is typically applied to signals defined on the entire real line or on a periodic interval. A lot of works have been done on wavelets on the interval (see, for instance [72, 78, 21, 37]). We will use one of the simplest approach, the so called mirror wavelet described in the book of Mallat [175].

We begin with $2^J + 1$ sampling points:

$$x_{J,k} = k2^{-J}, \quad 0 \leq k \leq 2^J, \quad J \geq 0, \tag{7.1}$$

where k corresponds to the index of the point in the grid $x_{J,k_{0\leq k\leq 2^J}}$. We can note that for any scale J, the first and last points are fixed:

$$x_{J,0} = 0, \quad x_{J,2^J} = 1. \tag{7.2}$$

For a given scale J, the signal is represented using $2^J + 1$ points. Let us point out that at the coarsest scale $J = 0$ the signal is represented by its values at the two boundary points $x = 0$ and $x = 1$.

We now define the wavelet coefficients $a_{j,k}$ and $d_{j,k}$ to refer to the approximation and detail coefficients at scale j, respectively. At the finest scale $j = J$, the approximation coefficients are the samples of the signal f:

$$a_{J,k} = f(x_{J,k}), \quad 0 \leq k \leq 2^J. \tag{7.3}$$

Then, the DWT allows to compute the approximation and detail coefficients at a coarser scale. If we adopt a generic view of the DWT, the approximation and detail coefficients at scale $j - 1$ are found by applying the DWT transform on the approximation coefficients at scale j.

$$a_{j-1,k} = T_{\text{j,k,approx}}(a_{j,0}, a_{j,1}, \ldots, a_{j,2^j}), \quad 0 \leq k \leq 2^{j-1} \tag{7.4}$$

$$d_{j-1,k} = T_{\text{j,k,detail}}(a_{j,0}, a_{j,1}, \ldots, a_{j,2^j}), \quad 0 \leq k < 2^{j-1} \tag{7.5}$$

for $0 \leq k \leq 2^j$, where $T_{\text{j,k,approx}}$ and $T_{\text{j,k,detail}}$ are the transforms that compute the approximation and detail coefficients at scale $j - 1$ from the approximation coefficients at scale j. However, if compactly supported wavelets are used, the transform can often be expressed in a more elegant manner. The goal is find the T transforms that satisfy a set of constraints. For our purposes, the constraints and goals have been stated in the previous section.

Finally, let us introduce the matrix representation of the DWT, which is often used in the classical wavelet theory. Firstly, the DWT can be expressed as a vector-to-vector application:

$$(u_0, u_1, \ldots, u_{2^j}) \mapsto (v_0, v_1, \ldots, v_{2^j}), \tag{7.6}$$

where u is the vector of approximation coefficients at scale j and v is the vector of approximation and detail coefficients at scale $j - 1$. For example, we can represent u and v as:

$$u = \begin{pmatrix} a_{j,0} \\ a_{j,1} \\ \vdots \\ a_{j,2^j} \end{pmatrix} \tag{7.7}$$

and

$$v = \begin{pmatrix} a_{j-1,0} \\ d_{j-1,0} \\ a_{j-1,1} \\ d_{j-1,1} \\ \vdots \\ a_{j-1,2^{j-1}-1} \\ d_{j-1,2^{j-1}-1} \\ a_{j-1,2^{j-1}} \end{pmatrix}. \tag{7.8}$$

Classically, u and v have the same size, which lets us represent the transform as a matrix A such that:

$$v = Au. \tag{7.9}$$

Then, if the transform is bijective, there exists a matrix A^{-1} such that:

$$u = A^{-1}v. \tag{7.10}$$

7.2.3 LGT5/3 and CDF9/7 Wavelets

LGT5/3 and CDF9/7 widely known wavelets that are extensively used in image and signal processing. The LGT5/3 wavelets, attributed to Le Gall and Tabatabai, are also known as first-order 5/3 biorthogonal wavelets [153, 84, 71], while the CDF9/7 wavelets are commonly referred to as the Cohen-Daubechies-Feauveau wavelets [71]. In this section, we explain how we build wavelets derived from these constructions for efficient CFD compression.

For constructing our DWT, we follow the lifting scheme introduced by Sweldens [258, 259], which provides a flexible and efficient approach for wavelet construction. At any scale, the first and last coefficients are fixed to the boundary values of the signal, with the intention of making the boundaries of the interval more easily accessible in CFD implementations:

$$a_{j,0} = x_{J,0} = f(0) \quad \text{and} \quad a_{j,2^j} = x_{J,2^J} = f(1), \quad 0 \leq j \leq J. \tag{7.11}$$

Since these two points remain unchanged, they can be accessed more directly during the synchronization of different subgrids. They are intentionally chosen to have even k indices for $j \geq 1$ and will always correspond to approximation coefficients. In our setting, at scale $j \geq 1$, there are $2^{j-1} + 1$ even indices and 2^{j-1} odd indices, which correspond to the framework we set in the previous section, but differs from the usual wavelet construction.

Now, let us introduce a linear interpolation expectation that will lead to vanishing moments. Firstly, we expect the approximation coefficients to satisfy:

$$a_{j,k} \approx f(x_{j,k}). \tag{7.12}$$

If we expect the signal to be locally linear, we can also expect the odd samples to satisfy the following linear interpolation:

$$a_{j,2k+1} \approx \frac{a_{j,2k} + a_{j,2(k+1)}}{2}, \quad 0 \leq k \leq 2^{j-1} - 1. \tag{7.13}$$

We can then define the detail coefficients at scale $j-1$ as:

$$d_{j-1,k} = a_{j,2k+1} - \frac{a_{j,2k} + a_{j,2(k+1)}}{2}, \quad 0 \leq k \leq 2^{j-1} - 1, \tag{7.14}$$

which will ensure near-zero detail coefficients if the signal is locally linear.

We now need to define the approximation coefficients $a_{j-1,k}$ at scale $j-1$. We want these coefficients to satisfy multiple properties. First, we want them to be close to the signal f at the sampling points $x_{j-1,k}$, to match the expectation we made in equation (7.12). Second, to achieve minimal memory intensity, we want to use as few other coefficients as possible. One way to achieve this is to only use the approximation coefficients at scale j and the detail coefficients at scale $j-1$. For example, we can introduce coefficients $\alpha_{j,k}$, and define:

$$a_{j-1,k} = a_{j,2k} + \alpha_{j-1,k-1} d_{j-1,k-1} + \alpha_{j-1,k} d_{j-1,k}, \quad 0 < k < 2^{j-1}. \tag{7.15}$$

This corresponds to the lifting part of the lifting scheme (because the coefficients $a_{j-1,k}$ are "lifted"). Let us note that the first and last coefficients (with $k = 0$ and $k = 2^{j-1}$) correspond to the fixed boundary values and are not modified. This approach ensures that the approximation coefficients remain close to the signal, as the detail coefficients are expected to be small. We have the relation:

$$a_{j-1,k} = a_{j,2k} + \mathcal{O}(d_{j-1,k-1} + d_{j-1,k}), \tag{7.16}$$

which, combined with equation (7.3), ensures that all the approximation coefficients remain "close" to the signal as long as the detail coefficients are small. Several choices of coefficients $\alpha_{j,k}$ are possible, but only one satisfies the mass conservation property. The mass conservation property ensures that the mass of the original sampled signal in $[0, 1]$ exactly equals that of the wavelets associated with the approximation coefficients at any scale. It can be expressed as:

$$\frac{f(0)+f(1)}{2} + \sum_{k=1}^{2^{j-1}-1} a_{j-1,k} = \frac{f(0)+f(1)}{4} + \frac{1}{2}\sum_{k=1}^{2^{j}-1} a_{j,k}. \tag{7.17}$$

This equation is essentially a trapezoidal quadrature formula. The $\frac{1}{2}$ factor on the boundary values stems directly from the shape of the wavelets. Appendix 11.2.5 provides a visual interpretation of this phenomenon. This $\frac{1}{2}$ factor does pose a practical challenge in the general case, since it could allow some mass to escape from the subgrids. However, thanks to the fixed boundaries we imposed in equation (7.11), mass conservation is guaranteed at the level of the subgrid, as we will show in Section 7.3.2 and Appendix 11.3.2. Another way of stating the mass conservation property is to say that the details wear no mass. This implies that any modification of the detail coefficients will not affect the mass of the reconstructed signal, which has important implications for the compression scheme.

We can check that the only choice of coefficients $\alpha_{j,k}$ that satisfies mass conservation is:

$$\alpha_{j-1,k} = \begin{cases} \frac{1}{4} & \text{if } 0 < k < 2^{j-1}-1 \\ \frac{1}{2} & \text{if } k = 0 \text{ or } k = 2^{j-1}-1. \end{cases} \tag{7.18}$$

Thanks to these coefficients, we can compute the approximation coefficients at scale $j-1$.

Hence, to perform the DWT using the lifting scheme, we first compute the detail coefficients at scale $j-1$ using equation (7.14), and then the approximation coefficients at scale $j-1$ using equation (7.15). The inverse transform can be obtained by inverting these equations. First the even approximation coefficients at scale j can by inverting equation (7.15):

$$a_{j,2k} = a_{j-1,k} - \alpha_{j-1,k-1} d_{j-1,k-1} - \alpha_{j-1,k} d_{j-1,k}, \quad 0 < k < 2^{j-1}. \tag{7.19}$$

Then, the odd approximation coefficients at scale j can be obtained by inverting equation (7.14):

$$a_{j,2k+1} = d_{j-1,k} + \frac{a_{j,2k} + a_{j,2(k+1)}}{2}, \quad 0 \le k \le 2^{j-1}-1. \tag{7.20}$$

We can see that the lifting scheme offers an efficient way to compute the DWT. Both the forward and inverse transforms can be computed in a single pass over the coefficients, which is a significant advantage in terms of memory access and computational complexity.

Another way to represent the same transform is the matrix form (see equations 7.9 and 7.10). For $j = 3$, we obtain

$$A = \frac{1}{8}\begin{bmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -4 & 8 & -4 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 4 & 5 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -4 & 8 & -4 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 6 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 & 8 & -4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & 5 & 4 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & -4 & 8 & -4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix} \tag{7.21}$$

and

$$A^{-1} = \frac{1}{8}\begin{bmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 6 & 4 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -4 & 8 & -2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2 & 4 & 6 & 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & 8 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 4 & 6 & 4 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 8 & -4 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 4 & 6 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}. \tag{7.22}$$

In the even rows of A (approximation coefficients), the wavelet low-pass filter coefficients are present. Conversely, the odd rows (detail coefficients) contain the wavelet high-pass filter coefficients. The sum of the coefficients in the even rows is 1, while the sum of the coefficients in the odd rows is 0, which is mandatory for the mass conservation property. Notably, the filters at the boundaries of the interval only require one coefficient, indicating their minimal support. In contrast, the filters in the middle of the interval have wider (but compact) support, extending over multiple coefficients. Away from the boundaries, we observe the filters corresponding to the LGT5/3 filter bank.

It is possible to increase the number of vanishing moments by using the CDF9/7 wavelets. With these wavelets, the construction is similar. The contruction of the details become:

$$d_{j-1,k} = s_{j,2k+1} - \frac{-s_{j,2(k-1)} + 9s_{j,2k} + 9s_{j,2(k+1)} - s_{j,2(k+2)}}{16}, \quad 0 \leq k \leq 2^{j-1} - 1. \tag{7.23}$$

To make it work for the edge values, we simply extend the coefficients by symmetry, which reduces the order of the wavelets to 1 at the boundaries.

We get the following matrix representation for the CDF9/7 wavelets:

$$A = \frac{1}{64}\begin{bmatrix} 64 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -36 & 64 & -32 & 0 & 4 & 0 & 0 & 0 & 0 \\ -17 & 32 & 39 & 16 & -7 & 0 & 1 & 0 & 0 \\ 4 & 0 & -36 & 64 & -36 & 0 & 4 & 0 & 0 \\ 1 & 0 & -8 & 16 & 46 & 16 & -8 & 0 & 1 \\ 0 & 0 & 4 & 0 & -36 & 64 & -36 & 0 & 4 \\ 0 & 0 & 1 & 0 & -7 & 16 & 39 & 32 & -17 \\ 0 & 0 & 0 & 0 & 4 & 0 & -32 & 64 & -36 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 64 \end{bmatrix} \tag{7.24}$$

and

$$A^{-1} = \frac{1}{64}\begin{bmatrix} 64 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 36 & 48 & 32 & -7 & -4 & 1 & 0 & 0 & 0 \\ 0 & -32 & 64 & -16 & 0 & 0 & 0 & 0 & 0 \\ -4 & -18 & 36 & 46 & 36 & -8 & -4 & 2 & 0 \\ 0 & 0 & 0 & -16 & 64 & -16 & 0 & 0 & 0 \\ 0 & 2 & -4 & -8 & 36 & 46 & 36 & -18 & -4 \\ 0 & 0 & 0 & 0 & 0 & -16 & 64 & -32 & 0 \\ 0 & 0 & 0 & 1 & -4 & -7 & 32 & 48 & 36 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 64 \end{bmatrix}. \tag{7.25}$$

We can see that the CDF9/7 wavelets have a larger support than the LGT5/3 wavelets, which is a direct consequence of the higher order of vanishing moments. This is a disadvantage in terms of memory usage and computational complexity, but also in terms of compression ratios when the signal includes a lot of high-frequency components. Such components are often present in CFD simulations, where discontinuities are common.

The choice between the LGT5/3 and the CDF9/7 wavelets is not straightforward, as both have their advantages and disadvantages. The LGT5/3 wavelets is less memory intensive and requires less computation, but the CDF9/7 wavelets can yield better compression ratios in smooth regions. However, although it can be debated, we believe that the increase in the order of vanishing moments does not justify the additional complexity of implementation and the additional memory usage. A more detailed discussion on the choice of wavelets can be found in Section 7.4.

Overall, both the LGT5/3 and CDF9/7 wavelets appear to be a suitable choice for CFD simulations, as they have vanishing moments, conserve mass, and are efficient to compute. However, one potential drawback is the constraint on the size of the data grid, which must be a power of 2 plus 1. It is conceivable that this could lead to an inefficient use of the hardware in practice, as most hardware is designed to work with powers of 2. We did not find a way to circumvent this constraint while ensuring all the properties we stated in Section 7.2.1. However, if we relieve the constraint of maintaining the boundary values constant, it is possible to design biorthogonal wavelets that do not require the grid size to be a power of 2 plus 1. The next section will present one such design, based on the Haar wavelets.

7.2.4 Lifted Haar Wavelets

We express our sincere thanks to Erwan Deriaz for his instrumental role in developing the wavelet design featured in this section. The Haar wavelets, recognized for their simplicity, serve not only as a fundamental tool for educational purposes but also demonstrate efficacy in practical applications. While the piecewise constant Haar wavelets are limited to filtering polynomial trends at order 0 (constant), which insufficient for appropriate data compression, we can enhance their capability by increasing their support. This section is dedicated to outlining a DWT scheme based on Haar wavelets, with specific modifications to ensure mass conservation and filtering of polynomial trends up to the second order.

We first modify the sampling points given in equation (7.1) to:

$$x_{J,k} = k2^{-J}, \quad 0 \leq k < 2^J, \quad J \geq 0. \tag{7.26}$$

Contrary to the previous construction, the first and last points depend on the scale J:

$$x_{J,0} = 2^{-J-1}, \quad x_{J,2^J-1} = 1 - 2^{-J-1}, \quad J \geq 0, \tag{7.27}$$

which is why we cannot ensure constant boundary values anymore.

We propose the following construction to compute the coefficients at scale $j-1$:

$$d_{j-1,k} = a_{j,2k+1} - a_{j,2k}, \quad 0 \leq k < 2^{j-1}, \quad j \geq 1, \tag{7.28}$$

$$a_{j-1,k} = a_{j,2k} + \frac{1}{2} d_{j-1,k}, \quad 0 \leq k < 2^{j-1}, \quad j \geq 1. \tag{7.29}$$

It is close to the piecewise constant Haar construction, which does not allow to filter polynomial trends greater than order 0 (constant), due the inability to use neighbor coefficients. It is, however, possible to increase the filtering order by introducing a second lifting, this time on the detail coefficients:

$$d'_{j-1,k} = \begin{cases} d_{j-1,0} + \frac{3}{4} a_{j-1,0} - a_{j-1,1} + \frac{1}{4} a_{j-1,2} & \text{if } k = 0 \\ d_{j-1,N-1} + \frac{1}{4} a_{j-1,N-1} - a_{j-1,N-2} + \frac{3}{4} a_{j-1,N-3} & \text{if } k = N-1 \\ d_{j-1,k} + \frac{1}{4}(a_{j-1,k-1} + a_{j-1,k+1}) & \text{otherwise,} \end{cases} \tag{7.30}$$

where $N = 2^{j-1}$ (the number of coefficients at scale $j-1$).

The matrix representation of this transform for $j = 3$ is:

$$A = \frac{1}{8} \begin{bmatrix} 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ -5 & 11 & -4 & -4 & 1 & 1 & 0 & 0 \\ 0 & 0 & 4 & 4 & 0 & 0 & 0 & 0 \\ 1 & 1 & -8 & 8 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 4 & 0 & 0 \\ 0 & 0 & 1 & 1 & -8 & 8 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 4 \\ 0 & 0 & 3 & 3 & -4 & -4 & -7 & 9 \end{bmatrix}. \tag{7.31}$$

and

$$A^{-1} = \frac{1}{8} \begin{bmatrix} 11 & 5 & 1 & -1 & 0 & 0 & 0 & 0 \\ -4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ -4 & 4 & 8 & 8 & 1 & -1 & 3 & -3 \\ 0 & 0 & -4 & 4 & 0 & 0 & 0 & 0 \\ 1 & -1 & -1 & 1 & 8 & 8 & -4 & 4 \\ 0 & 0 & 0 & 0 & -4 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 9 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & -4 & 4 \end{bmatrix}. \tag{7.32}$$

We can verify that polynomial trends are filtered out thanks to this construction by applying it to a polynomial signal:

$$\begin{pmatrix} P(1) \\ P(2) \\ \vdots \\ P(8) \end{pmatrix} \cdot A = \begin{pmatrix} \frac{5}{2}a + \frac{3}{2}b + c \\ 0 \\ \frac{25}{2}a + \frac{7}{2}b + c \\ 0 \\ \frac{61}{2}a + \frac{11}{2}b + c \\ 0 \\ \frac{113}{2}a + \frac{15}{2}b + c \\ 8a \end{pmatrix}, \tag{7.33}$$

where $P(x) = ax^2 + bx + c$. We can see that all the polynomial trends are filtered out up to the second order for all the detail coefficients (odd rows) appart from the last one, which only filters out the first order.

Thus, we have proposed three distinct wavelet designs, each with its own set of advantages and disadvantages. Excluding the initial and final coefficients, the CDF9/7 wavelets are capable of filtering out polynomial trends up to the third order, lifted Haar wavelets up to the second order, and LGT5/3 wavelets to the first order. The ability to filter higher-order polynomials should lead to higher compression ratios for signals with a limited presence of high-frequency components, a topic we will delve into in the following section. Additionally, the lifted Haar wavelets present a divergence in their design compared to the CDF9/7 and LGT5/3 wavelets. While the latter two require the grid size to be a power of 2 plus 1, ensuring boundary preservation, lifted Haar wavelets operate with a grid size that is a strict power of 2 and do not maintain boundary values. Nevertheless, all three wavelet designs uphold the principle of mass conservation, essential for the accuracy of simulations. In the subsequent section, we aim to broaden the application of these three wavelet designs to multi-dimensional data, proposing a comprehensive compression methodology built upon these principles.

7.3 Compression Scheme for multi-dimensional data

7.3.1 Multi-dimensional wavelet transform

The 1-dimensional wavelet transforms we presented are directly extendable to multi-dimensional data. The idea is to apply the 1-dimensional wavelet transform to each line of the data independently for each dimension. Let us showcase the 2-dimensional wavelet transform with the LGT5/3 wavelets. To have a visually appealing representation of the 2-dimensional wavelet transform, let us modify the v vector of our LGT5/3 wavelet transform (equation 7.8) as so:

$$v = \begin{pmatrix} a_{j-1,0} \\ a_{j-1,1} \\ \vdots \\ a_{j-1,2^{j-1}} \\ d_{j-1,0} \\ d_{j-1,1} \\ \vdots \\ d_{j-1,2^{j-1}-1} \end{pmatrix}. \tag{7.34}$$

This will group the approximation (low-pass) and detail (high-pass) coefficients on each side of the vector.

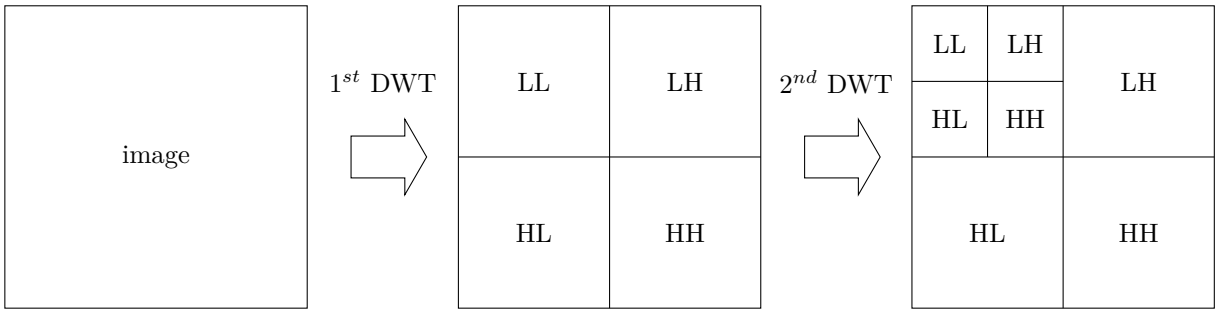


Figure 7.3: Representation of the 2-dimensional wavelet transform. The L and H correspond to the low-pass and high-pass filters on the corresponding axis.

Figure 7.3 shows the different parts of the 2-dimensional wavelet transform. The approximation coefficients are located in the upper-left corner of the image (LL), while the rest of the coefficients is divided into the LH, HL, and HH parts, depending on the dimension and the type of filter. For instance, the LH part corresponds to the low-pass filter on the x-axis and the high-pass filter on the y-axis. This representation is possible because the 1-dimensional wavelet transform is commutative across dimensions, meaning that the order in which we apply the wavelet transform to each dimension does not matter. In this view, only the LL part is directly related to the original signal, while the other parts represent different types of details. It is expected that most values in the details (LH, HL, HH) are small, which is the reason why the DWT should be able to compress the data efficiently (more on this in Section 7.3.3).

Figure 7.4 shows the result of the application of 3 steps of the wavelet scheme to a 2-dimensional image. We can recognize the original image in the approximation coefficients (upper-left corner of the image). We can also recognize a sketchy version of the image in the different detail levels. In particular, we can see that the outline of the cat is well distinguishable in the details. We can see that the sought property is reached: the parts of the image that are near-linear result in small coefficients (very dark or very bright colors), while the non-linear parts (outline) result in large coefficients.

This section has explained how the 1-dimensional wavelet transform can be extended to N dimensions. The successive application of the DWTs is a bijection that should result in a near-sparse grid of coefficients. To obtain compression, the DWT must be integrated in a compression scheme that allows to leverage the sparsity of the coefficients. In the next section, we will introduce the notion of thresholding, which is a first step towards achieving compression.

(a) Original grayscale image. (b) Resulting grayscale image.

Figure 7.4: Transform of a 2-dimensional image (left) after the application of 3 steps of the wavelet scheme (right) In the result (right), the approximation coefficients are located in the upper-left corner of the image. The remaining coefficients represent details at different levels. Details colors are obtained with $\text{mod}(x, 256)$, where x is the detail value. The resulting color ranges from 0 (black) to 255 (white).

7.3.2 Thresholding

The idea of thresholding is to nullify the detail coefficients that are below a certain threshold. There are multiple justifications to using this approach that we will discuss in this section. In this section, every reference to a DWT will be in the context of the LGT5/3 wavelets. Discussions regarding the CDF9/7 and lifted Haar wavelets will be presented in the next sections.

We explore two different scenarios as examples. The first scenario involves a function defined by the equation below, which includes a discontinuity, making it particularly interesting for our study:

$$f(x, y) = e^{x-y} \sin(2\pi(x + y)) \times \text{step}(y - x^2),$$

where the step function is defined as:

$$\text{step}(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 2 & \text{if } x < 0. \end{cases}$$

This function is shown in Figure 7.6a.

The second scenario involves data from a Saint-Venant (shallow water) simulation at time $t = 1s$, shown in Figure 7.6d. The simulation starts with a 2-meter square of water in the center of a 1-meter deep pool. More details about this simulation and its initial conditions will be provided in the next chapter (Section 8.3.1), along with Figures 8.7a, 8.7b, and 8.7d to illustrate the evolution over time. For our current discussion, we consider the simulation data at $t = 1s$ as a two-dimensional array representing the water height at each grid point. The two grids are set to a size of 1025×1025 points to match the requirements of our LGT5/3 wavelet scheme.

To discuss how the data are compressed, we analyze the histogram of the coefficients. Figures 7.5a and 7.5c show the histograms of the coefficients for the two scenarios. It provides insights into the distribution of the coefficients and the potential for compression. If the distribution is uniform, the potential for compression is low. If the distribution is skewed, the potential for compression is high.

Now let us introduce the histograms of the coefficients for the same scenarios, but after the application of 3 steps of the wavelet scheme. Figures 7.6b and 7.6e show the grids of coefficients after the application of

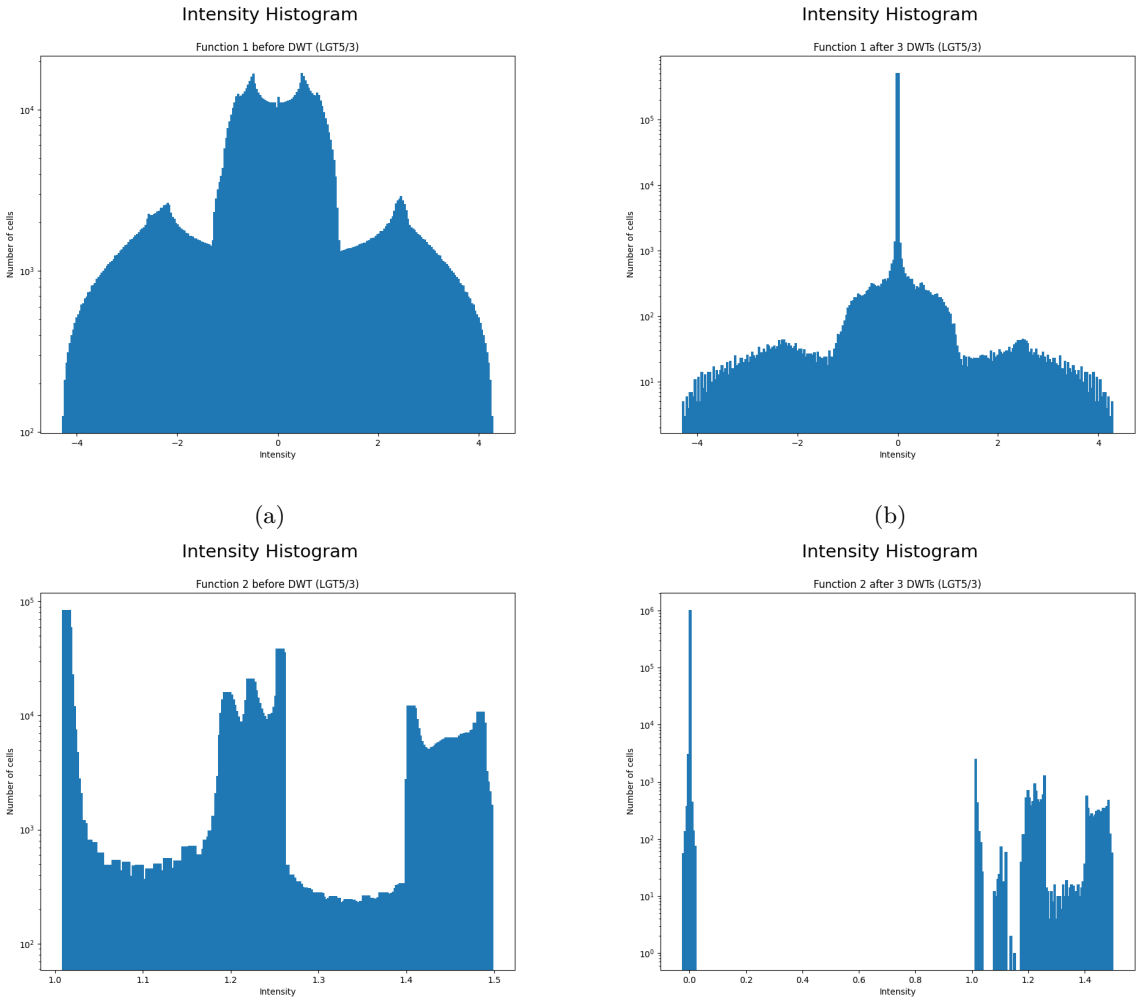


Figure 7.5: Intensity histograms of the coefficients for the two scenarios. The histograms are shown before (left) and after (right) the application of 3 steps of the wavelet scheme. The x axis represents the intensity of the coefficients, while the y axis represents the number of coefficients with the corresponding intensity. The coefficients are grouped into 256 bins distributed uniformly between the minimum and maximum values of the coefficients.

3 steps of the wavelet scheme. Figures 7.5b and 7.5d show the corresponding histograms. We can see that after the application of the wavelet scheme, the histograms become more skewed. In particular, most values are concentrated around zero. Hence, the histograms suggest that the wavelet scheme has the potential to compress the data efficiently.

A more definitive way to assess the potential for compression is to compute the entropy of the grids. The entropy is a measure of the average information content of the data and is defined as:

$$H(X) = -\sum_i p(x_i) \log_2(p(x_i)), \tag{7.35}$$

where X is the random variable, $p(x_i)$ is the probability of the value x_i, and the sum is over all possible values of X. In practice, $p(x_i)$ is estimated from the frequency of the value x_i in the data. Since we work with floating-point numbers, there is one x_i symbol per different floating-point number in the data. This is likely to create artifacts. A more common approach is to group the floating-point values into bins and compute the entropy of the resulting discrete distribution. The formula becomes:

$$H(X) = -\sum_b p(\text{bin}^b_{\min} \leq X < \text{bin}^b_{\max}) \log_2(p(\text{bin}^b_{\min} \leq X < \text{bin}^b_{\max})), \tag{7.36}$$

where the sum is over all the bins, and $\text{bin}^b_{\min}$ and $\text{bin}^b_{\max}$ are the minimum and maximum values of the b-th bin, respectively. X is the random variable representing the coefficients of the grid. The bins can be chosen in different ways, but the most common approach is to use a uniform binning. For the following, we use 256 bins, which is coarse but sufficient for our purposes. We verified that increasing the number of bins changes the entropy but not the conclusions.

The entropy of the grids provides a measure of the potential for compression. With grid cells assumed to be independent, the source coding theorem sets a minimum on the bits needed for encoding:

$$\text{Number of bits} \geq \text{Entropy} \times \text{Number of cells}. \tag{7.37}$$

Hence, there is a relationship between the entropy and the potential for compression. With 256 bins in use, the highest entropy is $\log_2(256) = 8$ bits per cell. Let us note that in the case of data produced by successive DWTs, grid cells are clearly not independent. Thus, a specific compression approach could achieve better than this theoretical minimum by leveraging the spatial correlation between the cells.

Going back to the histograms, we can compute their correponding entropies thanks to formula 7.36. The first image has an entropy of approximately 7.222 without the wavelet scheme and 1.291 after the application of 3 steps of the wavelet scheme. The second image has an entropy of approximately 6.511 without the wavelet scheme and 0.240 after the application of 3 steps of the wavelet scheme. Hence, applying the wavelet scheme, which is bijective and does not result in a loss of information, has increased the compression potential of the data. However, it is possible to further increase the compression ratio by allowing for a loss of information.

As we have stated in Section 7.2, the detail coefficients can be modified without affecting the mass of the reconstructed signal. This provides leverage to achieve higher compression ratios. One can notice that small details are responsible for a large part of the information content of the data, while having the least impact on the reconstructed signal. The idea is then to nullify the small details to achieve compression. For this we set a threshold and nullify the details that are below this threshold. What is expected is that for a low threshold value, the increase in compression ratio and the loss are low, while for a high threshold value, the increase in compression ratio is high, but the loss is also high. It is, hence, likely that there exists an optimal threshold value that maximizes the compression ratio for a given acceptable loss.

There are multiple ways to implement this idea. The most straightforward way, known as hard thresholding, is to set to zero all the details that are below the threshold. The rest of the coefficients are left unchanged. Another way, known as soft thresholding, is to move all the details towards zero by a certain amount (that can be assimilated to the threshold in hard thresholding). This second method avoids creating a hole in the histogram, which can lead to artifacts in the reconstructed signal. However, both methods can be used in practice.

It is also possible to set a different threshold depending on the scale of the details or on local properties of the data. Applying the same threshold to all the details is referred to as *global thresholding* or *VisuShrink*.

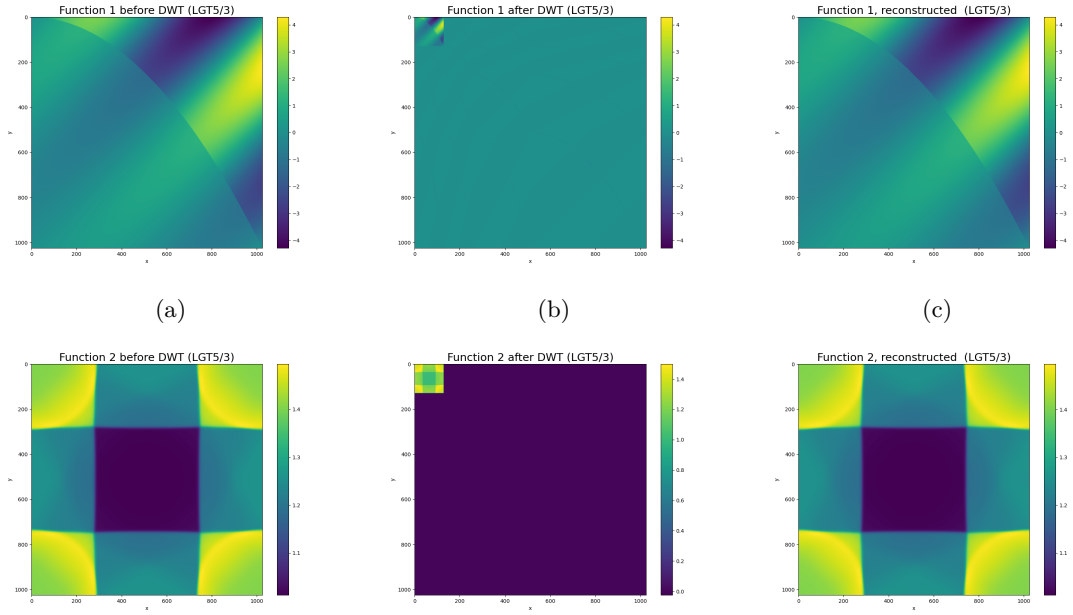


Figure 7.6: The original functions (left), the functions after 3 steps of the wavelet scheme (center), and the reconstructed functions (right). A hard threshold of 10^{-3} was used on the details before the reconstruction.

There exists several other approaches such as *SureShrink*, *BayesShrink*, and *NeighShrink* that set the threshold depending on the local properties of the data [252, 89, 90, 66]. These methods often minimize a different type of loss, such as the mean squared error or the mean absolute error [70]. However, we did not focus on these methods in this study, with the idea that if our proofs of concept are successful, the thresholding method can be refined in the future.

Thresholding	Entropy			Relative error		Max error
	Before DWT	After DWT	After threshold	eq. 7.38	eq. 7.39	
none	7.22	1.29	1.29	$-2.22e^{-16}$	$-4.44e^{-16}$	$3.55e^{-15}$
0.0001	7.22	1.29	0.47	$6.00e^{-15}$	0	$1.22e^{-4}$
0.001	7.22	1.29	0.35	$6.44e^{-15}$	$1.33e^{-15}$	$1.36e^{-3}$
0.01	7.22	1.29	0.33	$-2.55e^{-15}$	$-1.24e^{-14}$	$1.64e^{-2}$
0.1	7.22	1.29	0.28	$6.44e^{-15}$	$6.22e^{-15}$	$2.01e^{-1}$
1.0	7.22	1.29	0.23	$4.44e^{-15}$	$-6.88e^{-15}$	$9.17e^{-1}$
all	7.22	1.29	0.23	$4.44e^{-15}$	$-6.88e^{-15}$	$7.41e^{-1}$
random	7.22	1.29	6.74	$6.88e^{-15}$	$-4.77e^{-15}$	7.56

Table 7.1: Entropy and mass deviation for different thresholding methods on the first function, LGT5/3 wavelets, and hard thresholding.

Thresholding	Entropy			Relative error		Max error
	Before DWT	After DWT	After threshold	eq. 7.38	eq. 7.39	
none	6.51	0.24	0.24	$-2.22e^{-16}$	0	$1.11e^{-15}$
0.0001	6.51	0.24	0.24	$4.44e^{-16}$	$4.44e^{-16}$	$1.28e^{-4}$
0.001	6.51	0.24	0.23	$4.44e^{-16}$	$4.44e^{-16}$	$1.11e^{-3}$
0.01	6.51	0.24	0.20	$4.44e^{-16}$	$4.44e^{-16}$	$9.96e^{-3}$
0.1	6.51	0.24	0.20	$4.44e^{-16}$	$4.44e^{-16}$	$1.76e^{-2}$
1.0	6.51	0.24	0.20	$4.44e^{-16}$	$4.44e^{-16}$	$1.76e^{-2}$
all	6.51	0.24	0.20	$4.44e^{-16}$	$4.44e^{-16}$	$1.76e^{-2}$
random	6.51	0.24	6.78	$-3.33e^{-16}$	$6.66e^{-16}$	7.98

Table 7.2: Entropy and mass deviation for different thresholding methods on the second function, LGT5/3 wavelets, and hard thresholding.

Tables 7.1 and 7.2 show the entropy and the loss for the LGT5/3 wavelets with hard thresholding. The tables for the CDF9/7 and lifted Haar wavelets are shown in Appendix 11.3. The *random* row corresponds to setting all the details to a random value following a normal distribution with no relation to the original data. It is important to note that the *random* row does not correspond to an actual threshold, but rather serves as an extreme case where the information of the details is completely lost.

The maximum error is computed as the maximum absolute difference between the original data and the reconstructed data, while the average error is computed using two different methods. The first method corresponds to the relative difference between the mass of the original data and the mass of the reconstructed data:

$$\text{Mass deviation} = \frac{\sum_{i,j} f_{\text{reconstructed}}(x_i, y_j)}{\sum_{i,j} f(x_i, y_j)} - 1. \tag{7.38}$$

The second method is similar, but a weight of $\frac{1}{2}$ is applied to the borders:

$$\text{Mass deviation} = \frac{\sum_{i,j} \omega_{i,j} f_{\text{reconstructed}}(x_i, y_j)}{\sum_{i,j} \omega_{i,j} f(x_i, y_j)} - 1, \tag{7.39}$$

where $\omega_{i,j}$ is a weight that is defined as:

$$\omega_{i,j} = p_i \cdot q_j, \tag{7.40}$$

$$p_i = \begin{cases} \frac{1}{2} & \text{if } i = 0 \text{ or } i = x_{\max}, \\ 1 & \text{otherwise,} \end{cases} \tag{7.41}$$

$$q_j = \begin{cases} \frac{1}{2} & \text{if } j = 0 \text{ or } j = y_{\max}, \\ 1 & \text{otherwise,} \end{cases} \tag{7.42}$$

where $x_{\max}$ and $y_{\max}$ are the maximum indices in the x and y directions, respectively. This second method is used to verify that the mass conservation property, as formulated in equation 7.17, is respected. This formulation is only valid for the LGT5/3 and CDF9/7 wavelets, as the lifted Haar wavelets are constructed differently. In Appendix 11.3, we verify that the mass conservation property does not hold for the lifted Haar wavelets with this formulation. Note that the corners use a weight of $\frac{1}{4}$, due to the fact that they are shared by two borders.

Different conclusions can be drawn from the tables. The first function demonstrates the benefits of performing the thresholding, as the entropy is reduced by a factor of approximately 4. The second function, however, does not show the same benefits, as the entropy is not subsantially reduced. This is presumably due to the highly smooth nature of the function, which is perfectly captured by the wavelet scheme. We can see this from the fact that the entropy does not substantially change whether no thresholding is applied or all the details are thresholding. The maximum error also remains relatively low, even when all the details are thresholded.

All the computations are performed in double precision, the mantissa of which is 52 bits long ($\approx$ 16 decimal digits). The relative error reamins of the order of machine precision, which indicates that mass

conservation is achieved in practice. The slight deviations can reasonably be attributed to rounding errors. The fact that, even with randomly set details, the mass deviation remains low further supports this claim. Let us note that the rounding errors can be caused by the wavelet lifting scheme, which explains why even no thresholding can lead to a small mass deviation.

In principle, the mass conservation for the LGT5/3 wavelets should be computed with regards to formula 7.39. However, since the border values of the 1D wavelet transform remain unchanged, the formula 7.38 is equivalent to formula 7.39 . A proof of this is provided in Appendix 11.3.

Finally, the maximum error appears to be correlated with the threshold value. This is expected, as the thresholding controls the amount of information that is lost. Setting an optimal threshold value is a trade-off between the compression ratio and the loss. There are no generic methodological guidelines to set the threshold value. In particular for our purposes, where the loss in one compression step can impact the next simulation steps. However, we rely on the assumption that the threshold values can be set empirically most of the time.

In summary, we have shown that the wavelet scheme can be combined with thresholding to achieve lower entropy and presumably higher compression ratios. To achieve effective compression, the whole process must be combined with a lossless compression method. In the next section, we will discuss about the practical aspects of wavelet-based compression schemes.

7.3.3 Compression methodology

The application of wavelet transform coupled with thresholding has demonstrated its capacity to effectively lower the entropy of data. Following the framework of Shannon, a reduction in entropy signifies a decrease in information quantity, essential for achieving data compression. This section will explore the potential of integrating the wavelet transform in a compression scheme, leveraging the reduced entropy to achieve high compression ratios.

Since the entropy of the data has been reduced, simply using an existing entropy-based compression algorithm, such as Huffman coding or arithmetic coding, would be sufficient to achieve compression. These methods are generally not designed to handle floating-point numbers, but the DWTs and the thresholding step have made the data more convenient to compress. Hence, we can directly use these methods on the resulting data and expect high compression ratios.

After the thresholding step, the data are expected to be sparse, with most of the coefficients being zero. This observation leads to the idea of using sparse storage formats instead of classical lossless compression algorithms. Multiple sparse storage formats exist, such as the Compressed Sparse Row (CSR) format, the Compressed Sparse Column (CSC) format, and the Coordinate (COO) format. These formats are designed to store sparse matrices efficiently, and can be extended to store sparse tensors. It is important that these sparse storage methods can constitute a compression method only because the thresholding step has made the data sparse.

Wavelet type	**Original**	**LGT5/3**		**CDF9/7**		**Haar**	
Thresholding	**function**	**Hard**	**Soft**	**Hard**	**Soft**	**Hard**	**Soft**
Entropy	7.22	$3.53e^{-1}$	$3.53e^{-1}$	$3.82e^{-1}$	$3.81e^{-1}$	$3.59e^{-1}$	$3.58e^{-1}$
Max error	0	$1.36e^{-3}$	$4.20e^{-3}$	$1.73e^{-3}$	$5.08e^{-3}$	$1.56e^{-3}$	$6.08e^{-3}$
Sparsity	0%	96.8%	96.8%	96.1%	96.1%	96.7%	96.7%
Compresion ratio (COO)	x0.76	x24.07	x24.07	x19.63	x19.63	x23.34	x23.34
Compresion ratio (zlib)	x1.04	x29.95	x27.85	x24.46	x14.90	x28.62	x25.60
Compresion ratio (bz2)	x1.04	x36.88	x34.72	x28.31	**x21.99**	x32.27	x30.17
Compresion ratio (lzma)	**x1.58**	**x42.79**	**x38.71**	**x31.62**	x21.76	**x34.76**	**x31.58**

Table 7.3: Various metrics for the different compression methods after applying the DWT 3 times and applying a threshold of 0.001 on the function 1. Each column corresponds to a different type of wavelet and thresholding method. The first column corresponds to the original function. The best compression ratio for a wavelet type and thresholding method is emphasized in bold.

Wavelet type	**Original**	**LGT5/3**		**CDF9/7**		**Haar**	
Thresholding	**function**	**Hard**	**Soft**	**Hard**	**Soft**	**Hard**	**Soft**
Entropy	6.51	$2.32e^{-1}$	$2.28e^{-1}$	$2.09e^{-1}$	$2.08e^{-1}$	$2.21e^{-1}$	$2.16e^{-1}$
Max error	0	$1.11e^{-3}$	$2.71e^{-3}$	$1.37e^{-3}$	$2.16e^{-3}$	$2.26e^{-3}$	$2.67e^{-3}$
Sparsity	0%	97.9%	97.9%	98.2%	98.2%	98.1%	98.1%
Compresion ratio (COO)	x0.76	x36.03	**x36.03**	x43.10	**x43.10**	x39.27	**x39.27**
Compresion ratio (zlib)	x1.16	x47.34	x17.70	x57.15	x17.27	x51.96	x17.10
Compresion ratio (bz2)	x1.15	x63.84	x29.74	x78.16	x31.36	x72.40	x30.78
Compresion ratio (lzma)	**x1.82**	**x77.12**	x31.35	**x95.00**	x32.03	**x85.86**	x31.24

Table 7.4: Various metrics for the different compression methods after applying the DWT 3 times and applying a threshold of 0.001 on the function 2. Each column corresponds to a different type of wavelet and thresholding method. The first column corresponds to the original function. The best compression ratio for a wavelet type and thresholding method is emphasized in bold.

Tables 7.3 and 7.4 present the compression ratios and the maximum error across various wavelet types and thresholding techniques, leveraging four distinct lossless compression approaches: COO format, Deflate, Burrows-Wheeler, and Lempel-Ziv-Markov chain algorithms. The latter 3 methods are run thanks to the zlib, bzip2, and lzma libraries, respectively. The COO format is a sparse storage format that stores that data as a list of tuples (i, j, v), where i and j are the indices of the non-zero values, and v is the value of the non-zero value. For this experiment, we did not use actual COO storage, but computed the size according to the number of non-zero values (assuming 64 bits for v and 2×10 bits for i and j). The Deflate algorithm, on which zlib is based on, merges LZ77 with Huffman coding [311, 201]. The bzip2 implementation of the Burrows-Wheeler transform alongside Huffman coding [292, 5]. Lastly, the LZMA algorithm, powered by lzma, uses the Lempel-Ziv-Markov chain technique [216].

Regarding the entropy, this table lets us compare the achieved entropy reduction of all the wavelet types and thresholding techniques. The LGT5/3 and lifted Haar wavelets have similar entropy reductions, while the CDF9/7 wavelets appear to behave differently. The first function achieves less entropy reduction with the CDF9/7 wavelets (compared to LGT5/3 and lifted Haar), while the second function achieves more entropy reduction. This is explained by the clear discontinuity that is present in the first function that impacts more details with the CDF9/7 wavelets, which have a larger support. The thresholding method (hard or soft) does not appear to impact the entropy reduction significantly. However, using soft thresholding consistently results in an increase of the maximum error. Hence, in the context of CFD simulations, where the error is more important than the visual quality, hard thresholding should be preferred.

Finally, the compression ratios achieved in the different cases provide insights into the potential memory gains that can be achieved in practice. The point of this study is not to compare the lossless compression methods, but rather to show that they all achieve high compression ratios. It is also interesting to see that the COO format, which is not a generic compression algorithm, appears to be competitive with the other methods. This is because the sparsity of the data, which is the proportion of zero values, is particularly high.

On the first function, the LGT5/3 wavelets appear to be more compressive than the CDF9/7 and lifted Haar wavelets, while they are the least compressive on the second function. The second function demonstrate the strength of the higher filtering order of the different wavelets. As it does not possess any clear discontinuity, the best compression ratios are achieved with the CDF9/7 wavelets (third order filtering), followed by the lifted Haar wavelets (second order filtering), and then the LGT5/3 wavelets (first order filtering). The first function, however, demonstrates that the discontinuities are captured more efficiently by the LGT5/3 wavelets, as they achieve the best compression ratios.

Globally, the compression ratios achieved with hard thresholding are higher than those achieved with soft thresholding. This is counter-intuitive, as soft thresholding appears to tend to achieve slightly higher entropy reductions (which should lead to higher compression ratios). We believe that this is due to the floating-point representation, which makes the near-zero region denser. Intuitively, two values above a given threshold are more likely to be the same than two values below the threshold. Consequently hard thresholding appears to be better for our purposes, as it consistently achieves higher compression ratios and lower maximum errors.

Overall, the compression ratios achieved thanks to this compression scheme are high, with minimal

impact on the precision of the data. While various lossless compression algorithms could further compress the output, this study focuses on showcasing the wavelet transform as an efficient compression tool for CFD simulations. There is promising potential for the development of new lossless compression techniques tailored for optimal performance in CFD simulations within this framework. In concluding this study, the ensuing section will discuss the practical advantages and potential limitations of the proposed compression approach.

7.4 Discussions

The compression method discussed in this chapter shares its conceptual foundation with the JPEG2000 standard. Yet, our method diverges from this standard to better address the unique challenges of CFD simulations. These challenges not only encompass the numerical properties outlined in Section 7.2.1 but also the overall approach to compression, which can be tailored to match implementation-specific requirements. Throughout this chapter, we have explored various wavelets and conducted targeted testing with the goal of identifying the most effective strategies for practical application in CFD simulations.

In our evaluation, we focused on the maximum absolute error as a measure of loss, finding it to be particularly suited for CFD simulations. Many numerical schemes in CFD guarantee a specific accuracy level up to a certain error threshold. If the maximum error introduced by compression remains below this threshold, the simulation is likely to retain its accuracy. This is in contrast to other metrics, such as the mean squared error, which may overlook the significance of localized errors concentrated in small areas of the simulation.

When comparing thresholding methods, hard thresholding demonstrated superior performance over soft thresholding, delivering higher compression ratios and reduced maximum errors. Consequently, future discussions will concentrate on the application of hard thresholding. Additionally, our examination of various wavelets revealed significant differences in their properties and impacts. The LGT5/3 and CDF9/7 wavelets necessitate grid sizes that are either powers of 2 plus 1 or exact powers of 2, respectively. Given that most computational hardware is optimized for grid sizes that are powers of 2, this requirement could notably influence the efficiency of the wavelet compression scheme, in one way or another. Furthermore, in one-dimensional cases, both LGT5/3 and CDF9/7 wavelets are designed to preserve edges, a feature that, when applied in multi-dimensional simulations, means that edges (or faces in 3D) can be accessed with one fewer DWT. Unfortunately, a similar property could not be achieved with lifted Haar wavelets, highlighting a limitation in their design that could make them less efficient in future implementations that would leverage this property.

The main difference between the LGT5/3 and CDF9/7 wavelets lies in their respective supports. The CDF9/7 wavelets have a larger support and assure filtering up to the third order, a feature that the LGT5/3 wavelets, which guarantee filtering up to the first order, do not possess. However, this enhanced filtering capability of the CDF9/7 does not necessarily translate into significantly improved compression ratios in real-world applications. Moreover, the larger support of the CDF9/7 wavelets diminishes their efficiency in capturing discontinuities. Given the prevalence of discontinuities in CFD data, this makes the LGT5/3 wavelets a more fitting choice for our purposes. However, it should be acknowledged that specific scenarios could present optimal conditions for the CDF9/7 wavelets (or lifted Haar wavelets).

The significant advantage of adopting the wavelet compression approach lies in its ability to achieve high compression ratios while having a minimal impact on the precision of the data. This is especially true in three-dimensional simulations, where compression ratios are anticipated to increase cubically with the grid size (per dimension), while a quadratic growth is expected for two-dimensional simulations. Therefore, three-dimensional CFD simulations are expected to benefit from extremely high memory savings through this compression method, offering a promising solution for managing the large volumes of data generated by such simulations.

Nevertheless, the adoption of this compression method is not without its concerns, primarily due to the potential adverse effects of loss introduced by thresholding. In the realm of CFD, evaluating the quality of a simulation based solely on the error from a single compression cycle is inadequate. This is because the cumulative effect of errors introduced by successive compression cycles can significantly impact subsequent simulation steps, potentially leading to a detrimental feedback loop that undermines the integrity of the entire simulation. Consequently, our method necessitates a careful investigation into identifying an optimal

threshold value for a given simulation. Ideally, this value would as high as possible, given a constraint imposed on the simulation accuracy.

Another aspect warranting careful consideration is the computational cost associated with the compression process. Specifically, for DWTs applied in a single dimension, a minimum of $2N$ memory accesses is required, accommodating one read and one write operation for each coefficient. With successive applications of DWTs, the initial DWT necessitates $2N$ memory accesses, followed by N accesses for the second DWT, $N/2$ for the third, and so on, cumulatively leading to an asymptotic total of $4N$ memory accesses. When DWTs are applied across multiple dimensions, this figure multiplies accordingly. For instance, in three-dimensional applications, it is plausible for each coefficient to be accessed up to 12 times, representing a considerable computational demand. While strategies to reduce the number of memory accesses in multidimensional scenarios exist, they invariably complicate the data access pattern [260, 163]. This complexity can adversely affect overall system performance, particularly in the context of GPU computing, where memory access patterns are crucial for achieving optimal performance. Despite these challenges, the compelling compression ratios achieved through this method justify its utilization, especially when the available memory is significantly less than the demands of the simulation.

In the next chapter, we delve into the practical aspects of integrating wavelet compression into existing CFD simulations. This chapter focuses on the numerical aspects of the integration, as well as practical considerations for implementing the wavelet compression scheme on GPUs. The goal is to show that the wavelet compression scheme is not only theoretically sound but also practically feasible, offering a viable solution for managing the large volumes of data generated by CFD simulations.

Chapter 8

Integrating wavelets into CFD simulations

In the preceding chapter, the application of wavelets for compressing multi-dimensional data was examined. It was observed that wavelets could achieve exceptionally high compression ratios with minimal impact on the visual quality of the data. Additionally, careful design of the wavelet transform can ensure that the mass-conservation property of the data is preserved, an important consideration in physical simulations. However, the relationship between the visual quality of data and the accuracy of physical simulations is not directly evident. This chapter aims to show that wavelet transforms can be utilized to CFD data while minimally impacting the accuracy of simulations. The content and findings herein are based on the work published at the CEMRACS 2022 edition [97].

As in the rest of this document, we consider the CFD data as a regular multi-dimensional grid of floating-point values and associate a time step with a stencil operation. The goal of this work is to demonstrate how the wavelet transform can be used in practice to compress CFD data. Even though the execution is not distributed, we adopt the framework described in Section 3.3 to be able to extend the method to distributed simulations in the future. We, hence, remain as general as possible to show that the principles of the method are applicable to a wide range of CFD simulations.

We begin this study by providing implementation details of the compression algorithm and the simulation framework in Section 8.1. We then perform a series of experiments on a simplistic 2D transport simulation to evaluate the impact of the compression on the simulation in Section 8.2. Finally, in Section 8.3, we test our method on a more realistic 2D Saint-Venant (shallow water) simulation to demonstrate the applicability of the method to more complex simulations and evaluate the performance impact of the compression.

8.1 Description of the framework

The compression/decompression algorithms have been extensively discussed in the previous chapter. However, these algorithms alone do not provide a complete solution reaching effective memory gains. In this section, we present the framework that we use to integrate the compression algorithm into a simulation, as well as various technical details.

8.1.1 Workflow

To integrate the wavelet compression algorithm into a simulation, we define a framework that we describe in this section. It is important to note that the used framework is designed for simplicity rather than performance. The main goal of this work is to assess the numerical impact of the compression algorithm on the simulation quality.

As we have done until now, we divide the simulation grid into subgrids. Each subgrid can be processed individually and includes ghost cells that duplicate the values of the neighboring subgrids and require synchronization between time steps.

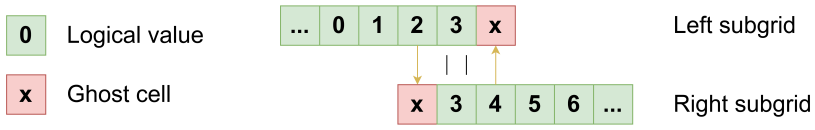


Figure 8.1: This schematic shows the synchronization process in 1 dimension. The logical space represents the cells that require computation, while the ghost cells are used to store the values of the neighboring subgrids. The ghost cells are updated by copying the corresponding values of the neighboring subgrids. One value is shared between both subgrids and does not need to be updated. The yellow arrows show which values are copied. In a real-case scenario, the data consist of floating-point values, rather than integers. In multiple dimensions, the synchronization process is performed successively in each dimension.

In this work, the approach is slightly modified to account for the fact that the wavelet transform might not be conservative at the boundaries of the space they are applied to. Section 7.3.2 discusses this concern and shows that our LGT5/3 and CDF9/7 wavelet transforms are actually also conservative at the boundaries. However, this was not clear at the time of the implementation. Hence, the edge values of the subgrids are represented and computed twice, in a fashion depicted in Figure 8.1. On this figure, the value "3" is shared between two subgrids and is never updated. The left subgrid is responsible for conserving half of its values, while the right subgrid is responsible for conserving the other half (see equation 7.39), reaching a global conservativeness of the scheme.

```
const directions = {(-1, 0), (1, 0), (0, -1), (0, 1)};

// Repeat until we reach tmax
for(float t=0;t<tmax;t+=dt) {
    // For each subgrid
    for (int i = 0; i < subgrids_number; ++i) {
        decompress(subgrids[i], subgrids_compressed[i]); // Decompress the subgrid

        // Write the ghost cells from all directions
        for(int dir=0;dir<4;dir++) {
            float *target_interface = get_interface(i, directions[dir]);
            interface_to_subgrid<<<...>>>(subgrids[i], target_interface, directions[dir]);
        }

        LBM_step(subgrids[i], subgrid_buffer); // Perform an LBM step
        swap(subgrids[i], subgrid_buffer); // subgrids[i] becomes the result and subgrid_buffer
    is the new buffer

        // Write the edge values
        for(int dir=0;dir<4;dir++) {
            float *target_interface = get_interface(i, directions[dir]);
            subgrid_to_interface<<<...>>>(subgrids[i], target_interface, directions[dir]);
        }

        compress(subgrids[i], subgrids_compressed[i]); // Compress the subgrid
    }
}
```

Code 8.1: Pseudo-code for running a simulation with the compression algorithm

Code 8.1 shows the overall process of the simulation. It relies on 3 main memory segments: the subgrids (decompressed), the subgrids compressed, and the interfaces (masked by the use of `get_interface`). The idea is go successively through each subgrid, decompress it, perform the LBM step, and compress it back. To synchronize the subgrids, the `subgrid_to_interface` and `interface_to_subgrid` functions are interleaved and store/retrieve the edge values of the subgrid in/from the interfaces. Note that the intermediate buffers are necessary to store the state of the neighboring subgrids *before* the (lossy) compression. If the values were taken from the neighbors after the compression (or decompression), the mass would not be conserved anymore.

This approach has multiple flaws. One of them is that the use of the `subgrids` array is not necessary, as the `subgrids_compressed` array could be used for "long-term" storage. Because of this, this algorithm is unable to achieve effective compression, as the subgrids are stored in memory in an uncompressed form.

However, this allows for a straightforward implementation of the algorithm and a clear demonstration of the impact of the compression on the simulation quality. The next chapter is more focused on reaching effective memory gains and includes an algorithm that is more elaborate and efficient. For the purpose of the current chapter, it is acceptable to use a simple algorithm, as long as all the information of the simulation is compressed once per time step. In the next section, we present the compression algorithms that we use in our experiments.

8.1.2 Compression pipeline

We refer to Section 7.3.3 for a reminder of the general compression methodology we use in this work. Let us simply recall that the compression pipeline consists of the following steps:

1. Perform the wavelet transform on the subgrid;
2. Apply a thresholding on the wavelet coefficients;
3. Perform a lossless compression on the resulting data.

These steps are applied independently on each "channel" of the data (conservative or kinetic variables in in the CFD terminology). Hence, each variable is compressed independently.

In this section, the discussion is focused on two items. We begin by presenting the lossless compression methods that we use in our experiments: the Compressed Sparse Row (CSR) format and the LZ4 compression algorithm. We then discuss the implementation of the wavelet transform and the thresholding.

Lossless compression methods

The choice of the lossless compression method impacts both the compression ratio and the computational cost of the compression. We consider two lossless compression methods: the Compressed Sparse Row (*CSR*) format and the *LZ4* compression algorithm. The first one, chosen for its simplicity, is a sparse matrix representation that is well-suited for compressing sparse data. The second one, LZ4, is a fast and efficient lossless compression algorithm that is widely used in data-intensive applications.

Sparse Matrix Representation The Compressed Sparse Row (CSR) format, also known as the Yale format, is a widely used representation for sparse matrices. In this format, the non-zero elements of the matrix are stored in 3 arrays. The first array, V, contains the non-zero elements of a $m \times n$ matrix. The second array, COL, of the same size as V, contains the column indices of the non-zero elements. The third array, ROW, of size $m + 1$ contains the offset of the first non-zero element of each row. This representation can be used as a lossless compression technique since we anticipate a large number of zero values in the matrix. While the CSR format was not designed specifically for compression purposes, it is both well-known and efficient. Additionally, its compressed size is proportional to the number of non-zero elements, making it a consistent representation. With this compression method, the compression kernel is the `dense-to-CSR` kernel, while the decompression kernel is the `CSR-to-dense` kernel.

LZ4 LZ4 is a widely used lossless compression algorithm known for its high compression ratios and fast processing speeds on GPU. It has gained popularity due to its efficient CUDA implementation in the nvCOMP library, which was developed by Nvidia [229]. LZ4 is a byte-oriented algorithm that is designed to be fast and parallelizable. It was initially developed to perform well on CPUs and has since been optimized for use on GPUs [250, 74]. The algorithm uses a block-based approach and compresses each block independently, with a configurable chunk size. The chunk size determines the size of the input data that is processed at once by the LZ4 algorithm. Usually, a larger chunk size implies a better compression ratio, but a slower compression speed. LZ4 is commonly used in data-intensive applications such as scientific simulations and big data analytics.

These two methods are used in our experiments to assess the impact of the lossless compression method. Their working principles are very different. *LZ4* is a general-purpose lossless compression algorithm that is

designed to be fast and efficient, while the *CSR* format is a sparse matrix representation that is not specifically designed for compression. In the context of this work, we use external libraries to perform the compression and decompression. The LZ4 algorithm is used through the nvCOMP library, while the CSR format is used through the cuSPARSE library. It is conceivable that the `dense-to-CSR` and/or `CSR-to-dense` kernels from cuSPARSE are not well optimized because they are rarely part of the critical path of an application. However, if our experiments show that the CSR format is competitive with LZ4 in terms of compression ratio, we can consider implementing a custom CSR compression algorithm to improve the performance.

Regarding the wavelet transform and the thresholding, we need to provide a custom implementation, as we designed a custom DWT scheme. In the next section, we present our idea for implementing an efficient DWT on GPUs.

Optimized DWT on the GPU

In this section, we present challenges and solutions for implementing a 3D DWT on GPUs. This 3D DWT can be modified to work on 2D data. We assume that the data are stored in a 3-dimensional array $f_{i,j,k}$, where i, j, and k are the indices along the x, y, and z axes, respectively. Performing the DWT on the y and z axes is straightforward, as the lifting scheme can be performed directly on vectors of lines of the x axis. On the x axis, however, a coalesced access leads to the values being stored on different threads because the x dimension is contiguous in memory. A solution is to store each line (of the x axis) in the shared memory of the block and then perform the DWT within the shared memory. With this method, one read and one write need to be performed per coefficient in the global memory for the x axis, which is optimal. Code 8.2 shows the pseudo-code of the DWT on the x axis.

```
template <bool compress>
// subgrid is the input and output grid, compression_level is the number of wavelet transforms to perform
__global__ void wavelet_x(float *subgrid, int compression_level)
{
    // Shared memory for storing the line
    __shared__ float line[MAX_LINE_SIZE];

    for (int yl = blockIdx.x; yl < grid_size_y_logical; yl += gridDim.x)
    {
        int xl = threadIdx.x;

        // True x and y (xl and yl are the logical coordinates)
        int xt = xl + grid.overlap[0];
        int yt = yl + grid.overlap[1];

        if(xl < grid_size_x_logical)
        {
            // Load line from global memory
            line[xl] = subgrid[yt * grid_size_x_true + xt];
        }

        // Wait for all threads to load the line
        __syncthreads();

        // wavelet transform
        wavelet_x_on_line<compress>(line, compression_level);

        __syncthreads();

        if(xl < grid_size_x_logical)
        {
            // Write the result to the global memory
            subgrid[yt * grid_size_x_true + xt] = line[xl];
        }
    }
}
```

Code 8.2: Pseudo-code of the DWT on the x axis.

The thresholding can be performed at the last step of the DWT on the x axis: if the detail is below a certain threshold, it is written as a zero in the global memory.

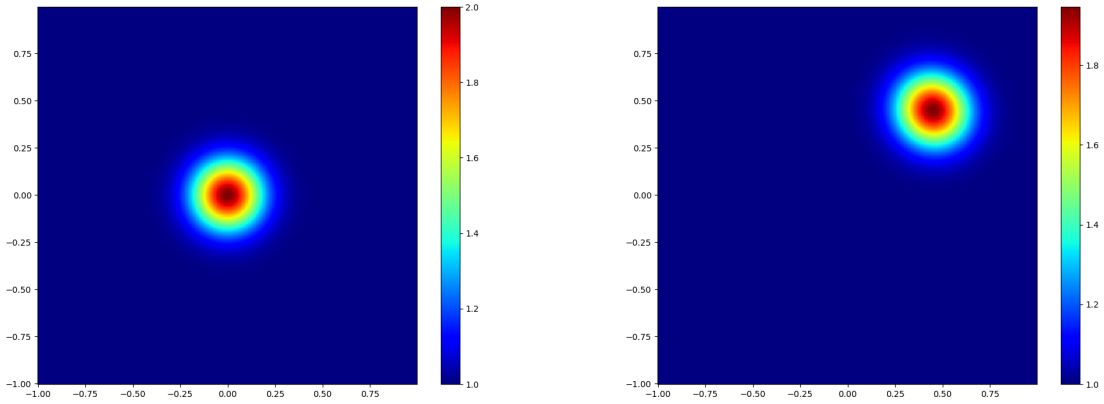
These kernels, although simple, are efficient enough for our purpose and the global memory is accessed in a coalesced way. We are aware that multi-dimensional DWTs can theoretically make better data reuse by merging the computations of the different axes, but it is not clear whether it is possible to achieve effective

gains on GPUs, due to the less regular memory access pattern. This is an interesting subject for future research that would aim to improve the performance of our compression method.

Now that we have described the compression algorithm, as well as the workflow of the simulation, we can run different experiments to assess the impact of the compression on the simulation quality. In the next sections, we present the results of these experiments.

8.2 Evaluating the method on a simple 2D transport simulation

8.2.1 Description of the scheme



(a) t=0s (b) t=0.5s

Figure 8.2: This figure shows the initial state of the simulation (left) and the exact solution of the simulation at t=0.5s for $\alpha = 0.9$ and $\beta = 0.9$ (right). The original structure is displaced with a speed of (α, β). The color represents the value of the function.

The first tested simulation is a 2D simplistic computation of the displacement of an arbitrary structure given by the following rules:

$$\begin{cases} f\_init(x,y) = 1 + e^{-30(x^2+y^2)} \\ f(x,y,t) = f\_init(x - \alpha t, y - \beta t), \end{cases} \tag{8.1}$$

where f_init is the initial state of the simulation, f is the exact solution of the simulation, and α and β are the speeds of the displacement along the x and y axes, respectively. Figure 8.2a shows f_init at t=0s. Figure 8.2b shows the exact solution at t=0.5s for $\alpha = 0.9$ and $\beta = 0.9$. To design a scheme whose exact solution is the displacement of the structure, we use a Finite Volume (FV) approach.

The objective is to solve numerically the following system of conservation laws:

$$\partial_t W + \nabla \cdot \left(W \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \right) = 0, \tag{8.2}$$

s where the unknown W is a vector of m conservative variables (here, $m = 1$) depending of the space variable $X = (x, y)$ and the time variable t. As always throughout this work, we assume that X is in the square $]0, L[\times]0, L[$, but more complex shapes are possible.

For a given two-dimensional vector $N = (N^x, N^y)$, we define the flux of the system of conservation laws by

$$Q(W, N) = W(\alpha N^x + \beta N^y), \tag{8.3}$$

with the four possible directions:

$$N^0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad N^1 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad N^2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad N^3 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}. \tag{8.4}$$

In this way:

$$Q(W, N^0) = Q^x(W), \quad Q(W, N^1) = -Q^x(W),$$
$$Q(W, N^2) = Q^y(W), \quad Q(W, N^3) = -Q^y(W).$$

To achieve the goal of having the initial structure displaced at a constant speed (α, β), we use the standard upwind flux:

$$Q(W_L, W_R, N) = W_L \max(\alpha N^x + \beta N^y, 0) + W_R \min(\alpha N^x + \beta N^y, 0). \tag{8.5}$$

The system of conservation laws (8.2) is then approximated by the following FV scheme, which allows computing the value at time step $n+1$ from that of the time step n

$$W_{i,j}^{n+1} = W_{i,j}^n - \frac{\tau}{\Delta x} \sum_{k=0}^{3} Q(W_{i,j}, W_{i',j'}, N^k), \quad \text{with} \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} + N^k, \tag{8.6}$$

where τ is the time step and Δx is the space step. In this formula, the numerical flux $Q(W, W', N)$ approximates the flux $Q(W, N)$ at the interface between two cells. The numerical flux has to satisfy some mathematical property in order to ensure a stable and accurate approximation. It is out of the scope of this work to discuss this aspect. We refer (for instance) to [93, 157].

This scheme can be implemented as follows:

```
for(int i = 1; i < n_x - 1; i++) {
    for(int j = 1; j < n_y - 1; j++) {
        W_next[i][j] = W_now[i][j];
        for(int dir = 0; dir < 4; dir++) {
            W_next[i][j] -= tau/dx * fluxnum(W_now[i][j],
                                             W_now[i+N[dir][0]][j+N[dir][1]],
                                             N[dir]);
        }
    }
}
```

Code 8.3: Pseudo-code of the transport kernel.

Where `fluxnum` is a function implementing the numerical flux and `N` is the array of the normal vectors of the cells defined in (8.4).

Note that indices i=0, j=0, i=n_x-1, and j=n_y-1 can be omitted as they are part of the ghost cells and will be overwritten by the neighboring subgrids during the synchronization phase. If a single grid is used, a modulo operation can be used to handle the edge cells. We recognize the algorithmic structure of a stencil operation, where the stencil consists of the 4 neighboring cells and the cell itself.

8.2.2 Methodology

In this section, we present the methodology used for the experiments, as well as various technical details.

The 2D scheme is applied on each z-slice of the grid, resulting in a 3D simulation. The reason for this choice is to have a simple simulation that can be easily visualized in 2D, while still being processed in 3D. The values are, therefore, constant across the z-axis. Hence, to avoid unfair comparisons, the wavelet transform is only applied on the x and y axes.

The simulation domain is discretized into a $128 \times 128 \times 128$ grid, divided into $2 \times 2 \times 2$ subgrids of (logical) size $65 \times 65 \times 65$. The logical size is the size of the subgrids that are processed for the LBM simulation and does not include the ghost cells. The bordering values of the logical space are shared between multiple subgrids. In Figure 8.1, the shared value is the cell "3". As these values are not synchronized, the same computation must be performed once per subgrid. The reason for introducing this overlap is to keep the mass-conservation property at the scale of the global grid (as explained in Section 8.1).

The benchmark program has been written in CUDA and compiled with the nvcc compiler and the `-O3 -use_fast_math` flags. We run the program on an NVIDIA Tesla V100 GPU with 16GB of memory.

We refer to the following program parameters:

- the number of performed wavelet transforms (i.e. the *compression level*);
- the threshold value (value below which the coefficients are set to zero);
- the used lossless compression (LZ4 or CSR);
- in the case of LZ4, the chosen chunk size. A lower chunk size leads to a lower compression rate, but a higher compression speed;
- the (scheme-wise) simulation time (related to the number of time steps and the grid size).

We set two measures of interest: the effective compression ratio and the quality of the simulation. The effective compression ratio is the initial data size divided by the data size after the compression. The quality of the simulation is measured by comparing the results of the simulation with the exact solution. It is measured with the L2 error against the exact solution at a given time step with the formula:

$$L2\_error = \frac{SizeX \times SizeY \times SizeZ}{NX \times NY \times NZ} \times \sum_{i=0}^{NX-1} \sum_{j=0}^{NY-1} \sum_{k=0}^{NZ-1} (f\_sim(i,j,k) - f\_exact(i,j,k))^2. \tag{8.7}$$

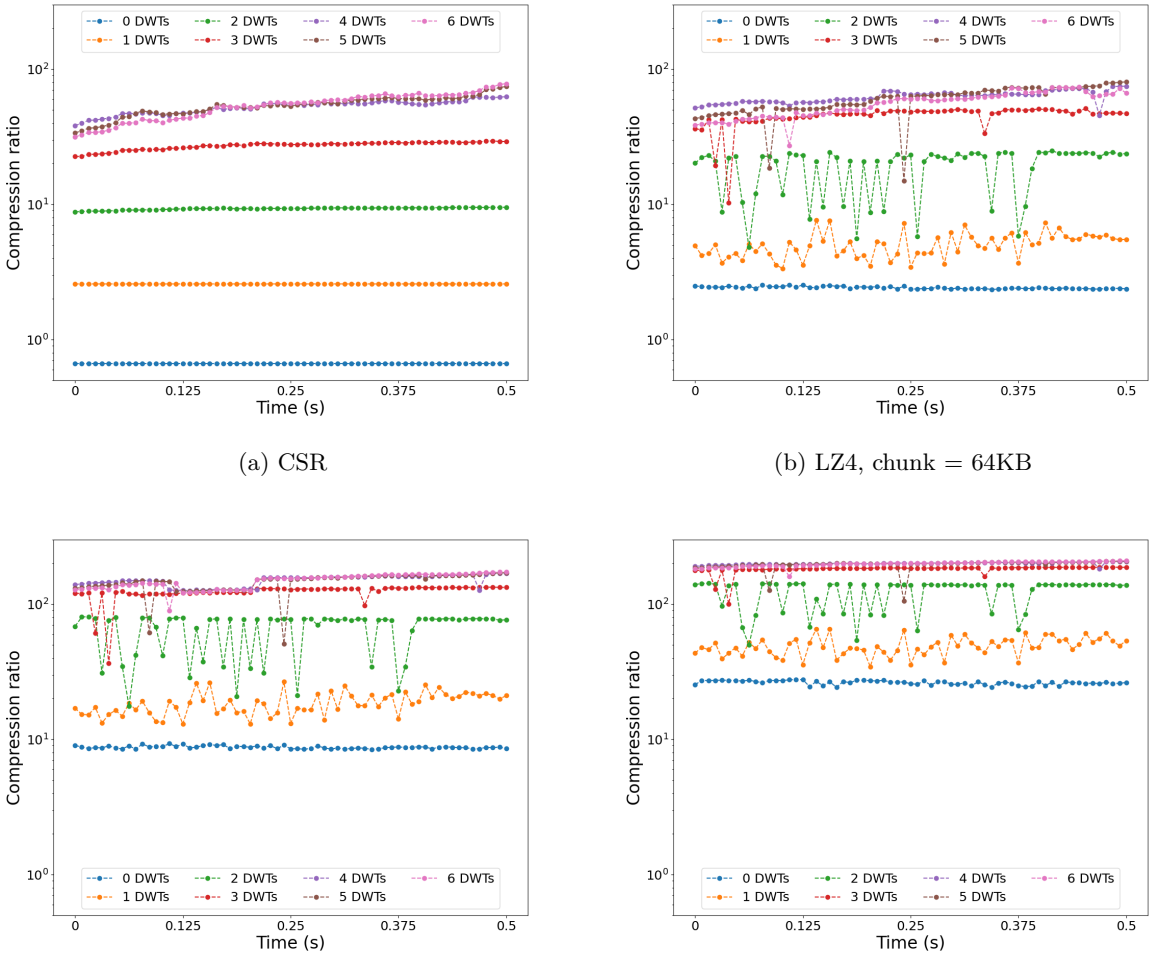
The central aim of this research is to assess the impact of various parameters on both the compression ratio and the quality of the simulation. Section 8.2.3 details the compression ratios achieved at each time step throughout the simulation process. Following this, Section 8.2.4 explores how different threshold values influence the overall compression ratio. Finally, in Section 8.2.5, the simulation quality under varying compression parameters is evaluated, providing a comprehensive analysis of how these factors interplay to affect the outcomes of the simulation.

8.2.3 Compression ratio during the simulation

In this section, we show the compression ratios during the simulation with different lossless compression methods. The tested compression methods include CSR (Figure 8.3a), LZ4 with a chunk size of 64 KB (Figure 8.3b), LZ4 with a chunk size of 256 KB (Figure 8.3c), and LZ4 with a chunk size of 1 MB (Figure 8.3d). A threshold value of 0.01 is used, and $\alpha = 0.9$, $\beta = 0.9$, $CFL = 0.45$ are chosen for the numerical simulation. The compression ratio, which is defined as the size of the uncompressed data divided by the size of the compressed data, is not constant during the simulation and tends to increase as the simulation progresses. This trend becomes more pronounced as the compression level increases, indicating that the data become more easily compressible as the simulation progresses. This is likely because each time step allows for removing more and more details from the simulation. An expected observation is that the compression ratio tends to increase with the compression level. Compression levels 5 and 6 are an exception to this trend but we can reasonably exclude them from the analysis, as they have too few sampling points to perform a reasonable DWT.

There are substantial differences between the CSR and the LZ4 compression methods. The CSR method produces smooth compression ratios over the simulation, while the LZ4 method produces more erratic compression ratios. For the CSR method, this is explained by the fact that the compression ratio is directly dependent on the number of non-zero values which have no reason to brutally change from one time step to another. For the LZ4 method, this is likely due to an inner mechanism of the LZ4 algorithm that makes it underperform in some time steps. The obtained compression ratios are comparable for the CSR method and the LZ4 method with a chunk size of 64 KB. However, the LZ4 method with chunk sizes of 256 KB and 1 MB both outperform the CSR method.

In terms of compression ratio, LZ4 with a chunk size of 1 MB is the best method. The CSR format is not designed to be a compression method. It is, therefore, not surprising that it can be outperformed. On the other hand, having a larger chunk size typically leads to increased compression ratios but at the cost of increased computation time. Finding the right balance between compression ratio and computation time is a challenge that we will address in the future.



(a) CSR

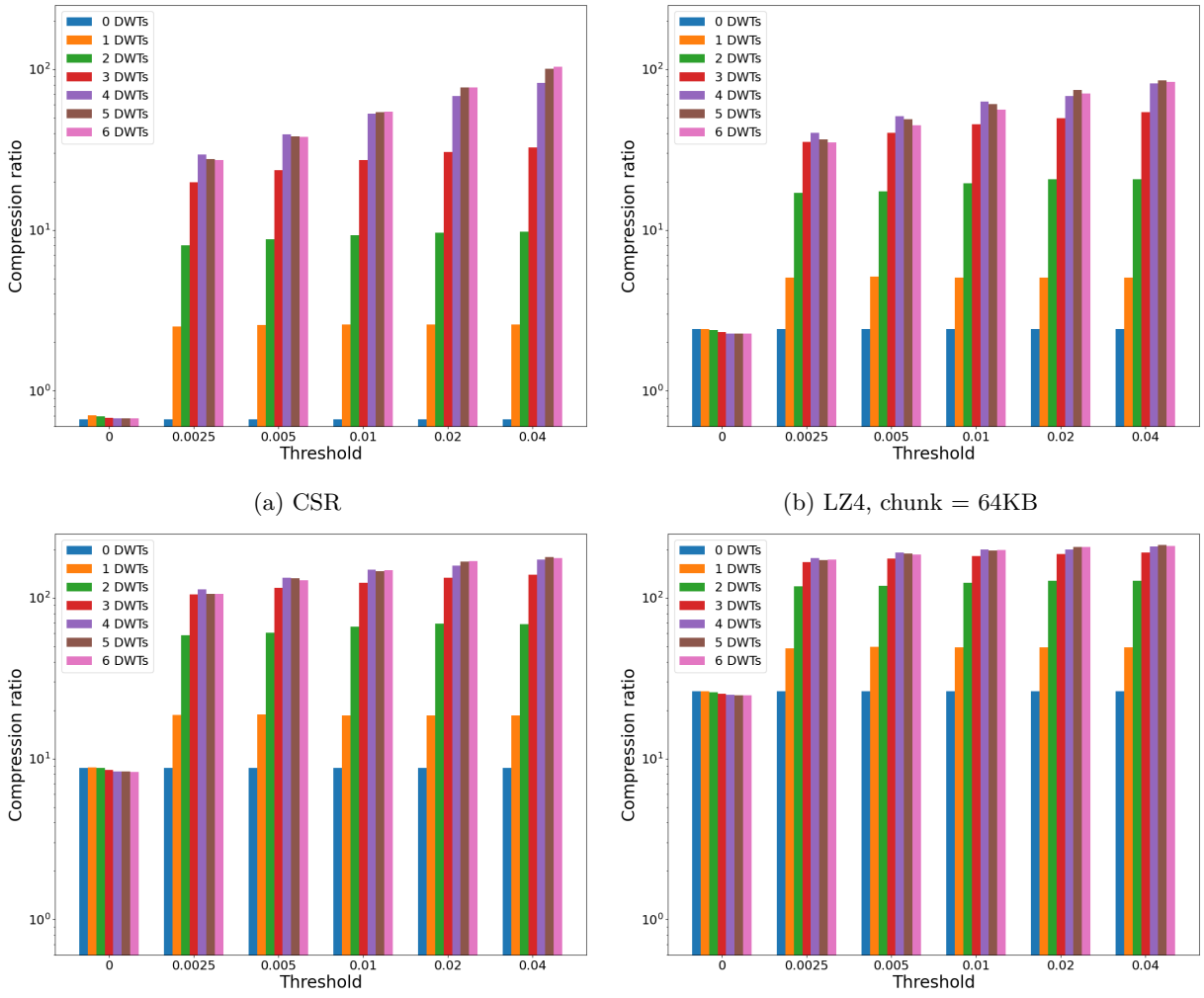
(b) LZ4, chunk = 64KB

(c) LZ4, chunk = 256KB

(d) LZ4, chunk = 1MB

Figure 8.3: This figure shows the compression ratios at each time step of the simulation with different lossless compression methods. The different curves (compression levels) represent the number of performed wavelet transforms.

8.2.4 Impact of the threshold value on the compression ratio



(a) CSR

(b) LZ4, chunk = 64KB

(c) LZ4, chunk = 256KB

(d) LZ4, chunk = 1MB

Figure 8.4: This figure shows the average compression ratio for different threshold values and lossless compression methods. The compression level is the number of performed wavelet transforms.

In this section, we aim to determine the impact of the threshold value on the compression ratio. We keep the previous simulation parameters: $\alpha = 0.9$, $\beta = 0.9$, and $CFL = 0.45$. We test the following threshold values: 0, 0.0025, 0.005, 0.01, 0.02, and 0.04. We show the results for 4 lossless compression methods: CSR (Figure 8.4a), LZ4 with a chunk size of 64 KB (Figure 8.4b), LZ4 with a chunk size of 256 KB (Figure 8.4c), and LZ4 with a chunk size of 1 MB (Figure 8.4d).

We observe that as the threshold value increases, the compression ratio generally increases as well. This trend is more pronounced at higher compression levels. At compression level 1, the compression ratio remains constant across non-zero threshold values, whereas at compression level 4, the compression ratio always increases with increasing threshold value. Additionally, we note that, up to compression level 4, the compression ratio tends to increase as the compression level increases. The four compression methods exhibit similar trends, although they differ in scale. CSR and LZ4 with a chunk size of 64 KB show similar performance, while LZ4 with a chunk size of 256 KB and 1 MB outperform the other two methods. With the 1 MB chunk size, the average effective compression ratio always reaches more than 100x if the threshold value and the compression level are greater than or equal to 0.0025 and 2, respectively.

We find that setting a non-null threshold value is crucial for achieving a high compression ratio. Furthermore, the first three compression levels have the most significant impact on the compression ratio in our case. We also note that while LZ4 compression without thresholding can already achieve a high compression ratio, the compression ratio significantly improves as the threshold value increases from 0 to 0.0025. This outcome is expected for CSR compression, where the compression ratio is directly related to the number of non-zero values. For LZ4 compression, thresholding the data increases the frequency of the "zero" symbol, resulting in a higher compression ratio. Nonetheless, LZ4 can still achieve a high compression ratio even without thresholding the data, with the compression ratio being greater than 2x for all chunk sizes. The 256 KB chunk size achieves an average compression rate of nearly 9x, while the 1 MB chunk size nearly achieves a compression rate of more than 25x, both with no pre-processing (i.e., threshold value and compression level of 0).

8.2.5 Quality of the simulation

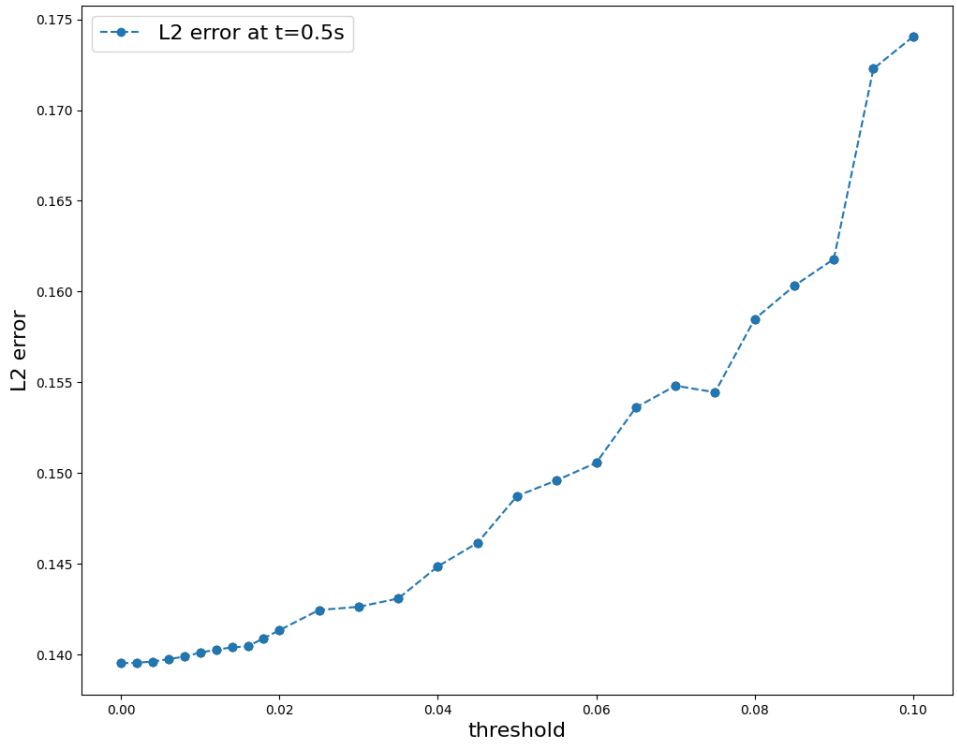
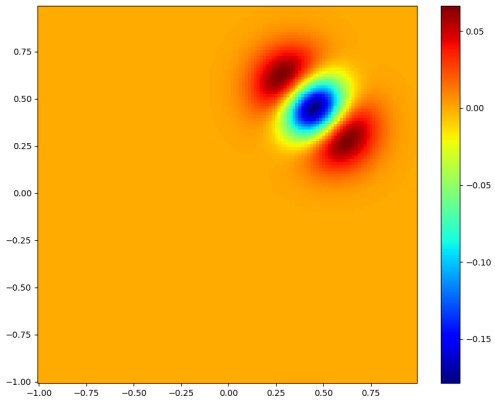
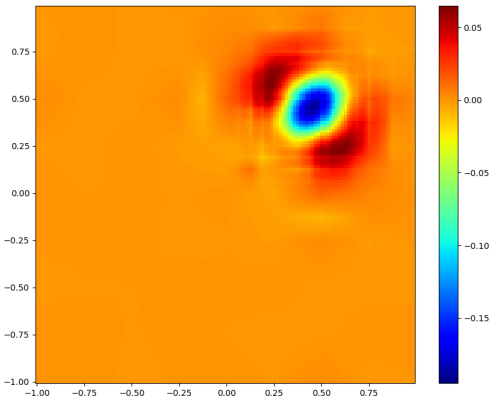


Figure 8.5: L^2 error (formula 8.7) for different threshold values at compression level 4 and t=0.5s, with $\alpha = 0.9$, $\beta = 0.9$, and $CFL = 0.45$.

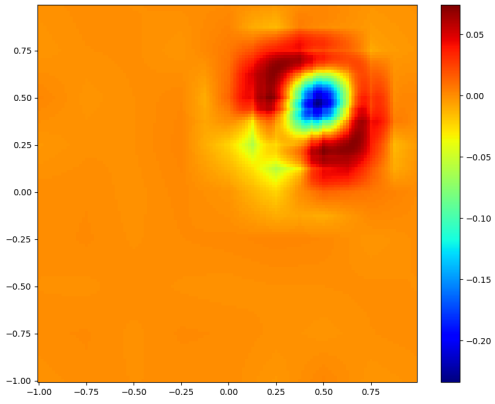
The accuracy of the simulation is evaluated by comparing it with the exact solution. To measure the error, we use the L^2 norm, which is calculated at the end of the simulation when the time is 0.5 seconds. We plot the global L^2 error depending on different threshold values (Figure 8.5), and the results show that the error tends to increase as the threshold value increases. This is expected, as the thresholding phase introduces an error in the signal reconstruction. This increase is slow for threshold values below 0.02. For higher values the L^2 increases faster. This indicates that the compression error is negligible, compared to the numerical scheme errors, for threshold values below 0.02.



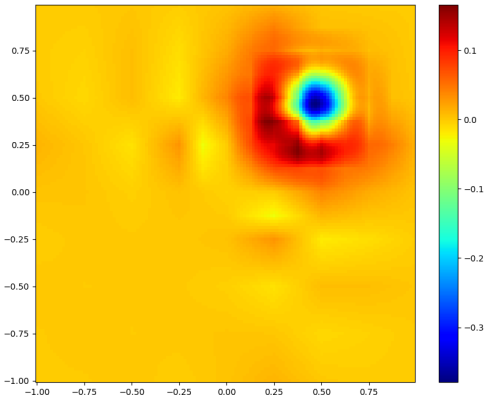
(a) threshold=0



(b) threshold=0.01



(c) threshold=0.02



(d) threshold=0.05

Figure 8.6: Distribution of the error over the domain at $t = 0.5s$ and with a compression level of 4 for different threshold values.

We also plot the difference between the simulation result and the exact solution for each point in the domain in (Figure 8.6). This plot confirms the above interpretation: the compression error starts to dominate the scheme error for a threshold greater than 0.02.

It is clear that the adequate threshold value will vary depending on the specific application. This is a real difficulty of the approach. However, one of the advantages is that the induced error can be controlled by the user through the threshold value. This study shows that our lossy compression approach can be tuned to have little impact on the simulation results.

8.3 Application to Saint-Venant equations

In this section, we use a more realistic simulation to evaluate the impact of the compression pipeline on a computationally intensive simulation. It is important to note that the compression and decompression kernels presented in this study are not intended to reach the highest level of optimization. Rather, the purpose of evaluating the computational cost is to provide a reference point for the reader and to give an estimate of the potential impact of our method on a real simulation. As such, the results should be considered as an indicative measure of the performance and not as an absolute representation of the optimization level achievable with further fine-tuning.

8.3.1 Description of the scheme

To evaluate the computational cost of the compression pipeline on a more computationally intensive simulation, we use a Godunov scheme to solve a shallow water model. The shallow water model is defined by

$$W = \begin{pmatrix} h \\ hu \\ hv \end{pmatrix}, \quad Q(W,N) = \begin{pmatrix} h(uN^x + vN^y) \\ hu(uN^x + vN^y) + \frac{1}{2}gh^2N^x \\ hv(uN^x + vN^y) + \frac{1}{2}gh^2N^y \end{pmatrix}, \tag{8.8}$$

where h is the water level, (u, v) the horizontal velocity vector and $g = 9.81\text{m/s}^2$ the gravitational acceleration.

The velocity set is defined by

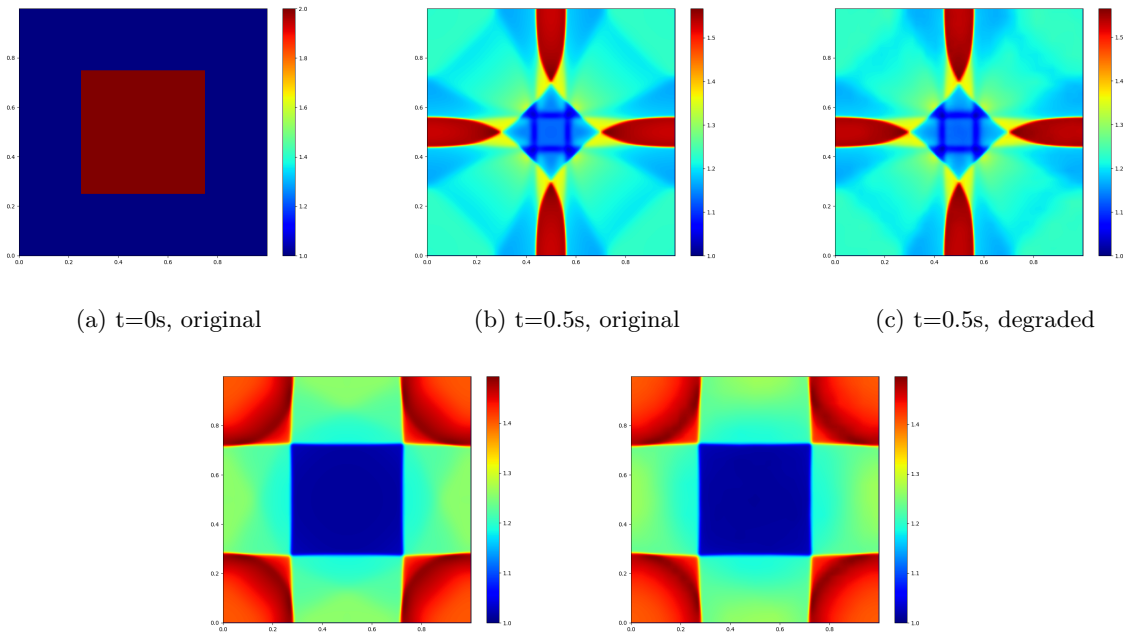
$$N^0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad N^1 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad N^2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad N^3 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}. \tag{8.9}$$

In the following simulations, we use the Godunov numerical flux, based on exact Riemann solvers (we refer, for instance, to [157] for the details). At $t = 0$, there is a 0.5-meter square where the water level is $h = 2$ meters high and the rest of the domain is at $h = 1$ meter (Figure 8.7a). We run the simulation on a 1025x1025 grid and perform 30390 time steps, which correspond to a simulation time of 10 seconds.

8.3.2 Computational cost

We use a compression level (number of DWTs) of 6 and a threshold value of 10^{-5}. We only show the results for the CSR lossless compression because the LZ4 compression provided poor compression ratios due to the method we used to implement fast wavelets (more details on this in the following). The lossless CSR compression is implemented using the `cuSPARSE` library.

Table 8.1 displays execution times of various kernels used for the Godunov simulation. Rows in the table indicate the total time spent in a kernel. The `total GPU time` row displays the total GPU computation time. The `time_step` row represents the execution time of a time step. The `wav_...` kernels correspond to the DWT kernels which include wavelet compression and decompression operations along the X and Y axes. The wav_y kernels are split into two parts and perform only one step of the DWT. The wav_x kernels perform the entire DWT along the X-axis in one kernel. The `dense_to_csr` and `csr_to_dense` kernels refer to the conversion between dense and compressed sparse row representations. The `cusparseParseDenseByRows_kernel` is an internal cuSPARSE kernel, and the remaining internal cuSPARSE kernels are grouped in the `other_cusparse_kernels` row. Finally, the `overhead` row shows the overhead percentage introduced by the compression method.



(a) t=0s, original (b) t=0.5s, original (c) t=0.5s, degraded

(d) t=1s, original (e) t=1s, degraded

Figure 8.7: These plots show the results of the simulation with no compression (original) and with the DWT (degraded, double precision). The threshold value is 10^{-5}.

Kernel	Execution time (s)					
	no compression		only wavelets		wavelets + CSR	
	single	double	single	double	single	double
total GPU time	19.6138	125.281	51.5696	168.010	64.5597	188.879
time_step	19.6138	125.281	19.6739	126.217	19.6581	126.439
wav_x_compress	∅	∅	4.08561	5.39602	4.09886	5.39384
wav_step_y_compress_samples	∅	∅	5.24588	7.50239	5.25510	7.50315
wav_step_y_compress_details	∅	∅	6.66140	8.09396	6.67206	8.09803
wav_x_decompress	∅	∅	3.47536	4.85359	3.48760	4.84650
wav_step_y_decompress_samples	∅	∅	7.23108	8.42857	7.37205	8.55370
wav_step_y_decompress_details	∅	∅	5.19632	7.51815	5.21071	7.55852
dense_to_csr	∅	∅	∅	∅	10.9194	11.8537
csr_to_dense	∅	∅	∅	∅	0.51037	0.52552
cusparseParseDenseByRows_kernel	∅	∅	∅	∅	0.12544	6.32183
other_cusparse_kernels	∅	∅	∅	∅	1.25000	1.785477
overhead	0%	0%	+162.12%	+33.11%	+228.41%	+49.38%

Table 8.1: Execution times of the different kernels for the Godunov simulation in single and double precision.

LZ4 compression was not tested on this simulation because it provides poor compression ratios. This is because the optimized DWT kernels used in this study are different from the ones used in the previous sections, where samples are grouped in a corner of the matrix and the details are stored in the remaining cells. This is convenient for the LZ4 compression because the details tend to be contiguous in memory, leading to a higher chance of finding matches in this area. In the optimized DWT kernels, samples are distributed evenly in the matrix and the details are stored in the remaining cells, which makes it harder for the LZ4 compression to find matches.

We measured 4.652 TFLOPS/s for the single precision simulation and 728.227 GFLOPS/s for the double precision simulation by accessing the performance counters. The peak performance of the V100 GPU according to the NVIDIA specification is 14 TFLOPS/s for single precision and 7 TFLOPS/s for double precision. The observed gap between our measured values and the peak performance specified by NVIDIA can be explained by the fact that some optimizations, such as the flush-to-zero, do not appear in the double precision version. This could be due to the compute capability of the V100 GPU not supporting these types of optimizations in double precision. We verified that the single precision version without the math optimizations has a ratio of approximately 2:1 compared to the double precision version.

The compression pipeline demonstrates impressive compression rates, with an average compression rate of **x31.8** for the single precision simulation and **x42.6** for the double precision simulation. The overhead introduced by the compression pipeline is **228.41%** for the single precision simulation and **49.38%** for the double precision simulation. The overhead introduced by the compression kernels is currently a bottleneck, particularly for the single precision simulation. However, the double precision simulation shows that the compression pipeline can be relevant, with a 50% overhead, which is acceptable given the extremely high compression ratio of more than x40. It is noteworthy that the overhead could be reduced significantly by investing efforts in optimizing the compression kernels. Our model suggests that the compression ratio can be even higher in a 3D case, with an expected ratio of around 160 for the single precision simulation and 250 for the double precision simulation.

The quality of the result can be evaluated by comparing the original uncompressed results to those obtained through DWT compression/decompression at each time step, as illustrated in Figure 8.7. While minor details may be lost in the compressed version, the general structure remains intact. The level of compression can be fine-tuned by adjusting the threshold value, but finding the optimal value can be challenging due to the artifacts introduced during detail thresholding and subsequent compression.

Despite the current overhead, our preliminary results are highly encouraging, as they showcase the potential of our compression pipeline for large-scale simulations. Future work will focus on further optimizing the compression kernels and exploring methods for decompressing only a limited number of subgrids at once, allowing for the surpassing of GPU memory limitations.

8.4 Discussions

This chapter explored using the discrete wavelet transform for high compression ratios in numerical simulations on regular grids. The algorithm allows a controlled loss of information while keeping the total mass consistent in the simulation. Experiments showed that for a 1025x1025 grid, two-dimensional compression could achieve a ratio of about 200x on a basic transport equation. For simulations using the shallow water equations, the compression ratio reached more than 40x, indicating the potential of the method for larger-scale simulations on GPUs. Expectations are that compression ratios will be higher in 3D simulations because they tend to increase with the number of dimensions.

The main challenge is the lossy nature of this compression. The study demonstrated that a minimal loss of information is needed for high compression ratios, controlled by an adjustable threshold. It is likely that most numerical schemes can tolerate a small threshold without significantly affecting the results, considering the inherent numerical errors in these schemes.

The study also found that the compression/decompression process slows down a double precision shallow-water simulation by less than 50%. This slowdown is the only additional cost of the method, which significantly reduces memory needs. However, to achieve real memory savings, only a few subgrids should be decompressed at a time, requiring a change in the current synchronization mechanism. The high compression ratios suggest that consequent memory savings are possible with this method.

This study focuses on situations where memory is limited. When the memory needed for a simulation exceeds the total memory of the system, the usefulness of a compression method lies in its balance between compression ratio and overhead. In cases where the required memory exceeds the available memory by a large margin, our method appears to achieve satisfactory trade-offs. Additionally, the overhead can be significantly lowered by optimizing the workflow, for example, by doing multiple time steps before compressing the subgrids back to global memory. This approach relates to temporal blocking, discussed in Section 3.4.

In summary, the method is an excellent option when the required memory far exceeds what is available. For situations where the memory gap is smaller, lighter methods may be preferred. Optimizing the compression kernels to lessen the overhead is possible, even if it means a lower compression ratio. This could involve fewer DWTs and making use of shared memory, which is faster but limited in size. The next chapter will look into a lighter compression method based on these principles, aiming for faster execution times with a focus on compression throughput.

Chapter 9

Designing a High-Performance Compression Scheme for CFD Simulations

This chapter is based on a reasearch article we have submitted to the *International Journal for Numerical Methods in Fluids*. In the preceding chapter, we have successfully designed a wavelet-based compression scheme for CFD data, focusing on the numerical aspects of the method. This previous work provided a practical implementation of the wavelet design we have proposed in Chapter 7 and demonstrated that it could be used to compress CFD data. The achieved compression ratios were very high, especially for a 2D simulation that is supposedly less compressible than a 3D simulation (the DWT is assumed to compress exponentially with the number of dimensions).

Achieving high compression ratios can always be an end in itself, if we consider that the required memory is by far the most important bottleneck of a simulation. However, it is more reasonable to consider the trade-off between the compression ratio and the compression overhead. This allows for a wide range of compression methodologies, ranging from rapid, but less compressive, to slower, but more compressive. We believe that the compression ratios achieved thanks to the DWT are high enough to make our approach competitive even against more rapid (and less compressive) compression methods. Hence, in this final work, we focus on providing a high-throughput compression scheme, based on the principles we have seen so far. The goal is to demonstrate that it is possible to base a high-performance compression scheme on the DWT, which goes against the common belief that the DWT is too slow for real-time applications (see Section 2.2.3).

9.1 Introduction

We recall that we consider fluid data to be regular grids of scalar values, which are updated at each time step. As we have seen several times, a downside of using regular grids is that the required memory increases tremendously with the grid size. Different methods exist to address this problem, as we have seen in Chapter 2. We choose to use explicit compression methods for reasons we detailed in the latter chapter.

The adoption of compression in numerical simulations is generally driven by two main needs: overcoming the memory capacity constraints of the hardware and accelerating simulations by utilizing computational resources more efficiently. Figure 9.1, illustrates the various levels of memories within the context of GPU programming and their typical bandwidths. Visualizing the memory setup in this manner helps to understand the different possible bottleneck scenarios. If the bottleneck of an application is the shared memory accesses (7514 GB/s on the figure), then little can be done to improve the performance, appart from using another less memory-bound algorithm. If the bottleneck is the global memory accesses (732 GB/s on the figure), then it is conceivable to expect gains by making better use of the shared memory. Finally, if the bottleneck is the CPU-GPU memory transfers (26 GB/s on the figure), then it is possible to expect gains by reducing the amount of data transferred between the CPU and the GPU. This latter case is the focus of our work. We can see that the CPU-GPU throughput is extremely low compared to the inner GPU bandwidths,

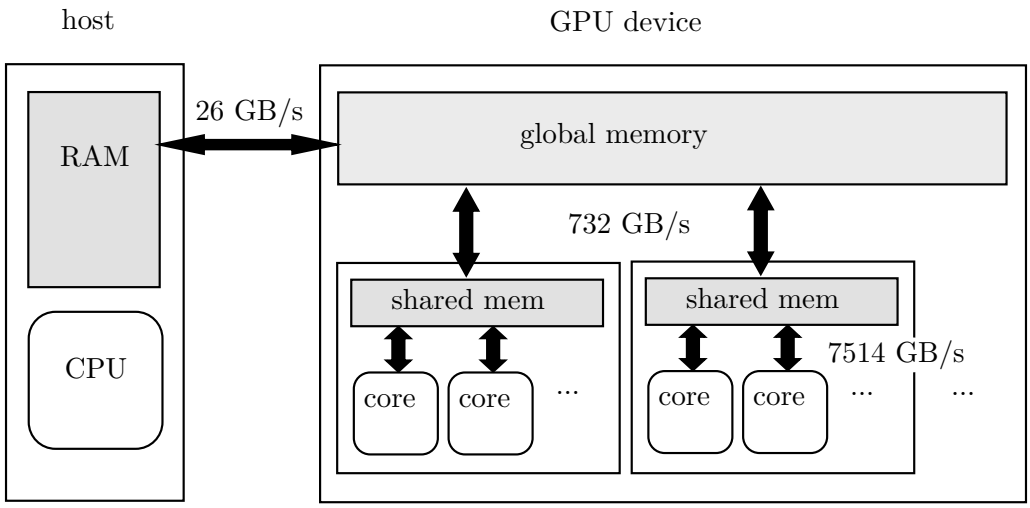


Figure 9.1: Schematic representation of the memory setup of a GPU. The provided throughputs have been measured with *gpumembench* [136] and *nvbandwidth* [198] on a P100 GPU. They are only indicative and can vary depending on the used hardware/software. The bandwidth of the memory transfers between the host and the GPU device is significantly lower that what can be achieved within the GPU. The main GPU memory (DRAM) is slower than the shared memory (SRAM), but has a much larger capacity and can be accessed by all the GPU cores. The shared memory is a block-level memory and can only be accessed by the cores of the same block.

bringing the performance of the GPU to a halt if the data does not fit in the GPU memory. Hence, introducing a compression methodology that allows for the simulation of larger grids on the GPU without too much overhead will lead to an undeniable performance improvement. As such, we are satisfied if the compression/decompression throughput is higher than the CPU-GPU throughput.

The work presented in the previous chapter focused on demonstrating the feasibility of using a lossy wavelets compression to accurately simulate fluid dynamics. It has showed that the approach is relevant for CFD simulations, but did not provide a satisfactory *on-the-fly* implementation of the compression scheme. The idea of this new work is to provide a high-throughput implementation of the wavelet-based compression scheme, which can be used in real-time simulations. The goal is to demonstrate that the DWT can be used in a high-performance context.

In this study, we explain our compression method, discuss why certain design choices were made, and assess its performance. Our tests demonstrate that our method allows for the simulation of grids that would not fit in the GPU memory without compression. We also provide insights into the impact of the compression on the overall simulation time. Depending on the configuration, we can expect the overall scheme to be between 2 and 3 times slower (although this measure can be disputed in cases where the scheme could not be run without compression). Given the significant reduction in memory requirements, this trade-off is acceptable in many scenarios, especially in cases where CPU-GPU transfers of the whole data are required to fit the simulation in the GPU memory. Overall, our method is a promising solution to improve the memory efficiency of large-scale CFD simulations.

In Section 9.2, we begin by describing the compression scheme, focusing on obtaining high-performance on GPUs. We then propose a methodology for reaching effective memory savings in CFD simulations. Building on this, we present the results of our experiments in Section 9.3, where we evaluate the performance of the compression scheme in a large-scale D3Q27 LBM simulation. Finally, we conclude with a discussion of the results and potential future work in Section 9.4.

9.2 Description of the Compression Scheme

This section provides details on our compression method. We first describe the compression scheme, focusing on the wavelet-based approach in Section 9.2.1. Then we explain how the method is integrated into a CFD simulation in Section 9.2.2.

9.2.1 Compression Scheme

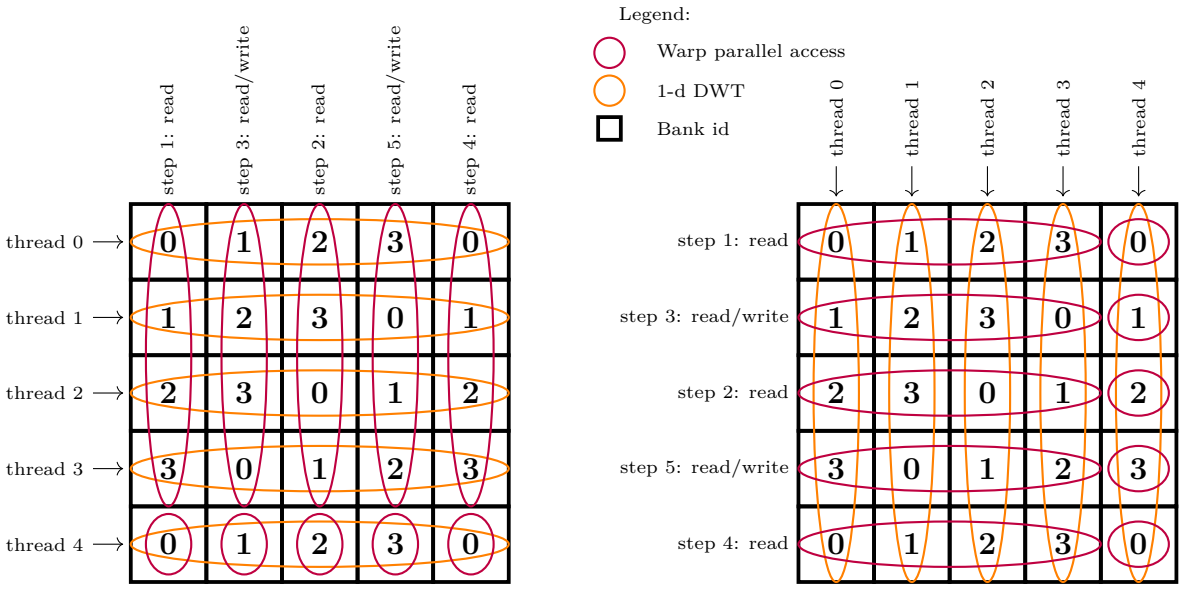
In our novel approach, we propose to perform smaller local DWTs on the data and leverage the shared memory of the GPU to store the data. As the shared memory is significantly faster than the global memory, we expect this approach to yield substantial performance improvements. The shared memory differs from global memory in different aspects. It is block-level memory with limited capacity (usually in the order of tens of kilobytes per block) and is accessible only by cores within the same block. Shared memory is also divided into several regions (banks) that can be accessed simultaneously by threads of a same warp (set of 32 threads). While exact coalesced access patterns are unnecessary, shared memory is prone to bank conflicts when multiple threads access the same bank simultaneously, leading to potential slowdowns due to serialized access. To leverage shared memory effectively, our parallelization strategy is designed to circumvent bank conflicts and optimize memory throughput.

Our algorithm divides the global grid into smaller blocks, each sized to fit within the capacity limitations of shared memory. Each CUDA block transfers data from global to shared memory, executes the DWT in-place within shared memory, and writes the compressed results back to global memory. The chosen block size is $33 \times 17 \times 17$, consuming approximately 37.25 KB of shared memory for single-precision data, which is within the capacity of most modern GPUs. The block sizes are intentionally set to powers of 2 plus 1 to accommodate the DWT scheme utilized. The 1-d DWT is applied consecutively along each axis of the 3-d block within the shared memory, with each thread processing a different line of the block. Threads employ the lifting scheme on their respective lines and synchronize between axes using the `__syncthreads()` function, hence requiring at least two thread synchronizations overall.

Figure 9.2 provides a schematic of the shared memory layout in a 2-d slice of the 3-d block fetched from global memory, demonstrating the execution of memory accesses for the DWT along the x and y axes. The "step x" labels indicate the sequence of memory accesses in the lifting scheme implementation. Assuming a bank number and a warp size of 4 for illustration, the principle remains applicable for any power of 2. Given the block sizes are powers of 2 plus 1, the bank numbers are the same across each axis, with bank numbers incrementing by 1 (modulo the total bank number) when transitioning to adjacent cells in the same row or column. This layout ensures an even distribution of memory accesses across banks, crucial for minimizing bank conflicts. Moreover, the indivual warp accesses (depicted by the red oval shapes) always access to different banks, which guarantees no bank conflicts. The same principles apply to the z axis, which is not shown for brevity.

Finally, the result of the DWT can be compressed using any lossless compression method. We choose to use a COO (Coordinate) format to achieve lossless compression. This choice offers both high compression ratios thanks to the sparsity of the data and relatively fast compression/decompression times. However, the GPU code for performing the *dense-to-COO* (compression) and *COO-to-dense* (decompression) operations is not trivial, as it involves irregular memory accesses and inter-thread communication. The *dense-to-COO* is close in spirit to a parallel *reduction* or *scan* [41], while the *COO-to-dense* is close to a parallel scatter operation. Our implementation of *dense-to-COO* is close to the idea for GPU parallel reduction provided by Harris [109], with the notable change that the warp-level reduction primitives are now directly available in the CUDA programming model. The idea is also similar to Code 3.5, that we used as an example in Chapter 3. The warps start by scanning the non-near-zero coefficients across the block and counting them by performing a warp-level reduction. Then, a block-level reduction is performed to compute the offset of each warp for the final write to the COO format. The *COO-to-dense* operation is more direct, as it is a simple scatter operation. The thresholding is performed during the *dense-to-COO* by integrating only the coefficients above the threshold into the COO format.

This new compression scheme, which works with local wavelets, is expected to be faster than the first version with global wavelets, as it minimizes the number of global memory accesses. The compression kernel performs a single read from global memory on the decompressed data, followed by a single write on the



(a) Wavelets on the x axis.

(b) Wavelets on the y axis.

Figure 9.2: Schematic representation of the shared memory layout in a 2-d slice of the 3-d block loaded from the global memory. For the sake of visibility, a bank number and a warp size of 4 are assumed. Part 9.2a shows how the wavelets on the x axis are performed, while part 9.2b shows how the wavelets on the y axis are performed. The parts where the 1-d DWT is performed are hightlighted with orange oval shapes. The individual warp accesses are represented by the red oval shapes. Each cell in the shared memory is represented by a square and its corresponding bank is written in the cell.

compressed data (and conversely for the decompression kernel). Data reuse is maximized by performing the required memory accesses on the shared memory, which moves the bottleneck from DRAM (global) accesses to faster SRAM (shared) accesses. The downside of this approach is that the compression is performed on local blocks, which hurts the compression ratio. However, we will see in Section 9.3 that the compression ratio is still acceptable in practice.

9.2.2 Methodology for CFD Data Compression

Our approach strategically partitions the computational grid into smaller, manageable subgrids, which lets us process the data in a more flexible manner. This partitioning is an important aspect of our compression scheme, as it is required to achieve actual memory savings. As depicted in Figure 9.3, the entire grid is divided into subgrids, which are further segmented into blocks. These blocks are only used in the DWT step, where they are loaded in the shared memory of a CUDA block to perform the DWT locally.

When decompressed, the subgrids use a classical row-major storage format, with each row stored contiguously in memory. This implies that the blocks are not contiguous in the global memory, as they are separated by the offsets between rows. Each partitioning serves a different purpose: subgrids allow for partial decompression of the grid, while blocks facilitate the compression/decompression (DWT and COO) operations.

LBM computations are performed directly on the decompressed subgrids, which are stored in the global memory. These necessitate the values of the neighboring subgrids to be available. To achieve this, we use a classical ghost cell approach, where the ghost cells duplicate the values of the neighboring subgrids. To account for the fact that the neighboring subgrids are not necessarily directly available (due to being compressed), we use interface buffers. These interface buffers let us have an uncompressed version of the relevant edge values of all the grid. This lets us divide the subgrid synchronization into two phases: reading from the interface buffers to update the ghost cells and writing to the interface buffers the results of the LBM computations.

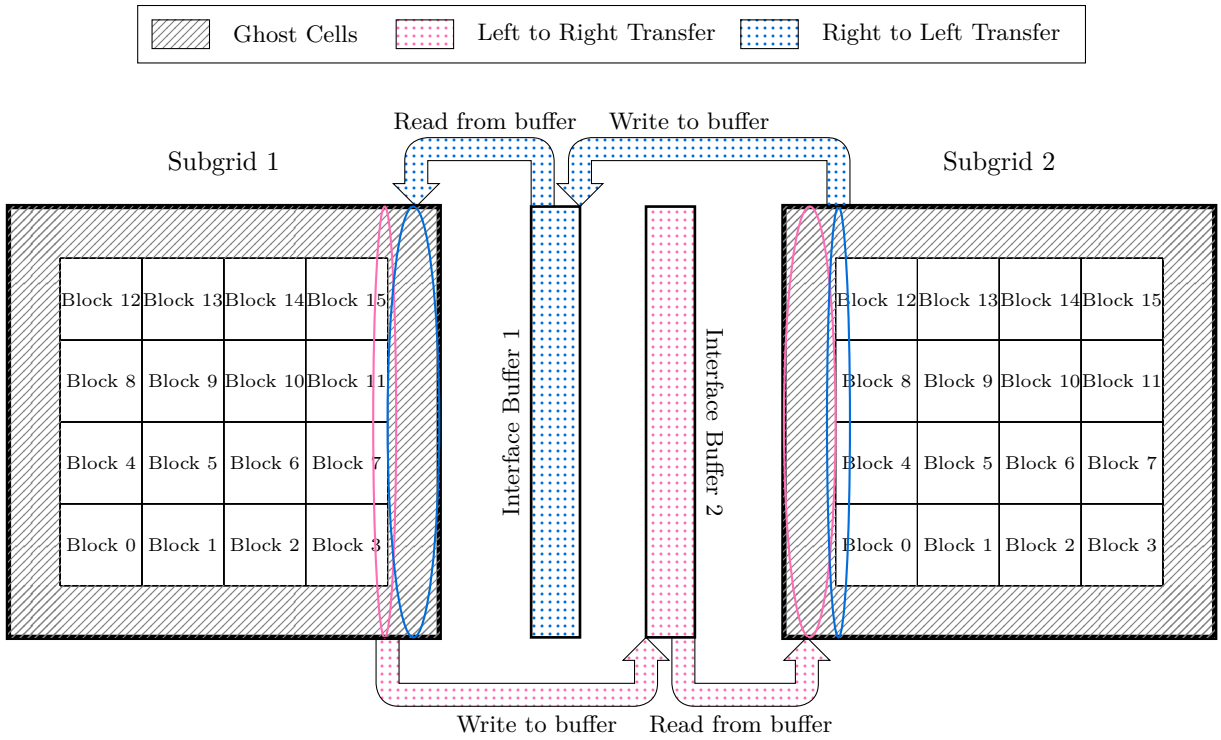


Figure 9.3: Illustration of the hierarchical grid subdivision in 2D. The grid is subdivided into subgrids, which are further segmented into blocks. Each block is processed by a single CUDA block for the DWT and is sized to fit within the shared memory of the GPU blocks. Subgrids represent contiguous memory segments in the global memory (when decompressed), while blocks include offsets between rows. Synchronization between subgrids is facilitated through interface buffers along each dimension/direction. Ghost cells are updated by reading from these buffers and values are written back to the buffers after each LBM iteration. Blocks do not need synchronization, as the LBM computations are performed directly on the global decompressed subgrid. This hierarchical model is extendable to multiple dimensions.

To achieve effective memory gains, we only reserve a fixed amount of memory for two subgrids and all the interface buffers. The idea is to process each subgrid consecutively, hence only allowing to have at most two uncompressed subgrids in the global memory at a time. The same idea would work with a single subgrid if we assumed in-place computation of the LBM step, but we do not make this assumption for the sake of generality. The interface buffers are used in a duplicated, alternating fashion to ensure coherent data accesses between different time steps. The rest of the memory is used for the compressed version of the subgrids and is stored in a circular buffer. This buffer can be viewed as an infinite succession of compressed subgrids $s_{0,\text{it}}, s_{1,\text{it}}, \ldots, s_{N-1,\text{it}}, s_{0,\text{it}+1}, s_{1,\text{it}+1}, \ldots$, where $s_{i,\text{it}}$ is the compressed subgrid i at iteration it and N is the number of subgrids.

Figure 9.4 illustrates how these segments are used in the execution of a LBM step on subgrid 0 at iteration *it*. Let us first notice that the circular buffer contains a window in which the compressed subgrids that are still needed for the simulation are stored. The goal of this whole process is to advance this window by one subgrid. The process begins with reading the compressed subgrid from the circular buffer and decompressing it into a buffer. Prior to the execution of the LBM step, the ghost cells are updated using data from the corresponding interface buffers. Post LBM step, the results are stored in the other interface buffer and the processed subgrid is re-compressed and written to the current cursor of the circular buffer. This process is then repeated for the next subgrid (or the next time step if there are no more subgrids), with the compressed subgrids window slided to the right. This cyclical process of reading, updating, processing, and writing back to the circular buffer ensures that the required memory remains below the GPU capacity (assuming a given compression ratio). Overall, this process flow allows to reach effective memory savings, as only partial

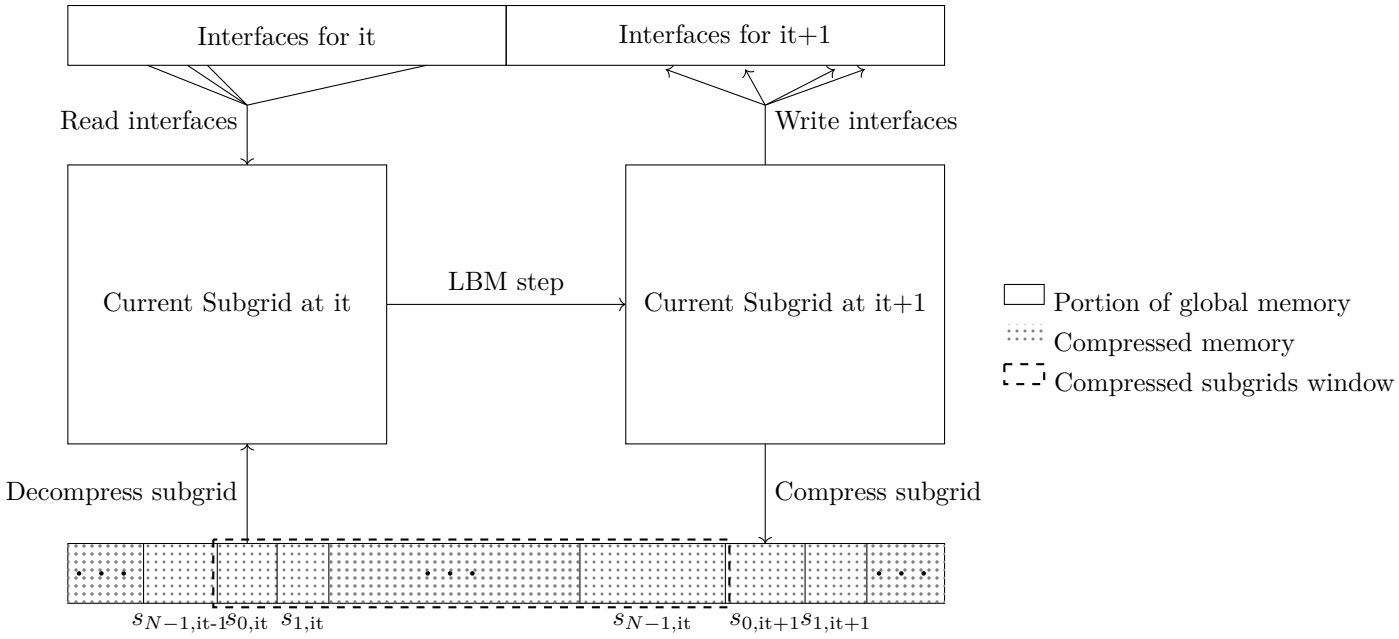


Figure 9.4: The figure illustrates the workflow for executing a Lattice-Boltzmann step on subgrid 0 at iteration *it*. All the data shown in the figure are stored at all time in the GPU global memory (DRAM). Initially, the subgrid is retrieved from the circular buffer in its compressed form and decompressed into a subgrid buffer. Prior to the LBM step, ghost cells are updated (indicated by "Read interfaces" arrows). Post-LBM step, the processed data are stored in a new buffer. Interface data for subsequent iterations are stored ("Write interfaces" arrows), and the resultant subgrid is re-compressed and written to the circular buffer. The GPU memory is strategically partitioned into three segments: the circular buffer for compressed subgrids, subgrid buffers for decompressed data, and interface buffers. These subgrid and interface segments are of fixed sizes, ensuring efficient reuse in each iteration. The design incorporates two sets of interface buffers that alternate between iterations to represent current and subsequent iteration data. It is assumed that $s_{0,\text{it}}, s_{1,\text{it}}, \ldots, s_{N-1,\text{it}}$ does not overlap with $s_{0,\text{it+1}}, s_{1,\text{it+1}}, \ldots, s_{N-1,\text{it+1}}$ (i.e., the compressed size fits within the circular buffer).

decompression of the data is performed at a time.

The integration of our compression methodology with an LBM simulation lets us bypass the need for CPU-GPU data transfers, as long as the compressed data and the decompressed data (the subgrid and interface buffers) fit in the GPU memory. By addressing these memory constraints, our approach enables the execution of larger-scale simulations for a given hardware. In the next section, we present results to demonstrate the effectiveness of this approach.

9.3 Results

9.3.1 Experimental Setup

To evaluate how our compression scheme influences the execution of an LBM simulation, we performed a set of experiments. Our selection for this assessment is a D3Q27 LBM flow simulation featuring a sphere as an obstacle. This scenario is a well-established challenge in fluid simulations, providing insights into various flow regimes. Our D3Q27 LBM is a variant of the initial scheme of d'Humières [87].

Additionally, it is known to present local complex fluctuation mixed with large quiet regions, especially in the presence of unsteady flows. These features are well adapted to wavelet compression. Additionally, our investigation confirmed that this approach exhibits a memory-bound characteristic on GPUs, achieving

a memory bandwidth of approximately 80% of the theoretical peak across all tested GPUs (excluding the bounce-back condition). This observation underscores rapid execution of the scheme on GPUs, a feature that poses challenges for employing compression, as it accentuates the overhead associated with compression techniques.

The conservative variables of this LBM scheme are the density and the density-weighted velocity: $W = (\rho, \rho u_x, \rho u_y, \rho u_z)$. We have thus 4 physical data by grid point. The physical data are represented at each grid point by a larger set of so called kinetic data f_i for $0 \leq i < 27$, hence the "D3Q27" terminology. The mapping between f_i and W_j is performed as described below. We refer to the (excellent) book of Krüger *et al.* [139] for an introduction to the LBM.

The equilibrium distribution function $f_{eq,i}$ provides the lattice velocity i at equilibrium, and is computed using the following equation:

$$f_{eq,i}(W) = C_i \rho \left(1 + \frac{3}{c^2}\vec{e}_i \cdot \vec{u} + \frac{9}{2c^4}(\vec{e}_i \cdot \vec{u})^2 - \frac{3}{2c^2}\vec{u} \cdot \vec{u}\right), \tag{9.1}$$

where C_i is the weight of the velocity $\vec{e}_i$ and c is the speed of sound. The implementation is based on the commonly used 27-velocity set: $(0,0,0)$, $(\pm 1,0,0)$, $(0,\pm 1,0)$, $(0,0,\pm 1)$, $(\pm 1,\pm 1,0)$, $(\pm 1,0,\pm 1)$, $(0,\pm 1,\pm 1)$, and $(\pm 1,\pm 1,\pm 1)$, with corresponding weights: $\frac{8}{27}$ (for the center), $\frac{2}{27}$ (for the 6 "faces"), $\frac{1}{54}$ (for the 12 "edges"), and $\frac{1}{216}$ (for the 8 "corners") [309]. This allows us to compute W from the distribution function f and vice versa using the following equations:

$$\begin{aligned} \rho &= \sum_{i=0}^{26} f_i &&= \sum_{i=0}^{26} f_{eq,i}, \\ \rho\vec{u} &= \sum_{i=0}^{26} c\vec{e}_i f_i &&= \sum_{i=0}^{26} c\vec{e}_i f_{eq,i}. \end{aligned} \tag{9.2}$$

Each step of the LBM algorithm starts by shifting the 27 kinetic data f_i in the directions of the corresponding 27 lattice velocities $\vec{e}_i$. This step induces memory transfer between the grid points. In the second step, the kinetic data are updated according to:

$$f = \omega f_{eq} + (1-\omega) f, \tag{9.3}$$

where ω is the relaxation parameter. This step is done locally at each grid point and is completely parallelizable on the GPU.

Thus, a time step consists of the following operations:

1. read the kinetic data f_i from the neighbors (shift phase). If the neighbor is in an obstacle, we consider reading $f_{i'}$ instead, where $e_{i'}$ is the opposite velocity to e_i in the lattice (bounce-back condition). This boundary correction can break the coalescent memory access, as we shall see later;
2. compute the conservative data W from equation (9.2);
3. compute the equilibrium distribution function f_{eq} from W with equation (9.1);
4. write the new distribution function f according to (9.3)

We divide the domain into $4 \times 16 \times 4$ subgrids, each of which being a grid of cells of size $\Delta x \times \Delta y \times \Delta z$, with $\Delta x = \Delta y = \Delta z$. The initial condition is $\rho = 1$ everywhere, $\vec{u} = (0.0001, 0.03, -0.0001)$ outside the obstacle, and $\vec{u} = (0,0,0)$ inside the obstacle. The time step Δt is deduced from Δx and the speed of sound c (set to a dimensionless value of 1): $\Delta t = \text{CFL}\frac{\Delta x}{c} = \Delta x$ in our case. The CFL number is set to one as is always the case in the LBM. We set the relaxation parameter ω depending on Δt so that the corresponding Reynolds number is 300. The ω parameter is computed from the following equations:

$$\nu = c^2 \left(\frac{1}{\omega} - \frac{1}{2}\right) \Delta t, \tag{9.4}$$

$$\Leftrightarrow \omega = \frac{2}{1 + 2\frac{\nu}{c^2 \Delta t}}, \tag{9.5}$$

while the viscosity ν is set such that the Reynolds number is 300:

$$\mathrm{Re} = \frac{cL}{\nu} = 300, \tag{9.6}$$

where L is the characteristic length of the obstacle; in our case, the diameter of the sphere. We use periodic boundary conditions in all directions except when we show the vortices passing the obstacle. In this case, we use a fixed boundary condition for the low y values, so that the incoming flow is constant.

We run tests with different NVIDIA GPUs: P100 (16GB), V100 (16GB), and A100 (40GB). The code is written in CUDA and runs mostly on the GPU. We use custom CUDA events to measure the time spent and verify that our measures are coherent with the output of `nvprof`. Our implementation allows for different scenarios, such as with or without compression, with or without subgrids, different threshold values, and different obstacles. The results can be saved thanks to a custom compressed file format and visualized thanks to a custom python script based on the mayavi library [215].

9.3.2 Setting the Threshold

The objective of this experiment is to assess the impact of the threshold value on simulation execution. Flow simulations were conducted at various threshold levels, with results recorded at $t_{\max} = 1.0s$. The grid size for these simulations was $231 \times 952 \times 238$, with $\Delta x \approx 0.01732$.

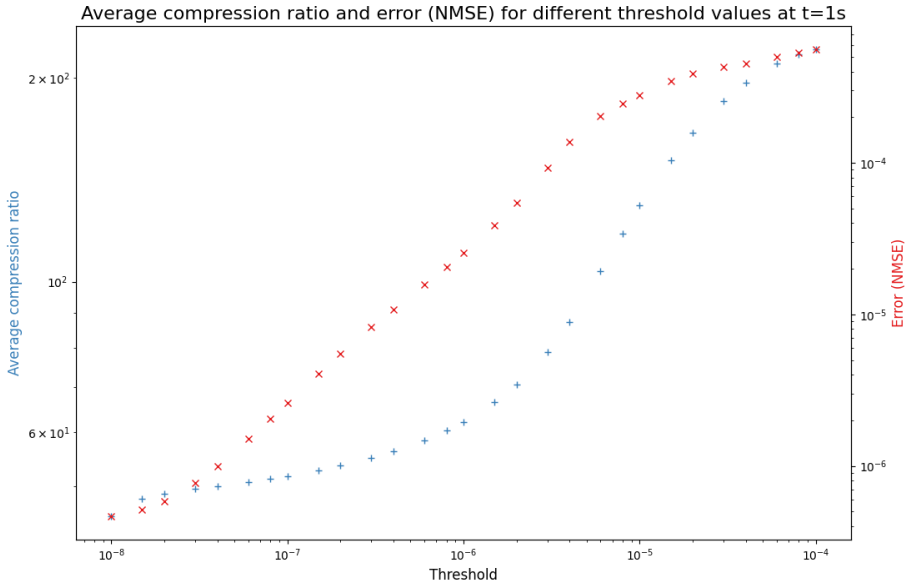


Figure 9.5: Impact of the threshold on compression ratio and error in a D3Q27 LBM simulation. The figure shows average compression ratios and errors across the domain for different threshold settings. Blue crosses indicate the compression ratio, and red crosses denote error, measured as the Normalized Mean Squared Error (NMSE) between reference density and density after lossy compression.

Figure 9.5 displays the results of the experiment, underscoring two main observations. The graph shows the average compression ratio, depicted by blue crosses, which compares the size of compressed data to uncompressed data at a specific timestep. It also presents the Normalized Mean Squared Error (NMSE) between the reference density and the density after lossy compression, defined as:

$$\mathrm{NMSE} = \frac{\sqrt{\sum_{i,j,k} \Delta x \Delta y \Delta z \left(f_{i,j,k} - \hat{f}_{i,j,k}\right)^2}}{\sqrt{\sum_{i,j,k} \Delta x \Delta y \Delta z \left(f_{i,j,k}\right)^2}}, \tag{9.7}$$

where $f_{i,j,k}$ is the reference density and $\hat{f}_{i,j,k}$ is the density after lossy compression.

This visual analysis permits a detailed examination of the effects of threshold variation on the simulation. Distinct behavioral regimes are identifiable:

- In the range of $[10^{-8}, 2 \cdot 10^{-8}]$, increasing the threshold significantly improves the compression ratio without altering the error, suggesting coefficients removed in this range likely correspond to noise.
- Between $]2 \cdot 10^{-8}, 10^{-6}]$, there is a nearly linear increase in error, with a less pronounced rise in the compression ratio. This indicates the beginning of an impact on the simulation by the removal of the coefficients, yet without major disruption.
- Beyond $]10^{-6}, 2 \cdot 10^{-4}]$, a sharp increase in compression ratio is observed alongside error stabilization, implying that artifacts at this stage severely compromise simulation integrity, rendering the simulation impractical.

The first two regimes are considered potentially beneficial for meaningful simulations. The initial regime offers an optimal scenario, enhancing compression without affecting accuracy and eliminating superfluous noise. The subsequent regime, though riskier, allows for increased compression at the risk of introducing disruptive artifacts, necessitating thorough result analysis. The final regime, marked by excessive error, is deemed unsuitable for productive simulation efforts.

Notably, threshold determination is influenced by specific problem parameters and poses a challenge due to interactions between numerical and compression schemes. However, chosing a threshold value in the first regime is likely to yield acceptable results, as it offers a balanced compromise between compression efficiency and error minimization without evident artifacts. Hence, the threshold value is set to $2 \cdot 10^{-8}$ for the subsequent experiments, as it falls within the optimal range.

For simulations using different grid sizes than $231 \times 952 \times 238$, the threshold value adjusts to ensure analyzed frequencies align with identical physical scales:

$$\tau = \tau_0 \frac{\Delta x}{\Delta x_0}, \tag{9.8}$$

where Δx_0 is the baseline spatial step (≈ 0.01732) and τ_0 is the predetermined threshold value ($2 \cdot 10^{-8}$).

9.3.3 Validation of the Scheme

We perform various tests to validate the scheme. We first verify that the scheme preserves the mass up to machine precision. This property holds true as long as the domain is periodic and no fixed boundary condition is used. It works, among other:

- with the direct implementation of the LBM scheme on the GPU (without compression);
- with the implementation with subgrids but no compression;
- with the implementation with subgrids and compression.

We, hence, have strong evidence that both the subgrid synchronization mechanism and the compression scheme are correctly implemented.

To further validate the scheme, we show the visual results of the simulation for various configurations. We set ω so that the corresponding Reynolds number is 300. Figures 9.6 and 9.7 show the results of the simulation at $t_{\max} = 600s$ with a $165 \times 680 \times 170$ grid. The first figure is the result of the simulation without compression, while the second figure is the result of the simulation with compression. In both cases, we observe the highly periodic flow that we would expect for a Reynolds number of 300 [126, 269]. Figure 9.8 shows the result of the simulation at $t_{\max} = 1000s$ with a $594 \times 2448 \times 612$ grid. This simulation is costly in terms of computations and took approximately 3 days to run on an A100. Only the subspace $x = [-1.25, 0.75]$, $y = [1.9, 5]$, $z = [-0.55, 1.45]$ is saved, corresponding to approximately 9GB of floating point data in our case, which is close to the maximum size that can be visualized on a regular laptop. The saved space captures two vortex rings, which consist of a similar flow pattern to the smaller simulation. These visual results indicate that the compression scheme does not introduce significant errors and demonstrate the

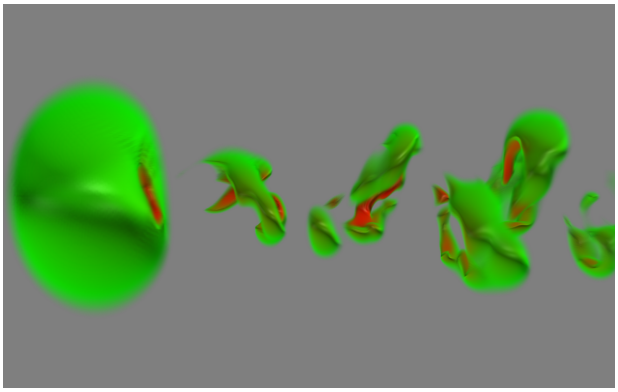


Figure 9.6: D3Q27 LBM simulation at $t_{\max} = 600s$ (24777 iterations) with a $165 \times 680 \times 170$ grid ($\approx$ 1.92GB) and no compression.

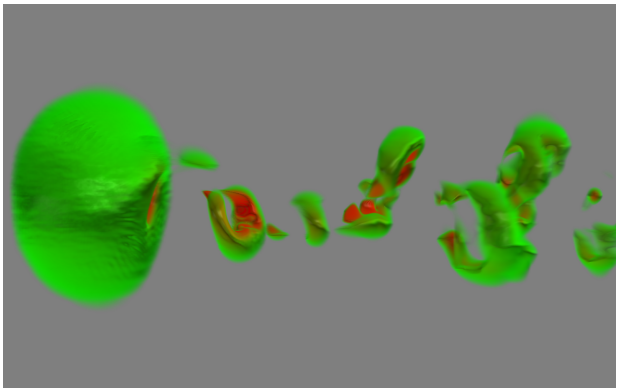


Figure 9.7: D3Q27 LBM simulation at $t_{\max} = 600s$ (24777 iterations) with a $165 \times 680 \times 170$ grid ($\approx$ 1.92GB) and lossy compression.

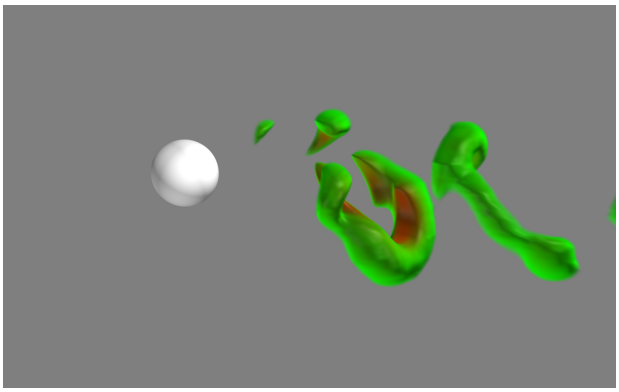
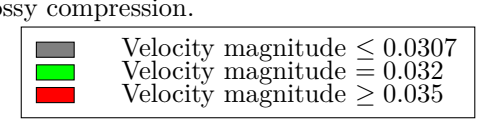


Figure 9.8: D3Q27 LBM simulation at $t_{\max} = 1000s$ (148685 iterations) with a $594 \times 2448 \times 612$ grid ($\approx$ 89.51GB).



potential to simulate higher-precision simulations on GPUs. Let us note that this D3Q27 scheme is a worst case scenario for the compression scheme, as the turbulences are known to originate from slight variations in the flow, which can be disrupted by the compression scheme. To obtain these results, the threshold must be particularly low, which severely impacts the compression ratio, as we will see in the next section.

9.3.4 Performance Evaluation

This section presents the results of our performance evaluation for the D3Q27 simulation, considering five configurations: direct implementation with two fully decompressed grids (with and without bounceback condition), implementation with subgrids and no compression, and implementation with subgrids and compression (block-level wavelets or global wavelets). The block-level compression is the novel compression scheme where the wavelets are performed locally in the shared memory, while the global compression is the previous compression scheme where the wavelets are performed on the whole subgrids. In the following experiments, the block-level performs 3 DWTs on the x axis and 2 DWTs on the y and z axes (to achieve $j = 2$ on all axes for a block size of $33 \times 17 \times 17$), while the global compression performs 2 DWTs on all axes.

For each configuration, we run the simulation with different grid sizes with $t_{\max} = 1.0s$, except for the compression with global wavelets, where $t_{\max}$ is lowered depending on the grid size because of how slow the execution is. We verified that the number of performed iterations does not significantly impact processing speeds.

Table 9.1 shows the average percentage of time spent in the different kernels for the different configurations. It provides insights into the performance of the different configurations. We can see that the global

		Proportion of time spent in the kernels (in percentage)				Compression ratio (first time step)
GPU	Configuration	Numerical scheme	Compression	Decompression	Synchronization	Ratio
A100	No subgrids, no compression	100.00%	0.00%	0.00%	0.00%	1.00
A100	Subgrids, no compression	91.12%	0.00%	0.00%	8.88%	1.00
A100	Subgrids, block compression	51.04%	28.59%	17.78%	2.59%	206.57
A100	Subgrids, global compression	3.18%	81.20%	15.49%	0.13%	1206.12
V100	No subgrids, no compression	100.00%	0.00%	0.00%	0.00%	1.00
V100	Subgrids, no compression	87.96%	0.00%	0.00%	12.04%	1.00
V100	Subgrids, block compression	46.57%	25.92%	22.65%	4.86%	200.93
V100	Subgrids, global compression	4.83%	73.12%	21.68%	0.36%	539.41
P100	No subgrids, no compression	100.00%	0.00%	0.00%	0.00%	1.00
P100	Subgrids, no compression	87.70%	0.00%	0.00%	12.30%	1.00
P100	Subgrids, block compression	36.25%	37.75%	22.27%	3.73%	200.93
P100	Subgrids, global compression	6.18%	68.60%	24.73%	0.49%	539.41

Table 9.1: Average percentage of time spent in the different kernels on the different configurations.

compression, where the wavelets are performed on the whole subgrids, is significantly slower than the block-level compression, where the wavelets are performed at the block level. With global compression, between 3% and 7% of the time is spent on average in the LBM computations. For the block-level wavelets, both the A100 and the V100 spend approximately 50% of the time in the LBM computations, while the P100 spends approximately 35% of the time. We hence, see that the global wavelets are drastically slower than the block-level wavelets. The table also shows the compression ratio achieved on the first time step for the different compression kernels. This metric highlights the fact that the global compression yields better compression ratios than the block-level compression, with one less DWT level on the x axis. Both algorithms, hence, provide a different trade-off between compression ratio and execution time. If the required compression ratio becomes the bottleneck, the global compression kernels can be used, and the amount of performed DWTs can be adjusted to reach the desired compression ratio.

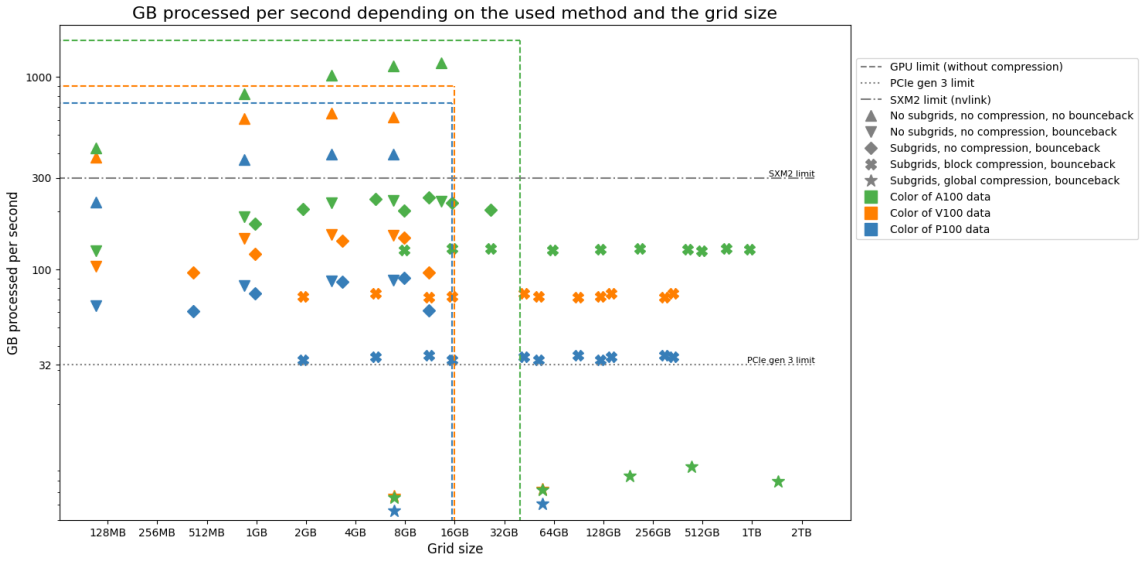


Figure 9.9: Performance evaluation of the D3Q27 LBM simulation with different configurations. The x-axis shows the total grid size (in GB) if decompressed, while the y-axis displays the processing speed (in GB/s). Hardware (P100, V100, or A100) is represented by color, while the marker denotes the method used. The markers represent the result for a single execution. The dashed colored lines represent the theoretical bounds of each tested GPU based on their specifications, assuming no compression.

To normalize the performance between different configurations, we propose to compare the processing speed of the different configurations in Figure 12.5. The x-axis represents the total grid size if decompressed,

while the y-axis indicates processing speed if decompressed. The processing speed (S) is calculated using Equation 9.9, considering grid size, number of iterations, and total time, with a factor of 2 for read-write cycles per iteration.

$$\mathrm{S} = 2 \times \frac{\mathrm{grid\_size} \times \mathrm{num\_iterations}}{\mathrm{total\_time}} \ \mathrm{GB/s} \tag{9.9}$$

This figure helps to evaluate the impact of the method on simulation performance. Firstly, the executions with no bounceback condition (dashes) achieve high processing speeds, typically about 80% of the theoretical peak of the GPU. On the other hand, the same executions with the bounceback condition (dots) are significantly slower, between 2 and 5 times slower, depending on the grid size and the GPU. This is explained by the fact that the bounceback condition breaks the coalesced memory access, which is a well-known issue when implementing the LBM on the GPU [199]. The executions with subgrids, but no compression (triangles), use the same workflow we use for compression (see Section 9.2.2). We can see that this method is able to run larger simulations. This is due to the used workflow, which does not require to store two fully decompressed grids at the same time. We can also see that the processing speed is not systematically lower than the version with no subgrids. This can be explained by the better granularity of the version with the subgrids, where the subgrids that have no bounceback condition (due to not including the obstacle) operate at the near-perfect speeds that we oberve for the version with no bounceback condition (dashes). This gain can overcome the overhead of the subgrid synchronization.

Finally, the executions with subgrids and compression (crosses) show the impact of the compression on performance. We can expect the simulations that integrate compression to be approximately 2 to 3 times slower than the best case scenario without compression. This slowdown allows the simulation of grids of size up to 13 times the capacity of the 40GB A100 and 8 times the capacity of the 16GB P100/V100 (up to $t_{\max} = 1.0s$). It is important to note that unless in-place computation is used, the grid size would normally be at most half the capacity of the GPU, as the grid needs to be stored twice (once for the input and once for the output). The implication of this observation is that for a given hardware, the effective grid size that can be simulated is significantly increased by the use of compression.

The figure also shows the maximum bandwidth of PCIe gen 3 and SXM2 (NVLink), which are associated with the V100 GPU. We can see that all the processing speeds are greater than the maximum PCIe (gen 3) bandwidth of 32GB/s. The maximum PCIe gen 4 bandwidth (64GB/s) is also surpassed by the A100, which is the only GPU that supports PCIe gen 4. This observation highlights the potential of the compression scheme to reduce the requirements for PCIe data transfers, which is generally a bottleneck in multi-GPU CFD simulations. No processing speed surpasses the maximum corresponding NVLink bandwidth (160GB/s for P100, 300GB/s for V100, and 600GB/s for A100), which are particularly fast.

Figure 9.10 illustrates how the compression ratio changes over time across different simulation setups. The compression ratio, calculated as the ratio between compressed and uncompressed data sizes, exhibits notable fluctuations during the simulation. If the threshold were set constant across simulations, we would expect setups with larger grid sizes to have higher compression ratios, as discontinuities would form a smaller proportion of the grid as the grid size increases. However, since we normalize the threshold with equation (9.8), this relationship is not as straightforward.

At the beginning of the simulation, when the grid values remain largely constant, all setups show a higher compression ratio. This ratio then rapidly decreases to its minimum before fluctuating over time without displaying significant abrupt changes, continuing to vary without settling into a stable state. The fluctuation over time is influenced by the changing shapes of vortices throughout the simulation. The rapid decline at the beginning is due to the emergence of shock waves from the obstacle at the start of the simulation. These waves propagate through the domain, causing discontinuities that are captured by the wavelet transform, leading to a drop in compression ratio. It is important to understand that these shock waves are numerical artifacts rather than physical phenomena. They are caused by the abrupt change at the initial condition, which is not representative of the physical reality. These artifacts are avoidable through methods such as a gradual initialization process. As these shock waves vanish, the compression ratio stabilizes at a higher level, fluctuating over time but remaining above the minimum.

In our implementation, the lowest compression ratio acts as a bottleneck that may render a simulation unexecutable. This is particularly problematic as simulations often start with shock waves, resulting in temporary drops in compression ratio. However, a more sophisticated implementation could identify such scenarios and employ a slower yet more compressive scheme during the necessary time steps. For example,

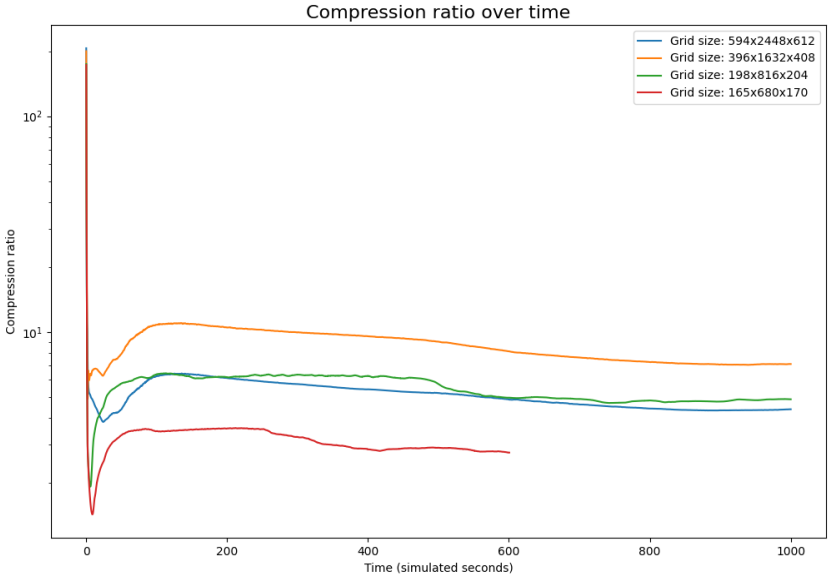


Figure 9.10: Compression ratio over time for the 3 tested grid sizes. The x-axis represents the time in (simulated) seconds, while the y-axis displays the compression ratio.

the global compression scheme can reach extremely high compression ratios, as we have shown in a previous work [97]. Alternatively, compressed subgrids could be stored on a storage device and retrieved when needed.

In conclusion, the results show that the compression scheme is able to run a simulation in a memory constrained environment, introducing an acceptable overhead in comparison to the best possible scenario using the PCIe bus. This compression overhead can even be adjusted downwards by performing multiple LBM steps per compression/decompression cycle, but this would introduce challenges regarding the synchronization of the LBM blocks. The observed compression ratios are high, typically between 5 and 10 for large grids, which validate the usefulness of the method. The trade-off between the compression ratio and the execution time is satisfactory, as the execution throughput is higher than the maximum PCIe bandwidths. Thus, the proposed compression scheme is a viable option in GPU memory-constrained configurations, in particular when the PCIe transfers are a bottleneck.

9.4 Conclusions

In this study, we have presented a novel approach to compressing large-scale CFD simulations on GPUs. Wavelet-based compression methods have been widely used in the past, but their application to GPU CFD simulations has been limited due to the memory intensity of the DWT. We have proposed a new method that leverages the SRAM of the GPU to perform local DWT and COO compression. This approach has been tested on a D3Q27 LBM simulation. The results show that it is possible to run simulations that would not be executable with a single GPU, due to the memory requirements. The tested D3Q27 simulation has an overall low execution time on GPUs, which makes it a less favorable candidate for compression. More computationally intensive simulations would presumably yield even better results, as the overhead of the compression would be less significant in comparison to the total execution time. Traditionally, compression is rarely integrated into CFD simulations due to its presumably bad compression ratio/execution time trade-off. However, our results show that it is possible to reach satistactory trade-off, which unlocks significant potential for the execution of larger-scale simulations. Our method can be be improved in different ways.

First, we have implemented a pipeline where the LBM step is performed once per compression/decompression cycle. It is, however, possible to perform the LBM step multiple times per compression/decompression

cycle. This would lower the overhead of the compression, as the compression would be performed less frequently. This technique is known as temporal blocking and is a common technique in stencil computations. It is normally used to improve the cache usage. Second, the used lossless compression algorithm has been chosen for its simplicity and speed, but it is likely that other algorithms could provide better compression ratios and/or execution times. In particular, for the tested D3Q27 scheme, where the sparsity of the data is not as high as expected due to the low threshold required for accurate results. Further works are being conducted to use better compression methods for near-sparse data. Third, the D3Q27 scheme is performed on the global memory, which is an unnecessary bottleneck. It is likely that the performance of the scheme could be improved by performing the LBM computations on the shared memory, but this introduces challenges regarding the synchronization of the LBM blocks. Finally, integrating our methodology into real-world multi-GPU frameworks is a natural next step. Multi-GPU LBM simulations are often bottlenecked by data accesses and transfers, both of which could be improved by our method.

Chapter 10

Conclusions

This thesis has systematically addressed the challenges of conducting large-scale CFD simulations on modern HPC platforms, with a particular focus on GPUs. Through a series of detailed investigations, we have presented novel strategies and optimizations that collectively aim to push the boundaries of current CFD simulation capabilities, particularly in the context of memory management.

10.1 Summary of Contributions

Our work began with a foundational overview of the different methods used in fluid simulations and how they translate to computational implementations. We have seen that the process of numerical discretization often leads to stencil algorithms, which are highly parallelizable and well-suited to GPU architectures. An important problem with the stencil approach is the memory requirements, which grows quadratically (cubically in 3D) with the resolution of the simulation. The usual approach to relieve this memory pressure is to use AMR, which reduces the memory footprint by breaking the regularity of the grid and focusing on the regions of interest. We, however, use another approach where we use explicit data compression to reduce the memory footprint.

In the context of HPC, it is not possible to focus solely on memory gains, as performance is also a critical factor. This is why we have spent a significant amount of time studying common techniques used for achieving high performance on GPUs with stencil computations. These techniques, summarized in Chapter 3, help ensuring that the computational resources of the system are used efficiently, maximizing the performance of the simulation. However, as the scale of the simulations grows, techniques for avoiding load imbalance and ensuring efficient scheduling become increasingly important. This is why we have focused on task-based runtime systems, which provide a flexible and efficient way to manage the computational resources of the system. PaRSEC and StarPU are two task-based runtime systems on which we have focused our attention, as they are widely used in the HPC community. In PaRSEC, we have developped a new feature on the PTG DSL to allow for more flexibility in the definition of the tasks and showed that it can be used to express a stencil computation in an elegant way with no visible cost on performance. In StarPU, we have developped a generic stencil solver that is able to run on a distributed environment. Finally, we have developped a new scheduler for StarPU, AutoHeteroprio, that is able to automatically adjust the priorities of the tasks to ensure that the system is used efficiently. These contributions aim to provide a baseline for the seamless integration of explicit compression within a distributed multi-GPU framework, thereby enabling the efficient execution of large-scale CFD simulations.

The rest of the thesis is dedicated to the design and implementation of a high-performance compression scheme for CFD simulations. In Chapter 7, we presented the preliminary work we performed to design wavelets for CFD simulations. In this work, careful consideration was given to the specific requirements of CFD simulations, such as the need for mass conservation and polynomial filtering. We then used these wavelets to design a lossy compression scheme for CFD simulations and observed promising results in terms of compression ratio and simulation accuracy. Finally, we conducted a last work to design a wavelet-based compression scheme with a focus on performance. We have shown that by leveraging the shared memory of

the GPU, our compression scheme can achieve satisfactory performance while maintaining a high compression ratio.

10.2 Future Directions

The work presented in this thesis opens up several avenues for future research.

First, there are several areas of improvement for the wavelet-based compression scheme. The DWT itself can be optimized to reduce the number of memory accesses and improve data locality. Currently, the DWT is performed successively on each dimension, which not optimal in terms of data locality. We are aware that multi-dimensional DWT algorithms can theoretically solve this issue, but we did not find any implementation that would work in practice with the GPU architecture.

The lossless compression method can also be improved. The current high-performance COO implementation appears to be fast, but relies on the sparsity of the data. This posed a problem in our D3Q27 experiment, where the data was less sparse than expected, leading to a lower compression ratio. This is probably due to the turbulent nature of the tested flow, which requires a low threshold to preserve the accuracy of the simulation. There are several other lossless compression methods that would presumably be less reliant on the sparsity of the data, such as entropy coding or dictionary-based methods. However, since the lossless compression must be integrated in our compression kernel, we must be able to provide a fast implementations, which we did not have time to do.

Another idea to improve performance is to pipeline the compression and the simulation. Currently, the decompression kernels write the decompressed data to the global memory, where the simulation kernels read and write them. Since the decompressed data appear at some point in the shared memory, it could be possible to directly perform the time step within the shared memory and then immediately compress it back to the global memory. This would significantly reduce the number of global memory accesses, because only the compressed data would be accessed in the global memory. However, is it not evident to find a new simulation design that would work with this idea. In particular, because of data access pattern, which requires to access the data of the neighboring blocks, which are not necessarily available.

Finally, another major area of improvement would be a flexible implementation of a large-scale CFD simulation that integrates the wavelet-based compression scheme. Currently, our StarPU multi-GPU implementation does not let the possibility of using compression, rendering it comparable to most state-of-the-art CFD solvers. However, in a next work, we plan on integrating the compression scheme into the solver, and to evaluate the performance impact of different strategies. We believe that a fully dynamic approach, where the subgrids can be reshaped depending on runtime factors, would be the most promising approach. Contrary to the more popular approaches, which rarely integrate compression, our approach would be able to perform better load balancing because of the reduced cost of sending compressed data. The use of temporal blocking would also be facilitate, as the increased synchronization cost would be offset by the reduced data transfer cost.

In conclusion, the work presented in this thesis provides a solid foundation for the integration of wavelet-based compression schemes into large-scale CFD simulations. The results obtained so far are promising, and we believe that with further research and development, it will be possible to achieve significant improvements in the performance and scalability of CFD simulations on modern HPC platforms.

10.3 Concluding Remarks

In this thesis, we have presented a series of novel strategies and optimizations that aim to push the boundaries of current CFD simulation capabilities, particularly in the context of memory management. Our work has focused on the design and implementation of a high-performance compression scheme for CFD simulations, with the goal of reducing the memory footprint of large-scale simulations while maintaining high levels of accuracy. We have shown that by leveraging the shared memory of the GPU, our compression scheme can achieve satisfactory performance while maintaining a high compression ratio. We believe that the results obtained so far are promising, and that with further research and development, it will be possible to achieve significant improvements in the performance and scalability of CFD simulations on modern HPC platforms.

Acknowledgement

Experiments presented in this report were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/). This work of the Interdisciplinary Thematic Institute IRMIA++, as part of the ITI 2021-2028 program of the University of Strasbourg, CNRS and Inserm, was supported by IdEx Unistra (ANR-10-IDEX-0002), and by SFRI-STRAT'US project (ANR-20-SFRI-0012) under the framework of the French Investments for the Future Program. This work was also supported by a Labex IRMIA (11-LABX-0055) grant.

Bibliography

[1] Anydsl - a partial evaluation framework for programming high-performance libraries.

[2] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, M. Gates, T. Grützmacher, N. J. Higham, S. Li, et al. A survey of numerical methods utilizing mixed precision arithmetic. *arXiv preprint arXiv:2007.06674*, 2020.

[3] S. Abdulah, Q. Cao, Y. Pei, G. Bosilca, J. Dongarra, M. G. Genton, D. E. Keyes, H. Ltaief, and Y. Sun. Accelerating geostatistical modeling and prediction with mixed-precision computations: A high-productivity approach with parsec. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):964–976, 2021.

[4] R. Abgrall. Multiresolution representation in unstructured meshes. *SIAM journal on numerical analysis*, 35(6):2128–2146, 1998.

[5] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Science & Business Media, 2008.

[6] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 932–943. IEEE, 2011.

[7] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In W. mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, Sept. 2010.

[8] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset. Bridging the gap between openmp and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2794–2807, 2017.

[9] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[10] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing*, 36(1):C66–C93, 2014.

[11] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-based fmm for heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, 28(9):2608–2629, 2016.

[12] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Multifrontal qr factorization for multicore architectures over runtime systems. In *European Conference on Parallel Processing*, pages 521–532. Springer, 2013.

[13] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Task-based multifrontal qr solver for gpu-accelerated multicore architectures. In *2015 IEEE 22nd international conference on high performance computing (HiPC)*, pages 54–63. IEEE, 2015.

[14] E. Agullo, M. Felšöci, and G. Sylvand. Direct solution of larger coupled sparse/dense linear systems using low-rank compression on single-node multi-core machines in an industrial context. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 25–35. IEEE, 2022.

[15] E. Agullo, L. Giraud, and S. Nakov. Task-based sparse hybrid linear solver for distributed memory heterogeneous architectures. In F. Desprez, P.-F. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. A. Vega-Rodríguez, A. L. Varbanescu, S. Hunold, S. L. Scott, S. Lankes, and J. Weidendorfer, editors, *Euro-Par 2016: Parallel Processing Workshops*, pages 83–95, Cham, 2017. Springer International Publishing.

[16] D. Angelis, F. Sofos, and T. E. Karakasidis. Artificial intelligence in physical sciences: Symbolic regression trends and perspectives. *Archives of Computational Methods in Engineering*, 30(6):3845–3865, Apr. 2023.

[17] D. Aregba-Driollet and R. Natalini. Discrete kinetic schemes for multidimensional systems of conservation laws. *SIAM Journal on Numerical Analysis*, 37(6):1973–2004, 2000.

[18] R. Arias Mallo. Particle-in-cell plasma simulation with ompss-2. Master's thesis, Universitat Politècnica de Catalunya, 2019.

[19] R. Artebrant and M. Torrilhon. Increasing the accuracy in locally divergence-preserving finite volume schemes for mhd. *Journal of computational physics*, 227(6):3405–3427, 2008.

[20] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*, pages 863–874. Springer, 2009.

[21] P. Auscher. Wavelets with boundary conditions on the interval. *Wavelets: A tutorial in theory and applications*, pages 217–236, 1992.

[22] J. Badwaik, M. Boileau, D. Coulette, E. Franck, P. Helluy, C. Klingenberg, L. Mendoza, and H. Oberlin. Task-based parallelization of an implicit kinetic scheme. *ESAIM: Proceedings and Surveys*, 63:60–77, 2018.

[23] P. Bailey, J. Myre, S. D. Walsh, D. J. Lilja, and M. O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *2009 international conference on parallel processing*, pages 550–557. IEEE, 2009.

[24] D. S. Balsara and J. Kim. A comparison between divergence-cleaning and staggered-mesh formulations for numerical magnetohydrodynamics. *The Astrophysical Journal*, 602(2):1079, 2004.

[25] C. M. Bard and J. C. Dorelli. A simple gpu-accelerated two-dimensional muscl-hancock solver for ideal magnetohydrodynamics. *Journal of Computational Physics*, 259:444–460, 2014.

[26] D. Bartlett, H. Desmond, and P. Ferreira. Priors for symbolic regression. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, GECCO '23 Companion, page 2402–2411, New York, NY, USA, 2023. Association for Computing Machinery.

[27] D. J. Bartlett, H. Desmond, and P. G. Ferreira. Exhaustive symbolic regression. *IEEE Transactions on Evolutionary Computation*, pages 1–1, 2023.

[28] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. *ACM sigplan notices*, 44(4):219–228, 2009.

[29] H. Baty, F. Drui, P. Helluy, E. Franck, C. Klingenberg, and L. Thanhäuser. A robust and efficient solver based on kinetic schemes for magnetohydrodynamics (mhd) equations. *Applied Mathematics and Computation*, 440:127667, 03 2023.

[30] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[31] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pages 1–32, 2011.

[32] O. Beaumont, L.-C. Canon, L. Eyraud-Dubois, G. Lucarelli, L. Marchal, C. Mommessin, B. Simon, and D. Trystram. Scheduling on two types of resources: A survey. *ACM Comput. Surv.*, 53(3), May 2020.

[33] T. Bellotti, L. Gouarin, B. Graille, and M. Massot. Multidimensional fully adaptive lattice boltzmann methods with error control based on multiresolution analysis. *Journal of Computational Physics*, 471:111670, 2022.

[34] T. Bellotti, L. Gouarin, B. Graille, and M. Massot. Multiresolution-based mesh adaptation and error control for lattice boltzmann methods with applications to hyperbolic conservation laws. *SIAM Journal on Scientific Computing*, 44(4):A2599–A2627, 2022.

[35] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics*, 53(3):484–512, 1984.

[36] M. Bernaschi, L. Rossi, R. Benzi, M. Sbragaglia, and S. Succi. Graphics processing unit implementation of lattice boltzmann models for flowing soft systems. *Physical Review E*, 80(6):066707, 2009.

[37] S. Bertoluzza and V. Perrier. A new construction of boundary interpolating wavelets for fourth order problems. *Acta Applicandae Mathematicae*, 152:33–56, 2017.

[38] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical review*, 94(3):511, 1954.

[39] L. Biggio, T. Bendinelli, A. Lucchi, and G. Parascandolo. A seq2seq approach to symbolic regression. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

[40] L. Biggio, T. Bendinelli, A. Neitz, A. Lucchi, and G. Parascandolo. Neural symbolic regression that scales. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 936–945. PMLR, 18–24 Jul 2021.

[41] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989.

[42] L. Boillot, G. Bosilca, E. Agullo, and H. Calandra. Task-based programming for seismic imaging: Preliminary results. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, pages 1259–1266. IEEE, 2014.

[43] S. Boopathiraja and P. Kalavathi. A near lossless multispectral image compression using 3d-dwt with application to landsat images. *Int J Comput Sci Eng*, 6(4):332–336, 2018.

[44] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al. Distibuted dense numerical linear algebra algorithms on massively parallel architectures: Dplasma. 2010.

[45] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1432–1441. IEEE, 2011.

[46] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[47] F. Bouchut. Construction of bgk models with a family of kinetic entropies for a given system of conservation laws. *Journal of Statistical Physics*, 95:113–170, 04 1999.

[48] F. Bouchut, Y. Jobic, R. Natalini, R. Occelli, and V. Pavan. Second-order entropy satisfying BGK-FVS schemes for incompressible Navier-Stokes equations. *SMAI Journal of Computational Mathematics*, 4:1–56, Mar. 2018.

[49] B. Bramas. *Optimization and parallelization of the boundary element method for the wave equation in time domain*. Theses, Université de Bordeaux, Feb. 2016.

[50] B. Bramas. Impact study of data locality on task-based applications through the heteroprio scheduler. *PeerJ Computer Science*, 5:e190, May 2019.

[51] B. Bramas, C. Flint, and L. Paillat. auto-heteroprio analysis. `https://gitlab.inria.fr/cflint/auto_heteroprio_analysis`, 2021.

[52] B. Bramas, P. Helluy, L. Mendoza, and B. Weber. Optimization of a discontinuous Galerkin solver with OpenCL and StarPU. *International Journal on Finite Volumes*, 15(1):1–19, Jan. 2020.

[53] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390, 1977.

[54] J. Brence, S. Džeroski, and L. Todorovski. Dimensionally-consistent equation discovery through probabilistic attribute grammars. *Information Sciences*, 632:742–756, 2023.

[55] J. Brence, L. Todorovski, and S. Džeroski. Probabilistic grammars for equation discovery. *Knowledge-Based Systems*, 224:107077, 2021.

[56] K. Brix, S. S. Melian, S. Müller, and G. Schieffer. Parallelisation of multiscale-based grid adaptation using space-filling curves. In *ESAIM: proceedings*, volume 29, pages 108–129. EDP Sciences, 2009.

[57] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[58] P. Brucker and S. Knust. *Complexity results for scheduling problems*, 2009. `http://www2.informatik.uni-osnabrueck.de/knust/class/`.

[59] J. Bruno, E. G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Commun. ACM*, 17(7):382–387, July 1974.

[60] S. L. Brunton, J. L. Proctor, and J. N. Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, mar 2016.

[61] C. Burstedde. Parallel tree algorithms for amr and non-standard data access. *ACM Transactions on Mathematical Software (TOMS)*, 46(4):1–31, 2020.

[62] C. Burstedde, D. Calhoun, K. T. Mandli, and A. R. Terrel. Forestclaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws. In M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 253–262. IOS Press, 2014.

[63] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications*, 33(6):1201–1220, 2019.

[64] J. M. C. Carpaye, J. Roman, and P. Brenner. Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping. *Journal of Computational Science*, 2018.

[65] R. Carratalá-Sáez, M. Faverge, G. Pichon, G. Sylvand, and E. S. Quintana-Ortí. Tiled algorithms for efficient task-parallel $\mathfrak{H}$-matrix solvers. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 757–766. IEEE, 2020.

[66] S. G. Chang, B. Yu, and M. Vetterli. Adaptive wavelet thresholding for image denoising and compression. *IEEE transactions on image processing*, 9(9):1532–1546, 2000.

[67] S. Chen and G. D. Doolen. Lattice boltzmann method for fluid flows. *Annual review of fluid mechanics*, 30(1):329–364, 1998.

[68] T. Chen, P. Xu, and H. Zheng. Bootstrapping ots-funcimg pre-training model (botfip) – a comprehensive symbolic regression framework, 2024.

[69] H. Choi, D. Son, S. Kang, J. Kim, H.-H. Lee, and C.-H. Kim. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *The Journal of Supercomputing*, 65, 08 2013.

[70] A. Cohen, W. Dahmen, and R. DeVore. Adaptive wavelet techniques in numerical simulation. *Encyclopedia of Computational Mechanics*, 1:157–197, 2004.

[71] A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on pure and applied mathematics*, 45(5):485–560, 1992.

[72] A. Cohen, I. Daubechies, and P. Vial. Wavelets on the interval and fast wavelet transforms. *Applied and computational harmonic analysis*, 1993.

[73] A. Cohen, S. Kaber, S. Müller, and M. Postel. Fully adaptive multiresolution finite volume schemes for conservation laws. *Mathematics of computation*, 72(241):183–225, 2003.

[74] Y. Collet. Lz4 - extremely fast compression.

[75] K. Coulomb, A. Degomme, M. Faverge, and F. Trahay. An open-source tool-chain for performance analysis. In *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*, pages 37–48. Springer, 2012.

[76] D. Coyle and L. Hampton. 21st century progress in computing. *Telecommunications Policy*, 48(1):102649, 2024.

[77] R. Dahlburg and J. Picone. Evolution of the orszag–tang vortex system in a compressible medium. i. initial average subsonic flow. *Physics of Fluids B: Plasma Physics*, 1(11):2153–2171, 1989.

[78] W. Dahmen, A. Kunoth, and K. Urban. Biorthogonal spline wavelets on the interval—stability and moment conditions. *Applied and computational harmonic analysis*, 6(2):132–196, 1999.

[79] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. Ptg: an abstraction for unhindered parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30. IEEE, 2014.

[80] V. Danjean, R. Namyst, and P.-A. Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lisbon, Portugal, August 30-September 2, 2005. Proceedings 11*, pages 166–175. Springer, 2005.

[81] S. d'Ascoli, S. Bengio, J. Susskind, and E. Abbé. Boolformer: Symbolic Regression of Logic Functions with Transformers. *arXiv e-prints*, page arXiv:2309.12207, Sept. 2023.

[82] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-point stencil for multicore. In *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, volume 70, 2009.

[83] I. Daubechies. Orthonormal bases of compactly supported wavelets. *Communications on pure and applied mathematics*, 41(7):909–996, 1988.

[84] I. Daubechies. *Ten lectures on wavelets*. SIAM, 1992.

[85] J. de Fine Licht, A. Kuster, T. De Matteis, T. Ben-Nun, D. Hofer, and T. Hoefler. Stencilflow: Mapping large stencil programs to distributed spatial computing systems. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 315–326. IEEE, 2021.

[86] A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer, and M. Wesenberg. Hyperbolic divergence cleaning for the mhd equations. *Journal of Computational Physics*, 175(2):645–673, 2002.

[87] D. d'Humières. Multiple–relaxation–time lattice boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 360(1792):437–451, 2002.

[88] S. Donfack, L. Grigori, W. D. Gropp, and V. Kale. Hybrid static/dynamic scheduling for already optimized dense matrix factorization. Research Report RR-7775, INRIA, Oct. 2011.

[89] D. L. Donoho. De-noising by soft-thresholding. *IEEE transactions on information theory*, 41(3):613–627, 1995.

[90] D. L. Donoho and I. M. Johnstone. Adapting to unknown smoothness via wavelet shrinkage. *Journal of the american statistical association*, 90(432):1200–1224, 1995.

[91] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.*, 9(3):302–325, Sept. 1983.

[92] V. Engelson, P. Fritzson, and D. Fritzson. *Lossless compression of high-volume numerical data from simulations*. LINKÖPING University Electronic Press, 2000.

[93] R. Eymard, T. Gallouët, and R. Herbin. Finite volume methods. *Handbook of numerical analysis*, 7:713–1018, 2000.

[94] J. G. Faris, C. F. Hayes, A. R. Goncalves, K. G. Sprenger, D. faissol, B. K. Petersen, M. Landajuela, and F. L. da Silva. Pareto front training for multi-objective symbolic optimization. In *The Sixteenth Workshop on Adaptive and Learning Agents*, 2024.

[95] C. Flint and B. Bramas. Finding new heuristics for automated task prioritizing in heterogeneous computing. working paper, Nov. 2020.

[96] C. Flint, B. Bramas, S. Genaud, and P. Helluy. Parallelization of the Lattice-Boltzmann schemes using the task-based method. In *COMPAS 2022 - Conférence francophone d'informatique en Parallélisme, Architecture et Système*, Amiens, France, July 2022.

[97] C. Flint and P. Helluy. Reducing the memory usage of lattice-boltzmann schemes with a dwt-based compression. *ESAIM: Proceedings*, 2022.

[98] C. Flint, L. Paillat, and B. Bramas. Automated prioritizing heuristics for parallel task graph scheduling in heterogeneous computing. *PeerJ Computer Science*, 8:e969, 2022.

[99] N. Fout and K.-L. Ma. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2295–2304, 2012.

[100] L. Freret and C. P. Groth. Anisotropic non-uniform block-based adaptive mesh refinement for three-dimensional inviscid and viscous flows. In *22nd AIAA Computational Fluid Dynamics Conference*, page 2613, 2015.

[101] L. Freret, L. Ivan, H. De Sterck, and C. P. Groth. High-order finite-volume method with block-based amr for magnetohydrodynamics flows. *Journal of Scientific Computing*, 79:176–208, 2019.

[102] M. N. Gamito and M. S. Dias. Lossless coding of floating point data with jpeg 2000 part 10. In *Applications of Digital Image Processing XXVII*, volume 5558, pages 276–287. SPIE, 2004.

[103] U. Ganse, T. Koskela, M. Battarbee, Y. Pfau-Kempf, K. Papadakis, M. Alho, M. Bussov, G. Cozzani, M. Dubart, H. George, et al. Enabling technology for global 3d+ 3v hybrid-vlasov simulations of near-earth space. *Physics of Plasmas*, 30(4), 2023.

[104] M. Geier and M. Schönherr. Esoteric twist: An efficient in-place streaming algorithmus for the lattice boltzmann method on massively parallel hardware. *Computation*, 5(2):19, 2017.

[105] L. Gouarin, B. Graille, M. Mancini, and B. Thibert. pylbm, 2020.

[106] A. Grossmann and J. Morlet. Decomposition of hardy functions into square integrable wavelets of constant shape. *SIAM journal on mathematical analysis*, 15(4):723–736, 1984.

[107] R. Guimerà, I. Reichardt, A. Aguilar-Mogas, F. A. Massucci, M. Miranda, J. Pallarès, and M. Sales-Pardo. A bayesian machine scientist to aid in the solution of challenging scientific problems. *Science Advances*, 6(5):eaav6971, 2020.

[108] L. Han, T. Indinger, X. Hu, and N. A. Adams. Wavelet-based adaptive multi-resolution solver on heterogeneous parallel architecture for computational fluid dynamics. *Computer Science-Research and Development*, 26:197–203, 2011.

[109] M. Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.

[110] A. Harten. Multiresolution representation of data: A general framework. *SIAM Journal on Numerical Analysis*, 33(3):1205–1256, 1996.

[111] M. Hasert, K. Masilamani, S. Zimny, H. Klimach, J. Qi, J. Bernsdorf, and S. Roller. Complex fluid simulations with the parallel tree-based lattice boltzmann solver musubi. *Journal of Computational Science*, 5(5):784–794, 2014.

[112] Y. He, B. Sheng, and Z. Li. Channel modeling based on transformer symbolic regression for inter-satellite terahertz communication. *Applied Sciences*, 14(7), 2024.

[113] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, 2002.

[114] G. Herschlag, S. Lee, J. S. Vetter, and A. Randles. Analysis of gpu data access patterns on complex geometries for the d3q19 lattice boltzmann algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2400–2414, 2021.

[115] N. J. Higham and T. Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 31:347–414, 2022.

[116] S. Holt, Z. Qian, and M. van der Schaar. Deep generative symbolic regression. In *The Eleventh International Conference on Learning Representations*, 2023.

[117] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2017.

[118] T. Y. Hou and P. G. LeFloch. Why nonconservative schemes converge to wrong solutions: error analysis. *Mathematics of computation*, 62(206):497–530, 1994.

[119] Z. Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery*, 2(3):283–304, 1998.

[120] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Computer Graphics Forum*, volume 22-3, pages 343–348. Wiley Online Library, 2003.

[121] T. Isaac, C. Burstedde, and O. Ghattas. Low-cost parallel algorithms for 2: 1 octree balance. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 426–437. IEEE, 2012.

[122] M. Jacquelin, M. Araya-Polo, and J. Meng. Massively scalable stencil algorithm. *arXiv preprint arXiv:2204.03775*, 2022.

[123] Y. Jiang, Z. Shao, and Y. Guo. A dag scheduling scheme on heterogeneous computing systems using tuple-based chemical reaction optimization. *The Scientific World Journal*, 2014, 2014.

[124] G. Jin, J. Lin, and T. Endo. Efficient utilization of memory hierarchy to enable the computation on bigger domains for stencil computation in cpu-gpu based systems. In *2014 International Conference on High Performance Computing and Applications (ICHPCA)*, pages 1–6. IEEE, 2014.

[125] Y. Jin, W. Fu, J. Kang, J. Guo, and J. Guo. Bayesian symbolic regression. *arXiv preprint arXiv:1910.08892*, 2019.

[126] T. Johnson and V. Patel. Flow past a sphere up to a reynolds number of 300. *Journal of Fluid Mechanics*, 378:19–70, 1999.

[127] P.-A. Kamienny, S. d'Ascoli, G. Lample, and F. Charton. End-to-end symbolic regression with transformers. In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[128] P.-A. Kamienny and S. Lamprier. Symbolic-model-based reinforcement learning. In *NeurIPS 2022 AI for Science: Progress and Promises*, 2022.

[129] H. Kang, D. Lee, and D. Lee. A study on cfd data compression using hybrid supercompact wavelets. *KSME international journal*, 17:1784–1792, 2003.

[130] M. A. Khan. Scheduling for heterogeneous systems using constrained critical paths. *Parallel Computing*, 38(4):175–193, 2012.

[131] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18–27, 1993.

[132] J. T. Kim, M. Landajuela, and B. K. Petersen. Distilling wikipedia mathematical knowledge into neural network models. In *1st Mathematical Reasoning in General Artificial Intelligence, International Conference on Learning Representations (ICLR)*, 2021.

[133] S. Kim, P. Y. Lu, S. Mukherjee, M. Gilbert, L. Jing, V. Čeperić, and M. Soljačić. Integration of neural network-based symbolic regression in deep learning for scientific discovery. *IEEE transactions on neural networks and learning systems*, 32(9):4166–4177, 2020.

[134] F. Knorr, P. Thoman, and T. Fahringer. ndzip-gpu: efficient lossless compression of scientific floating-point data on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[135] M. Kommenda, B. Burlacu, G. Kronberger, and M. Affenzeller. Parameter identification for symbolic regression using nonlinear least squares. *Genetic Programming and Evolvable Machines*, 21(3):471–501, 2020.

[136] E. Konstantinidis and Y. Cotronis. A quantitative performance evaluation of fast on-chip memories of gpus. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 448–455. IEEE, 2016.

[137] M. J. Krause, A. Kummerländer, S. J. Avis, H. Kusumaatmaja, D. Dapelo, F. Klemens, M. Gaedtke, N. Hafen, A. Mink, R. Trunk, et al. Openlb—open source lattice boltzmann code. *Computers & Mathematics with Applications*, 81:258–288, 2021.

[138] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *ACM sigplan notices*, 42(6):235–244, 2007.

[139] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen. The lattice boltzmann method. *Springer International Publishing*, 10(978-3):4–15, 2017.

[140] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, Dec. 1999.

[141] W. La Cava, P. Orzechowski, B. Burlacu, F. de Franca, M. Virgolin, Y. Jin, M. Kommenda, and J. Moore. Contemporary symbolic regression methods and their relative performance. In J. Vanschoren and S. Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1. Curran, 2021.

[142] X. Lacoste. *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu duster systems.* PhD thesis, Université de Bordeaux, 2015.

[143] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 29–38. IEEE, 2014.

[144] F. Lalande, Y. Matsubara, N. Chiba, T. Taniai, R. Igarashi, and Y. Ushiku. A Transformer Model for Symbolic Regression towards Scientific Discovery. *arXiv e-prints*, page arXiv:2312.04070, Dec. 2023.

[145] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 100(9):690–691, 1979.

[146] M. Landajuela, C. S. Lee, J. Yang, R. Glatt, C. P. Santiago, I. Aravena, T. Mundhenk, G. Mulcahy, and B. K. Petersen. A unified framework for deep symbolic regression. *Advances in Neural Information Processing Systems*, 35:33985–33998, 2022.

[147] M. Landajuela, B. K. Petersen, S. K. Kim, C. P. Santiago, R. Glatt, T. N. Mundhenk, J. F. Pettit, and D. M. Faissol. Improving exploration in policy gradient search: Application to symbolic optimization. In *1st Mathematical Reasoning in General Artificial Intelligence, International Conference on Learning Representations (ICLR)*, 2021.

[148] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman. Devito: Towards a generic finite difference dsl using symbolic python. In *2016 6th workshop on python for high-performance and scientific computing (PyHPC)*, pages 67–75. IEEE, 2016.

[149] H. P. Langtangen and S. Linge. *Finite difference computing with PDEs: a modern software approach.* Springer Nature, 2017.

[150] V. Lanore and C. Pérez. A reconfigurable component model for hpc. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, pages 1–10, 2015.

[151] J. Latt. How to implement your ddqq dynamics with only q variables per node (instead of 2q). In *tech. rep.* Tufts University, 2007.

[152] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M. B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, et al. Palabos: parallel lattice boltzmann solver. *Computers & Mathematics with Applications*, 81:334–350, 2021.

[153] D. Le Gall and A. Tabatabai. Sub-band coding of digital images using symmetric short kernel filters and arithmetic coding techniques. In *ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing*, pages 761–764. IEEE, 1988.

[154] M. Lehmann. Esoteric pull and esoteric push: Two simple in-place streaming schemes for the lattice boltzmann method on gpus. *Computation*, 10(6):92, 2022.

[155] M. Lehmann, M. J. Krause, G. Amati, M. Sega, J. Harting, and S. Gekle. Accuracy and performance of the lattice boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats. *Physical Review E*, 106(1):015308, 2022.

[156] J. Y.-T. Leung and G. H. Young. Minimizing schedule length subject to minimum flow time. *SIAM J. Comput.*, 18(2):314–326, Apr. 1989.

[157] R. J. LeVeque. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[158] M. Li, Y. Liu, H. Yang, Y. Hu, Q. Sun, B. Chen, X. You, X. Liu, Z. Luan, and D. Qian. Automatic code generation and optimization of large-scale stencil computation on many-core processors. In *Proceedings of the 50th International Conference on Parallel Processing*, pages 1–12, 2021.

[159] S. Li, I. Marinescu, and S. Musslick. GFN-SR: Symbolic regression with generative flow networks. In *NeurIPS 2023 AI for Science Workshop*, 2023.

[160] Y. Li, W. Li, L. Yu, M. Wu, J. Liu, W. Li, M. Hao, S. Wei, and Y. Deng. Discovering mathematical formulas from data via gpt-guided monte carlo tree search, 2024.

[161] Y. Li, J. Liu, W. Li, L. Yu, M. Wu, W. Li, M. Hao, S. Wei, and Y. Deng. Mmsr: Symbolic regression is a multimodal task, 2024.

[162] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, et al. Sz3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data*, 9(2):485–498, 2022.

[163] H. Liao, M. K. Mandal, and B. F. Cockburn. Efficient architectures for 1-d and 2-d lifting-based wavelet transforms. *IEEE Transactions on Signal Processing*, 52(5):1315–1326, 2004.

[164] T. Liebchen. Mpeg-4 als-the standard for lossless audio coding. *the Journal of the Acoustical Society of Korea*, 28(7):618–629, 2009.

[165] J. V. Lima, G. Freytag, V. G. Pinto, C. Schepke, and P. O. Navaux. A dynamic task-based d3q19 lattice-boltzmann method for heterogeneous architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 108–115. IEEE, 2019.

[166] H. Lin, M.-F. Li, C.-F. Jia, J.-N. Liu, and H. An. Degree-of-node task scheduling of fine-grained parallel programs on heterogeneous systems. *Journal of Computer Science and Technology*, 34(5):1096–1108, 2019.

[167] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[168] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[169] Y. Liu, Y. Wang, L. Deng, F. Wang, F. Liu, Y. Lu, and S. Li. A novel in situ compression method for cfd data based on generative adversarial network. *Journal of Visualization*, 22:95–108, 2019.

[170] F. Lopez. *Task-based multifrontal QR solver for heterogeneous architectures*. PhD thesis, Université Paul Sabatier-Toulouse III, 2015.

[171] F. Lopez and I. Duff. Task-based sparse direct solver for symmetric indefinite systems. *PMAA, TASK-BASED PROGRAMMING FOR SCIENTIFIC COMPUTING MS, Zurich, Switzerland*, 2018.

[172] C. Luo, C. Chen, and Z. Jiang. Divide and conquer: A quick scheme for symbolic regression. *International Journal of Computational Methods*, 19(08):2142002, 2022.

[173] J. Luo, X. Li, M. Yuan, J. Yao, and J. Zeng. Learning to optimize dag scheduling in heterogeneous environment, 2021.

[174] N. Makke and S. Chawla. Interpretable scientific discovery with symbolic regression: A review. *arXiv preprint arXiv:2211.10873*, 2022.

[175] S. Mallat. *A wavelet tour of signal processing*. Elsevier, 1999.

[176] Z. Mao, X. Li, S. Hu, G. Gopalakrishnan, and A. Li. A gpu accelerated mixed-precision smoothed particle hydrodynamics framework with cell-based relative coordinates. *Engineering Analysis with Boundary Elements*, 161:113–125, 2024.

[177] I. Marinescu, Y. Strittmatter, C. Williams, and S. Musslick. Expression sampler as a dynamic benchmark for symbolic regression. In *NeurIPS 2023 AI for Science Workshop*, 2023.

[178] V. Martínez, D. Michéa, F. Dupros, O. Aumage, S. Thibault, H. Aochi, and P. O. Navaux. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 1–8. IEEE, 2015.

[179] G. Martius and C. H. Lampert. Extrapolation and learning equations, 2017.

[180] Y. Matsubara, N. Chiba, R. Igarashi, and Y. Ushiku. SRSD: Rethinking datasets of symbolic regression for scientific discovery. In *NeurIPS 2022 AI for Science: Progress and Promises*, 2022.

[181] A. K. Maurya and A. K. Tripathi. On benchmarking task scheduling algorithms for heterogeneous computing systems. *The Journal of Supercomputing*, 74(7):3039–3070, 2018.

[182] G. G. McNamara and G. Zanetti. Use of the boltzmann equation to simulate lattice-gas automata. In *Lattice Gas Methods For Partial Differential Equations*, pages 289–296. CRC Press, 2019.

[183] K. Meidani, P. Shojaee, C. K. Reddy, and A. B. Farimani. SNIP: Bridging mathematical symbolic and numeric realms with unified pre-training. In *The Twelfth International Conference on Learning Representations*, 2024.

[184] D. Melching, F. Paysan, T. Strohmann, and E. Breitbarth. A universal crack tip correction algorithm discovered by physical deep symbolic regression, 2024.

[185] J. C. Meyera, T. Arslana, J. Valstadb, J. Ragunathanb, N. Brownc, and L. Cebamanosc. Lbm and sph scalability using task-based programming.

[186] Y. Michishita. Alpha zero for physics: Application of symbolic regression with alpha zero to find the analytical methods in physics, 2024.

[187] H. Midorikawa, H. Tan, and T. Endo. An evaluation of the potential of flash ssd as large and slow memory for stencil computations. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 268–277. IEEE, 2014.

[188] N. Miki, F. Ino, and K. Hagihara. Pacc: a directive-based programming framework for out-of-core stencil computation on accelerators. *International Journal of High Performance Computing and Networking*, 13(1):19–34, 2019.

[189] S. Mittal and J. S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015.

[190] M. Mohrhard, G. Thäter, J. Bludau, B. Horvat, and M. J. Krause. Auto-vectorization friendly parallel lattice boltzmann streaming scheme for direct addressing. *Computers & Fluids*, 181:1–7, 2019.

[191] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[192] S. Müller. *Adaptive multiscale schemes for conservation laws*, volume 27. Springer Science & Business Media, 2002.

[193] S. Müller and Y. Stiriba. Fully adaptive multiscale schemes for conservation laws employing locally varying time stepping. *Journal of scientific computing*, 30(3):493–531, 2007.

[194] T. Muranushi and J. Makino. Optimal temporal blocking for stencil computation. *Procedia Computer Science*, 51:1303–1312, 2015.

[195] S. M. Najmabadi, P. Offenhäuser, M. Hamann, G. Jajnabalkya, F. Hempert, C. W. Glass, and S. Simon. Analyzing the effect and performance of lossy compression on aeroacoustic simulation of gas injector. *Computation*, 5(2):24, 2017.

[196] P. Neumann, H.-J. Bungartz, M. Mehl, T. Neckel, and T. Weinzierl. A coupled approach for fluid dynamic problems using the pde framework peano. *Communications in Computational Physics*, 12(1):65–84, 2012.

[197] R. Nourgaliev, S. Wiri, N. Dinh, and T. Theofanous. On improving mass conservation of level set by reducing spatial discretization errors. *International journal of multiphase flow*, 31(12):1329–1336, 2005.

[198] NVIDIA. nvbandwidth. `https://github.com/NVIDIA/nvbandwidth`, 2024.

[199] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, 61(12):3628–3638, 2011.

[200] S. A. Orszag and C.-M. Tang. Small-scale structure of two-dimensional magnetohydrodynamic turbulence. *Journal of Fluid Mechanics*, 90(1):129–143, 1979.

[201] S. Oswal, A. Singh, and K. Kumari. Deflate compression algorithm. *International Journal of Engineering Research and General Science*, 4(1):430–436, 2016.

[202] R. Ouyang, S. Curtarolo, E. Ahmetcik, M. Scheffler, and L. M. Ghiringhelli. Sisso: A compressed-sensing method for identifying the best low-dimensional descriptor in an immensity of offered candidates. *Phys. Rev. Mater.*, 2:083802, Aug 2018.

[203] M. Panju and A. Ghodsi. A neuro-symbolic method for solving differential and functional equations. *arXiv preprint arXiv:2011.02415*, 2020.

[204] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. *Parallel lossless data compression on the GPU*. IEEE, 2012.

[205] Y. Pei, Q. Cao, G. Bosilca, P. Luszczek, V. Eijkhout, and J. Dongarra. Communication avoiding 2d stencil implementations over parsec task-based runtime. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 721–729. IEEE Computer Society, 2020.

[206] J. Pekkilä, M. S. Väisälä, M. J. Käpylä, M. Rheinhardt, and O. Lappi. Scalable communication for high-order stencil computations using cuda-aware mpi. *Parallel Computing*, 111:102904, 2022.

[207] B. K. Petersen, M. L. Larma, T. N. Mundhenk, C. P. Santiago, S. K. Kim, and J. T. Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations*, 2021.

[208] B. K. Petersen, C. Santiago, and M. Landajuela. Incorporating domain knowledge into neural-guided search via in situ priors and constraints. In *8th ICML Workshop on Automated Machine Learning (AutoML)*, 2021.

[209] J. M. Picone and R. B. Dahlburg. Evolution of the orszag–tang vortex system in a compressible medium. ii. supersonic flow. *Physics of Fluids B: Plasma Physics*, 3(1):29–44, 1991.

[210] R. Prat, T. Carrard, L. Soulard, O. Durand, R. Namyst, and L. Colombet. Amr-based molecular dynamics for non-uniform, highly dynamic particle simulations. *Computer Physics Communications*, 253:107177, 2020.

[211] T. A. Purcell, M. Scheffler, and L. M. Ghiringhelli. Recent advances in the sisso method and their implementation in the sisso++ code. *arXiv preprint arXiv:2305.01242*, 2023.

[212] T. M. Quan and W.-K. Jeong. A fast mixed-band lifting wavelet transform on the gpu. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 1238–1242. IEEE, 2014.

[213] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[214] S. M. F. Rahman, Q. Yi, and A. Qasem. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, pages 1–10, 2011.

[215] P. Ramachandran and G. Varoquaux. Mayavi: 3d visualization of scientific data. *Computing in Science & Engineering*, 13(2):40–51, 2011.

[216] N. Ranganathan and S. Henriques. High-speed vlsi designs for lempel-ziv-based data compression. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 40(2):96–106, 1993.

[217] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142. IEEE, 2006.

[218] E. Raut, J. Anderson, M. Araya-Polo, and J. Meng. Evaluation of distributed tasks in stencil-based application on gpus. In *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 45–52. IEEE, 2021.

[219] E. Raut, J. Anderson, M. Araya-Polo, and J. Meng. Porting and evaluation of a distributed task-driven stencil-based application. In *Proceedings of the 12th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 21–30, 2021.

[220] E. Raut, J. Meng, M. Araya-Polo, and B. Chapman. Evaluating performance of openmp tasks in a seismic stencil application. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems: 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings 16*, pages 67–81. Springer, 2020.

[221] M. Ravishankar, J. Holewinski, and V. Grover. Forma: A dsl for image processing applications to target gpus and multi-core cpus. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 109–120, 2015.

[222] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Register optimizations for stencils on gpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–182, 2018.

[223] C. W. Rowley, I. Mezić, S. Bagheri, P. Schlatter, and D. S. Henningson. Spectral analysis of nonlinear flows. *Journal of fluid mechanics*, 641:115–127, 2009.

[224] M. L. Sætra, A. R. Brodtkorb, and K.-A. Lie. Efficient gpu-implementation of adaptive mesh refinement for the shallow-water equations. *Journal of Scientific Computing*, 63:23–48, 2015.

[225] S. Sahoo, C. Lampert, and G. Martius. Learning equations for extrapolation and control. In *International Conference on Machine Learning*, pages 4442–4450. PMLR, 2018.

[226] R. Sakai, D. Sasaki, and K. Nakahashi. Parallel implementation of large-scale cfd data compression toward aeroacoustic analysis. *Computers & Fluids*, 80:116–127, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[227] R. Sakai, D. Sasaki, S. Obayashi, and K. Nakahashi. Wavelet-based data compression for flow simulation on block-structured cartesian mesh. *International Journal for Numerical Methods in Fluids*, 73(5):462–476, 2013.

[228] S. Sakane, T. Aoki, and T. Takaki. Parallel-gpu amr implementation for phase-field lattice boltzmann simulation of a settling dendrite. *Computational Materials Science*, 211:111542, 2022.

[229] N. Sakharnykh, D. LaSalle, and B. Karsin. Optimizing data transfer using lossless compression with nvidia nvcomp, 2020.

[230] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros. Dendro: parallel algorithms for multigrid and amr methods on 2: 1 balanced octrees. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2008.

[231] V. Sathish, M. J. Schulte, and N. S. Kim. Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 325–334, 2012.

[232] K. Sato, H. Takizawa, K. Komatsu, and H. Kobayashi. Automatic tuning of cuda execution parameters for stencil processing. *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, pages 209–228, 2010.

[233] M. T. Satria, B. Huang, T.-J. Hsieh, Y.-L. Chang, and W.-Y. Liang. Gpu acceleration of tsunami propagation model. *IEEE Journal of Selected Topics in Applied Earth Observations and remote Sensing*, 5(3):1014–1023, 2012.

[234] H.-Y. Schive, Y.-C. Tsai, and T. Chiueh. Gamer: a graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 186(2):457, 2010.

[235] H.-Y. Schive, U.-H. Zhang, and T. Chiueh. Directionally unsplit hydrodynamic schemes with hybrid mpi/openmp/gpu parallelization in amr. *The International Journal of High Performance Computing Applications*, 26(4):367–377, 2012.

[236] J. Schmalzl. Using standard image compression algorithms to store data from computational fluid dynamics. *Computers & geosciences*, 29(8):1021–1031, 2003.

[237] M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009.

[238] M. Schmidt and H. Lipson. *Age-Fitness Pareto Optimization*, pages 129–146. Springer New York, New York, NY, 2011.

[239] S. Schmieschek, L. Shamardin, S. Frijters, T. Krüger, U. D. Schiller, J. Harting, and P. V. Coveney. Lb3d: A parallel implementation of the lattice-boltzmann method for simulation of interacting amphiphilic fluids. *Computer physics communications*, 217:149–161, 2017.

[240] M. Schmitt, P. Helluy, and C. Bastoul. Adaptive code refinement: A compiler technique and extensions to generate self-tuning applications. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 172–181. IEEE, 2017.

[241] M. Schmitt, P. Helluy, and C. Bastoul. Automatic adaptive approximation for stencil computations. In *Proceedings of the 28th International Conference on Compiler Construction*, pages 170–181, 2019.

[242] M. Schmitt, C. Sabater, and C. Bastoul. Semi-automatic generation of adaptive codes. In *IMPACT 2017-7th International Workshop on Polyhedral Compilation Techniques*, pages 1–7, 2017.

[243] K. Schneider and O. V. Vasilyev. Wavelet methods in computational fluid dynamics. *Annual review of fluid mechanics*, 42:473–503, 2010.

[244] P. Scholl, K. Bieker, H. Hauger, and G. Kutyniok. ParFam – Symbolic Regression Based on Continuous Global Optimization. *arXiv e-prints*, page arXiv:2310.05537, Oct. 2023.

[245] F. Schornbaum and U. Rüde. Extreme-scale block-structured adaptive mesh refinement. *SIAM Journal on Scientific Computing*, 40(3):C358–C387, 2018.

[246] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[247] J. Shen, X. Deng, Y. Wu, M. Okita, and F. Ino. Compression-based optimizations for out-of-core gpu stencil computation. *arXiv preprint arXiv:2204.11315*, 2022.

[248] J. Shen, F. Ino, A. Farrés, and M. Hanzich. A data-centric directive-based framework to accelerate out-of-core stencil computation on a gpu. *IEICE TRANSACTIONS on Information and Systems*, 103(12):2421–2434, 2020.

[249] M.-Y. Shih, J.-W. Jheng, L.-F. Lai, et al. A two-step method for clustering mixed categroical and numeric data. *Journal of Applied Science and Engineering*, 13(1):11–19, 2010.

[250] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross. Massively-parallel lossless data decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 242–247. IEEE, 2016.

[251] A. Skodras, C. Christopoulos, and T. Ebrahimi. The jpeg 2000 still image compression standard. *IEEE Signal processing magazine*, 18(5):36–58, 2001.

[252] C. M. Stein. Estimation of the mean of a multivariate normal distribution. *The annals of Statistics*, pages 1135–1151, 1981.

[253] T. Stephens. Gplearn, 2015.

[254] Q. F. Stout, D. L. De Zeeuw, T. I. Gombosi, C. P. Groth, H. G. Marshall, and K. G. Powell. Adaptive blocks: A high performance data structure. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–10, 1997.

[255] M. Stéphane. Chapter 7 - wavelet bases. In M. Stéphane, editor, *A Wavelet Tour of Signal Processing (Third Edition)*, pages 263–376. Academic Press, Boston, third edition edition, 2009.

[256] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Numerical Mathematics and Scientific Computation. Clarendon Press, Oxford, 2001.

[257] D. Sukkari, H. Ltaief, M. Faverge, and D. Keyes. Asynchronous task-based polar decomposition on single node manycore architectures. *IEEE Transactions on parallel and distributed systems*, 29(2):312–323, 2017.

[258] W. Sweldens. Lifting scheme: a new philosophy in biorthogonal wavelet constructions. In *Wavelet applications in signal and image processing III*, volume 2569, pages 68–79. SPIE, 1995.

[259] W. Sweldens and P. Schröder. Building your own wavelets at home. In *Wavelets in the Geosciences*, pages 72–107. Springer, 2000.

[260] Z. Taghavi and S. Kasaei. A memory efficient algorithm for multi-dimensional wavelet transform based on lifting. In *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03).*, volume 6, pages VI–401. IEEE, 2003.

[261] H. Tayeb, B. Bramas, M. Faverge, and A. Guermouche. Dynamic tasks scheduling with multiple priorities on heterogeneous computing systems. *IEEE Heterogeneity in Computing Workshop (HCW'24), IPDPS 2024*, 2024.

[262] W. Tenachi, R. Ibata, and F. I. Diakogiannis. Deep symbolic regression for physics guided by units constraints: toward the automated discovery of physical laws. *arXiv e-prints*, page arXiv:2303.03192, Mar. 2023.

[263] W. Tenachi, R. Ibata, and F. I. Diakogiannis. Physical Symbolic Optimization. *arXiv e-prints*, page arXiv:2312.03612, Dec. 2023.

[264] W. Tenachi, R. Ibata, T. L. François, and F. I. Diakogiannis. Class Symbolic Regression: Gotta Fit 'Em All. *arXiv e-prints*, page arXiv:2312.01816, Dec. 2023.

[265] S. Thibault. On runtime systems for task-based programming on heterogeneous platforms. habilitationa diriger des recherches, université de bordeaux (dec 2018).

[266] N. Thiery. *Matrix: JGD_Forest/TF16*, 2008. `https://www.cise.ufl.edu/research/sparse/matrices/JGD_Forest/TF16.html`.

[267] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018.

[268] Y. Tian, W. Zhou, H. Dong, D. S. Kammer, and O. Fink. Sym-q: Adaptive symbolic regression via sequential decision-making, 2024.

[269] S. S. Tiwari, E. Pal, S. Bale, N. Minocha, A. W. Patwardhan, K. Nandakumar, and J. B. Joshi. Flow past a single stationary sphere, 2. regime mapping and effect of external disturbances. *Powder Technology*, 365:215–243, 2020.

[270] T. Tohme, D. Liu, and K. YOUCEF-TOUMI. GSR: A generalized symbolic regression approach. *Transactions on Machine Learning Research*, 2023.

[271] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[272] M. Torrilhon. Non-uniform convergence of finite volume schemes for riemann problems of ideal magnetohydrodynamics. *Journal of Computational Physics*, 192(1):73–94, 2003.

[273] M. Torrilhon. Locally divergence-preserving upwind finite volume schemes for magnetohydrodynamic equations. *SIAM Journal on Scientific Computing*, 26(4):1166–1191, 2005.

[274] A. Trott, R. Moorhead, and J. McGinley. Wavelets applied to lossless compression and progressive transmission of floating point data in 3-d curvilinear grids. In *Proceedings of Seventh Annual IEEE Visualization'96*, pages 385–388. IEEE, 1996.

[275] S.-M. Udrescu, A. Tan, J. Feng, O. Neto, T. Wu, and M. Tegmark. Ai feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. *Advances in Neural Information Processing Systems*, 33:4860–4871, 2020.

[276] S.-M. Udrescu and M. Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

[277] M. Usama and I.-Y. Lee. Data-driven non-linear current controller based on deep symbolic regression for spmsm. *Sensors*, 22(21):8240, 2022.

[278] P. Valero-Lara. Leveraging the performance of lbm-hpc for large sizes on gpus using ghost cells. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 417–430. Springer, 2016.

[279] P. Valero-Lara. Reducing memory requirements for large size lbm simulations on gpus. *Concurrency and Computation: Practice and Experience*, 29(24):e4221, 2017.

[280] P. Valero-Lara, A. Pinelli, J. Favier, and M. P. Matias. Block tridiagonal solvers on heterogeneous architectures. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 609–616. IEEE, 2012.

[281] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias. Fast finite difference poisson solvers on heterogeneous architectures. *Computer Physics Communications*, 185(4):1265–1272, 2014.

[282] C. M. C. Valle and S. Haddadin. Syrenets: Symbolic residual neural networks. *arXiv preprint arXiv:2105.14396*, 2021.

[283] M. Van de Velden, A. Iodice D'Enza, and A. Markos. Distance-based clustering of mixed data. *Wiley Interdisciplinary Reviews: Computational Statistics*, 11(3):e1456, 2019.

[284] M. Vastl, J. Kulhánek, J. Kubalík, E. Derner, and R. Babuška. Symformer: End-to-end symbolic regression using transformer-based architecture. *arXiv preprint arXiv:2205.15764*, 2022.

[285] M. Virgolin, T. Alderliesten, and P. A. Bosman. Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression. In *Proceedings of the genetic and evolutionary computation conference*, pages 1084–1092, 2019.

[286] M. Virgolin, T. Alderliesten, C. Witteveen, and P. A. N. Bosman. Improving Model-Based Genetic Programming for Symbolic Regression of Small Expressions. *Evolutionary Computation*, 29(2):211–237, 06 2021.

[287] M. Wahib and N. Maruyama. Automated gpu kernel transformations in large-scale production stencil applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 259–270, 2015.

[288] M. Wahib, N. Maruyama, and T. Aoki. Daino: a high-level framework for parallel and efficient amr on gpus. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 621–632. IEEE, 2016.

[289] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice boltzmann kernels. *Computers & Fluids*, 35(8-9):910–919, 2006.

[290] Y. Wen, Z. Wang, and M. F. P. O'Boyle. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014.

[291] C. Wilstrup and J. Kasak. Symbolic regression outperforms other models for small data sets. *arXiv preprint arXiv:2103.15147*, 2021.

[292] Y. Wiseman. Burrows-wheeler based jpeg. *Data Science Journal*, 6:19–27, 2007.

[293] M. Wittmann, G. Hager, and G. Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–7. IEEE, 2010.

[294] D. A. Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: an introduction.* Springer, 2004.

[295] T. Wu and M. Tegmark. Toward an artificial intelligence physicist for unsupervised learning. *Physical Review E*, 100(3):033311, 2019.

[296] Y. Xu, K. Li, J. Hu, and K. Li. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Information Sciences*, 270:255–287, 2014.

[297] G. Yang, Y. Chen, S. Chen, and M. Wang. Implementation of a direct-addressing based lattice boltzmann gpu solver for multiphase flow in porous media. *Computer Physics Communications*, 291:108828, 2023.

[298] K. Yoshikawa, S. Tanaka, and N. Yoshida. A 400 trillion-grid vlasov simulation on fugaku supercomputer: large-scale distribution of cosmic relic neutrinos in a six-dimensional phase space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2021.

[299] Y. Yu, D. H. Rudd, Z. Lan, N. Y. Gnedin, A. Kravtsov, and J. Wu. Improving parallel io performance of cell-based amr cosmology applications. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 933–944. IEEE, 2012.

[300] Z. Yu and L.-S. Fan. An interaction potential based lattice boltzmann method with adaptive mesh refinement (amr) for two-phase flow simulation. *Journal of Computational Physics*, 228(17):6456–6478, 2009.

[301] Yu-Kwong Kwok and I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[302] S. Zabelok, R. Arslanbekov, and V. Kolobov. Multi-gpu kinetic solvers using mpi and cuda. In *AIP Conference Proceedings*, volume 1628, pages 539–546. American Institute of Physics, 2014.

[303] A. Zakirov, A. Perepelkina, V. Levchenko, and S. Khilkov. Streaming techniques: revealing the natural concurrency of the lattice boltzmann method. *The Journal of Supercomputing*, 77:11911–11929, 2021.

[304] B. Zhang, J. Tian, S. Di, X. Yu, Y. Feng, X. Liang, D. Tao, and F. Cappello. Fz-gpu: A fast and high-ratio lossy compressor for scientific computing applications on gpus. *arXiv preprint arXiv:2304.12557*, 2023.

[305] N. Zhang, M. Driscoll, C. Markley, S. Williams, P. Basu, and A. Fox. Snowflake: A lightweight portable stencil dsl. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 795–804. IEEE, 2017.

[306] Y. Zhao. Lattice boltzmann based pde solver on the gpu. *The visual computer*, 24:323–333, 2008.

[307] W. Zheng, S. Sharan, Z. Fan, K. Wang, Y. Xi, and Z. Wang. Symbolic visual reinforcement learning: A scalable framework with object-level abstraction and differentiable expression search. *arXiv preprint arXiv:2212.14849*, 2022.

[308] J. Zhou, T. Wei, M. Chen, J. Yan, X. S. Hu, and Y. Ma. Thermal-aware task scheduling for energy minimization in heterogeneous real-time mpsoc systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(8):1269–1282, 2016.

[309] B. Zhu, Y. Guan, and J. Wu. Two-relaxation time lattice boltzmann models for the ion transport equation in electrohydrodynamic flow: D2q5 vs d2q9 and d3q7 vs d3q27. *Physics of Fluids*, 33(4), 2021.

[310] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)*, 23(4):550–560, 1997.

[311] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Chapter 11

Appendix

11.1 Heteroprio

In this section, we provide additional details related to our work from Chapter 6. In Section 11.1.1, we present an example of a DAG and the associated costs to illustrate the theoretical principle of Heteroprio. In Section 11.1.2, we provide the manual priorities used for the results presented in Section 6.3.2.

11.1.1 Heteroprio execution example

To understand the theoretical principle of Heteroprio, let us consider the example DAG shown in Figure 11.1 and the associated costs of Table 11.1.

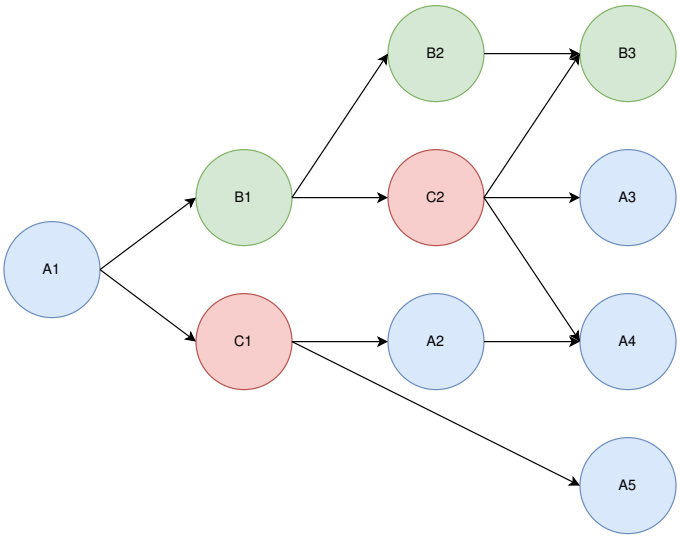


Figure 11.1: Example of a DAG with three task types (blue, red, and green).

There are three task types (A, B, and C). We assume that within a type, all tasks have the same costs. Let us consider a case where there are 2 CPU workers and 1 GPU worker. For the sake of simplicity, let us assume that the tasks are selected in a predefined order: CPU-1 pops a task if there is one available, then GPU-1, and then CPU-2. In practice, this order is not known in advance. In a real StarPU execution, there is a "prefetch" mechanism that ensures that a worker can start the job immediately after the dependencies are satisfied.

Intuitively, "A" tasks seem to be better suited for CPUs, which execute them twice as fast, whereas "B" tasks seem better suited for GPUs. The "C" tasks do not, seem to have particular affinities. In our model,

Task \ Architecture	CPU	GPU
A	1s	2s
B	2s	1s
C	1s	1s

Table 11.1: Example execution times of for the two processing unit types (CPU/GPU) and the three task types (A, B, and C).

whatever priorities we set, A1 is always executed by CPU-1. Also, C2 seemingly has great importance, since it has three successors.

Let us test what happens under three different priority lists. In Table 11.2, we show three different test cases.

Case \ Architecture	CPU	GPU
1	B-C-A	A-C-B
2	A-C-B	B-C-A
3	C-A-B	B-C-A

Table 11.2: Example priorities for a configuration of two processing unit types (CPU/GPU) and three task types (A, B, and C).

In case 1, B is the highest-priority task type on CPUs and A is the lowest one. On GPUs, the priorities are reversed. For both processors, C is the median priority. In this first case, the slowest architectures are intentionally promoted. For case 2 and case 3, we promote the fastest architectures. The difference between the two is that the priorities in case 2 are mirrored compared to the ones in case 1, whereas in case 3, we exchange the C and A types in the CPU. The idea of this swap is to favor the execution of C2 which has numerous successors.

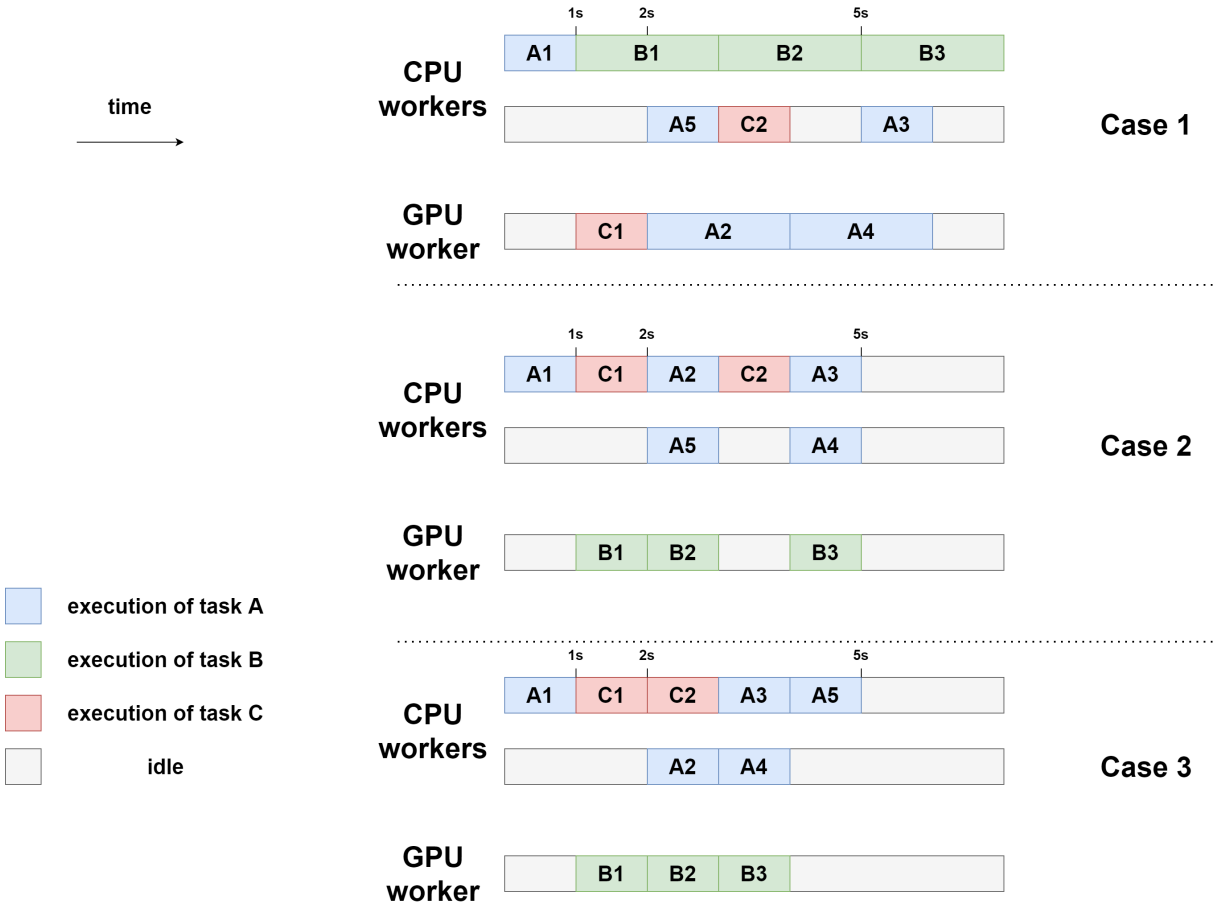


Figure 11.2: Example executions for the 3 different priority settings using two CPU workers and one GPU worker.

The three executions are schematized in Figure 11.2. In this model, the makespan of case 1 is lower (7s) than the one of case 2 and case 3 (5s). Here, case 2 and case 3 are equivalent in terms of makespan. In case 3, however, CPU-2 and GPU-1 are freed sooner (after 4s of execution) than in case 2, where they are still working after 5s of execution. Case 3 can, therefore, be seen as potentially better. This emphasizes the difficulty of finding heuristics automatically. Indeed, some tasks should be prioritized depending on their execution time, but others should be prioritized because they have particular importance in the execution graph (as C in our example).

11.1.2 Manual priority settings

For the results we provide in section 6.3.2, the non-automatic Heteroprio executions use manual priorities. These priorities are selected from a careful benchmark for each application. We follow different strategies for choosing them and we provide the different priorities that we test: Table 11.3 for POTRF, Table 11.4 for GEMM, Table 11.5 for GEQRF, and Table 11.6 for PaStiX. For QrMumps and Scalfmm, we use the already existing priorities set in the code.

Name	CPU Priorities	Cuda priorities	Slowdown factors		
			trsm	syrk	gemm
base	portf - splgsy - trsm - syrk - gemm	trsm - syrk – gemm	11.0	26.0	29.0
inverted-CPU	trsm - syrk - gemm - portf - splgsy	trsm - syrk – gemm	11.0	26.0	29.0
inverted-GPU	portf - splgsy - gemm - syrk - trsm	gemm - syrk – trsm	11.0	26.0	29.0
low-factors	**portf - splgsy - trsm - syrk - gemm**	**trsm - syrk – gemm**	**2.0**	**2.0**	**4.0**
high-factors	portf - splgsy - trsm - syrk - gemm	trsm - syrk – gemm	25.0	45.0	49.0

Table 11.3: Tested priorities and slowdown factors for the POTRF operation (Chameleon's Cholesky factorization). Rows in bold are the priorities used in our benchmarks.

Name	CPU Priorities	Cuda priorities	Slowdown factors gemm
base	plrnt - gemm	gemm	29.0
inverted	gemm - plrnt	gemm	29.0
low-factors	**plrnt - gemm**	**gemm**	**1.0**
high-factors	plrnt - gemm	gemm	40.0

Table 11.4: Tested priorities and slowdown factors for the GEMM operation (Chameleon's matrix/matrix multiplication). Rows in bold are the priorities used in our benchmarks.

Name	CPU Priorities	Cuda priorities	Slowdown factors	
			ormqr	tpmqrt
base	**geqrt - tpqrt - plrnt - lacpy - laset - ormqr - tmpqrt**	**ormqr - tmpqrt**	**10.0**	**10.0**
inverted_CPU	ormqr - tmpqrt - geqrt - tpqrt - plrnt - lacpy - laset	ormqr - tmpqrt	10.0	10.0
inverted_others	lacpy - laset - geqrt - tpqrt - plrnt - ormqr - tmpqrt	ormqr - tmpqrt	10.0	10.0
low-factors	geqrt - tpqrt - plrnt - lacpy - laset - ormqr - tmpqrt	ormqr - tmpqrt	2.0	2.0
high-factors	geqrt - tpqrt - plrnt - lacpy - laset - ormqr - tmpqrt	ormqr - tmpqrt	22.0	22.0

Table 11.5: Tested priorities and slowdown factors for the GEQRF operation (Chameleon's QR factorization). Rows in bold are the priorities used in our benchmarks.

Name	CPU Priorities	Slowdown factors		
		cblk_gemm	blok_trsm	blok_gemm
base	solve_blok_{trsm - gemm} - cblk_{getrf1d - gemm} - blok_{getrf - trsm - gemm}	1.0	10.0	10.0
better_factors	**solve_blok_{trsm - gemm} - cblk_{getrf1d - gemm} - blok_{getrf - trsm - gemm}**	**4.0**	**2.0**	**3.0**
inverted_groups	blok_{getrf - trsm - gemm} - cblk_{getrf1d - gemm} - solve_blok_{trsm - gemm}	1.0	10.0	10.0
better_factors_higher	solve_blok_{trsm - gemm} - cblk_{getrf1d - gemm} - blok_{getrf - trsm - gemm}	5.0	3.0	4.0
low-factors	blok_{getrf - trsm - gemm} - cblk_{getrf1d - gemm} - solve_blok_{trsm - gemm}	1.0	1.0	1.5
high-factors	blok_{getrf - trsm - gemm} - cblk_{getrf1d - gemm} - solve_blok_{trsm - gemm}	5.0	15.0	15.0
	Cuda priorities			
base	cblk_gemm - blok_{trsm - gemm}	-	-	-
better_factors	**cblk_gemm - blok_{trsm - gemm}**	**-**	**-**	**-**
inverted_groups	blok_{trsm - gemm} - cblk_gemm	-	-	-
better_factors_higher	cblk_gemm - blok_{trsm - gemm}	-	-	-
low-factors	blok_{trsm - gemm} - cblk_gemm	-	-	-
high-factors	blok_{trsm - gemm} - cblk_gemm	-	-	-

Table 11.6: Tested priorities and slowdown factors for the PaStiX. Rows in bold are the priorities used in our benchmarks.

11.2 Wavelet properties

This section provides a general overview of common properties of wavelets that are relevant to signal processing applications. More elaborate works can be found in the literature, such as the works of Mallat [255] and Daubechies [84]. This sections aim to provide a more brief and applied overview of the properties of wavelets.

11.2.1 Compact Support

Wavelet basis functions can be compactly supported, which means that these functions are nonzero only within a limited interval or region. This characteristic offers several advantages in signal processing applications, such as reduced computational complexity, efficient representation of localized features, and precise localization of signal components.

Mathematically, this property can be expressed as follows:

$$\psi(t) = \begin{cases} 0 & \text{for } t < a \text{ or } t > b \\ \text{nonzero function} & \text{for } a \leq t \leq b \end{cases} \tag{11.1}$$

The benefits of compact support include more efficient computations by limiting the operational range to a finite interval, which is particularly advantageous in real-time applications. This confinement helps accurately capture and localize transient signals and sharp features, essential for tasks like denoising and feature extraction. In the context of data compression, especially in CFD, this property ensures that discontinuities in the data affect a limited number of wavelet coefficients, leading to more efficient compression. Overall, the more compact the support of the wavelet basis function, the better it can capture localized features and provide a more efficient representation of the signal.

11.2.2 Symmetry

Symmetry is a property that some wavelet basis functions can exhibit, enhancing their effectiveness in certain signal processing applications. Mathematically, the symmetry property of wavelets can be expressed as:

$$\psi(t) = \pm\psi(-t) \tag{11.2}$$

where $\psi(t)$ represents the wavelet function. The sign ($\pm$) indicates that the wavelet function can be positive or negative, depending on the required symmetry.

This property is significant as detailed in the work of Daubechies [84], particularly for its role in applications involving human visual perception, which tends to be more forgiving of symmetric errors.

However, the utility of symmetric wavelets is not universally applicable across all signal processing tasks. Beyond visual applications, the benefits are generally limited and may introduce complexities, especially when enforcing symmetry in signals that are confined to a limited interval, such as images. This enforcement can lead to boundary artifacts or distortions.

In summary, while symmetry in wavelet basis functions can be advantageous in applications reliant on visual processing due to the human visual system, it presents limited advantages and potential challenges in other areas. Practitioners must evaluate the appropriateness of symmetric wavelets based on the specific needs of their applications and the inherent properties of the signals involved.

11.2.3 Orthogonal and Biorthogonal Basis

The orthogonality property of wavelet basis functions is defined as follows:

$$\langle \psi_m(t), \psi_n(t) \rangle = \int_{-\infty}^{\infty} \psi_m(t)\psi_n(t)\, dt = 0, \tag{11.3}$$

where $\psi_m(t)$ and $\psi_n(t)$ represent two wavelet basis functions. The orthogonality property plays a crucial role in the wavelet scheme for several reasons. Firstly, it ensures that the wavelet decomposition provides a

faithful representation of the signal, as different wavelet basis functions do not interfere or overlap with each other. This allows for a clean and accurate decomposition of the signal into different frequency bands.

Furthermore, the orthogonality property facilitates efficient signal reconstruction. Since the inner product between different wavelet basis functions is zero, it implies that the contribution of each basis function to the reconstructed signal can be isolated and reconstructed individually. This property enables perfect reconstruction of the original signal from the wavelet coefficients. However, in some cases, strict orthogonality between wavelet basis functions can be too restrictive. In practical applications, it is often more desirable to use pairs of wavelets that are orthogonal within a pair but not necessarily orthogonal to each other. Such pairs of wavelets are known as biorthogonal wavelets.

Biorthogonal wavelets offer a flexible and essential framework within wavelet schemes for signal processing. Unlike strictly orthogonal wavelets, biorthogonal wavelet pairs satisfy specific biorthogonality conditions, ensuring accurate and efficient signal decomposition and reconstruction. In the biorthogonal wavelet transform, the signal undergoes a process of decomposition and reconstruction using analysis filters and synthesis filters. The analysis filters, associated with the *father wavelets* ϕ and their corresponding approximation coefficients (a_k), capture the low-frequency components of the signal. On the other hand, the synthesis filters, associated with the *mother wavelets* ψ and their detail coefficients (d_k), capture the high-frequency components. Mathematically, the original signal x can be expressed as a linear combination of the father and mother wavelets:

$$x(t) = \sum_{k=0}^{N-1} a_k \phi_k(t) + \sum_{k=0}^{N-1} d_k \psi_k(t), \tag{11.4}$$

where N represents the number of wavelet pairs, a_k represents the approximation coefficients, d_k represents the detail coefficients, and $\phi_k(t)$ and $\psi_k(t)$ denote the corresponding wavelet functions. The left sum can be view as a coarse approximation of the original signal, while right sum captures the fine details. By incorporating both father and mother wavelets, biorthogonal wavelets enable accurate signal analysis and reconstruction, making them highly valuable in various signal processing applications.

Overall, biorthogonal wavelets represent a flexible and widely used approach in signal processing. The distinction between the analysis and synthesis filters can be leveraged to optimize the wavelet transform for specific applications. In particular, the approximation coefficient can be used again as input to an additional wavelet transform, allowing for a hierarchical decomposition of the signal. This approach is known as *Multiresolution Analysis* and is discussed in more detail in Chapter 7.

11.2.4 Vanishing Moments

Vanishing moments are a fundamental concept in wavelet theory that play a significant role in signal analysis and processing. This property is closely related to the ability of a wavelet to filter out polynomial trends in a signal. The nth moment of a wavelet ψ is defined as the integral of the function multiplied by a polynomial of order n:

$$M_n(\psi) = \int_{-\infty}^{\infty} p(t) \cdot \psi(t)\, dt, \tag{11.5}$$

where $p(t)$ represents a polynomial of order n. When this moment equals zero, it signifies that ψ is orthogonal to polynomials of order n. In other words, the wavelet can effectively filter out polynomial trends of order n from the signal. The number of vanishing moments possessed by a wavelet determines the maximum order of polynomials that can be filtered out.

A wavelet with n vanishing moments satisfies the following condition:

$$\int_{-\infty}^{\infty} p(t) \cdot \psi(t)\, dt = 0, \quad \text{for } k = 0, 1, \ldots, n-1. \tag{11.6}$$

For example, a wavelet with one vanishing moment can effectively remove constant components, while a wavelet with two vanishing moments can eliminate linear trends. When a signal contains polynomial components, the wavelet transform generates null detail coefficients, which can be discarded during reconstruction. This enables the analysis to focus on the remaining non-polynomial features, such as abrupt changes, edges,

textures, or singularities. In the context of data compression, which is our primary application, the vanishing moments property allows to effectively filter out polynomial trends, leading to more efficient representation of the signal. In principle, reaching higher orders of vanishing moments comes with a higher computational cost. It is, therefore, essential to weigh the benefits of filtering out higher-order polynomial trends against the computational complexity of the wavelet transform.

11.2.5 Mass Conservation

Throughout this work, we have emphasized the importance of mass conservation in the context of wavelet transforms. In general, the wavelet theory focuses on the conservation of signal energy, ensuring that the total energy of the signal is preserved during the wavelet decomposition and reconstruction. It is defined as follows:

$$\sum_i |\alpha_i|^2 = \sum_i |\beta_i|^2 , \tag{11.7}$$

However, this property is not particularly relevant for our purposes. What is relevant for our work is the conservativeness of the overall scheme. This property is crucial in most numerical schemes that are based on physical phenomena, as it ensures that the total mass of the system is preserved and is often required for the stability of the scheme.

Depending on the shape of the wavelet, the mass conservation property can be expressed in different ways. For the LGT 5/3 wavelets we built in Section 7.2.3, the mass conservation property is given by equation 7.17. This equation sums all the wavelet coefficients apart from the first and last ones, which are halved. Figure 11.3 provides a visual interpretation of this formula. The reason is basically that the first and last wavelets are only half within the interval, and therefore, only half of their mass is considered in the sum. Since the wavelets associated with the detail coefficients have a vanishing integral, the wavelet scheme is conservative whatever change we make to the detail coefficients.

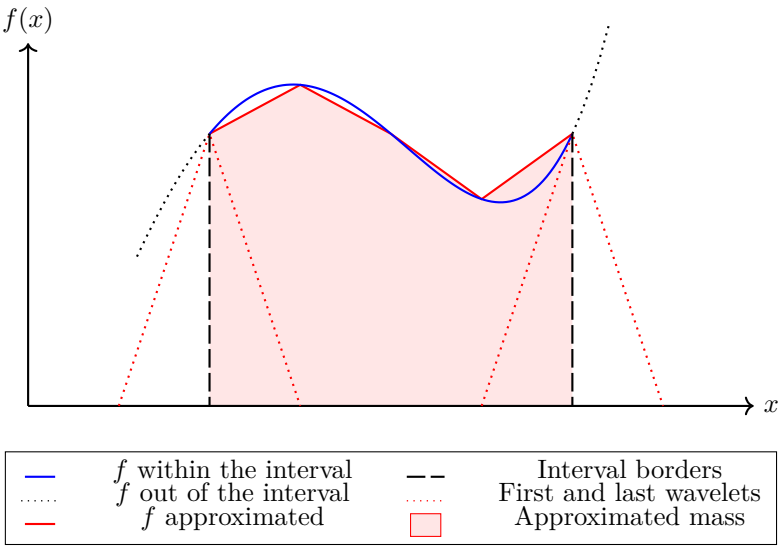


Figure 11.3: Visual interpretation of the trapezoidal quadrature formula. For clarity, only the first and last wavelets are represented as red dotted lines. The function f is depicted by the blue curve. It is approximated by the sum of five wavelets (red line) on an interval (black dashed line). Notably, only half of the mass of the first and last wavelets lies within the interval, leading to their halved weights in the trapezoidal quadrature formula Equation 7.17.

11.3 Wavelet Transform and Mass Conservation

11.3.1 Results for the CDF 9/7 and Haar wavelets

This section completes the results provided in Chapter 7 by providing the same results as in Section 7.3.2 but for the CDF 9/7 and lifted Haar wavelets (Section 7.3.2 only presented the results for the LGT5/3 wavelets). Table 11.7 and Table 11.8 present the results for the CDF 9/7 wavelets, while Table 11.9 and Table 11.10 present the results for the lifted Haar wavelets.

Thresholding	Entropy			Relative error		Max error
	Before DWT	After DWT	After threshold	eq. 7.38	eq. 7.39	
none	7.22	1.29	1.29	$3.11e^{-15}$	$2.22e^{-16}$	$6.22e^{-15}$
0.0001	7.22	1.29	0.40	$6.44e^{-15}$	$-8.66e^{-15}$	$1.37e^{-4}$
0.001	7.22	1.29	0.38	$6.22e^{-15}$	$9.77e^{-15}$	$1.73e^{-3}$
0.01	7.22	1.29	0.34	$3.11e^{-15}$	$2.66e^{-15}$	$1.91e^{-2}$
0.1	7.22	1.29	0.28	$7.11e^{-15}$	$1.33e^{-15}$	$1.79e^{-1}$
1.0	7.22	1.29	0.23	$7.11e^{-15}$	$-3.22e^{-15}$	$9.82e^{-1}$
all	7.22	1.29	0.23	$5.55e^{-15}$	$-3.22e^{-15}$	$7.31e^{-1}$
random	7.22	1.29	6.74	$2.22e^{-15}$	$-5.77e^{-15}$	8.67

Table 11.7: Entropy and mass deviation for different thresholding methods on the first function, CDF9/7 wavelets, and hard thresholding.

Thresholding	Entropy			Relative error		Max error
	Before DWT	After DWT	After threshold	eq. 7.38	eq. 7.39	
none	6.51	0.21	0.21	$-2.22e^{-16}$	0	$1.78e^{-15}$
0.0001	6.51	0.21	0.21	$6.66e^{-16}$	$-2.22e^{-16}$	$1.08e^{-4}$
0.001	6.51	0.21	0.21	$1.11e^{-15}$	0	$1.37e^{-3}$
0.01	6.51	0.21	0.20	$6.66e^{-16}$	$-2.22e^{-16}$	$9.02e^{-3}$
0.1	6.51	0.21	0.20	$6.66e^{-16}$	0	$1.31e^{-2}$
1.0	6.51	0.21	0.20	$6.66e^{-16}$	0	$1.31e^{-2}$
all	6.51	0.21	0.20	$6.66e^{-16}$	0	$1.31e^{-2}$
random	6.51	0.21	6.78	$2.22e^{-16}$	$2.22e^{-16}$	9.04

Table 11.8: Entropy and mass deviation for different thresholding methods on the second function, CDF9/7 wavelets, and hard thresholding.

Thresholding	Entropy			Relative error		Max error
	Before DWT	After DWT	After threshold	eq. 7.38	eq. 7.39	
none	7.22	1.29	1.29	$6.22e^{-15}$	$6.88e^{-15}$	$4.44e^{-15}$
0.0001	7.22	1.29	0.36	$6.66e^{-15}$	$-6.28e^{-7}$	$1.52e^{-4}$
0.001	7.22	1.29	0.36	$6.44e^{-15}$	$-1.19e^{-6}$	$1.56e^{-3}$
0.01	7.22	1.29	0.34	$6.66e^{-15}$	$2.17e^{-5}$	$2.96e^{-2}$
0.1	7.22	1.29	0.28	$4.22e^{-15}$	$-7.13e^{-4}$	$2.69e^{-1}$
1.0	7.22	1.29	0.23	$6.88e^{-15}$	$-6.78e^{-4}$	$9.33e^{-1}$
all	7.22	1.29	0.23	$6.88e^{-15}$	$-6.78e^{-4}$	$8.15e^{-1}$
random	7.22	1.29	6.75	$4.88e^{-15}$	$-4.65e^{-1}$	$1.50e^{1}$

Table 11.9: Entropy and mass deviation for different thresholding methods on the first function, Haar wavelets, and hard thresholding.

Thresholding	Entropy			Relative error		Max error
	Before DWT	After DWT	After threshold	eq. 7.38	eq. 7.39	
none	6.51	0.23	0.23	0	0	$1.55e^{-15}$
0.0001	6.51	0.23	0.23	$2.22e^{-16}$	$-5.48e^{-9}$	$1.79e^{-4}$
0.001	6.51	0.23	0.22	0	$-2.90e^{-9}$	$2.26e^{-3}$
0.01	6.51	0.23	0.20	$2.22e^{-16}$	$8.08e^{-9}$	$1.10e^{-2}$
0.1	6.51	0.23	0.19	$2.22e^{-16}$	$8.08e^{-9}$	$2.15e^{-2}$
1.0	6.51	0.23	0.19	$2.22e^{-16}$	$8.08e^{-9}$	$2.15e^{-2}$
all	6.51	0.23	0.19	$2.22e^{-16}$	$8.08e^{-9}$	$2.15e^{-2}$
random	6.51	0.23	6.78	0	$1.30e^{-3}$	$1.44e^{1}$

Table 11.10: Entropy and mass deviation for different thresholding methods on the second function, Haar wavelets, and hard thresholding.

The observations made for the LGT 5/3 wavelets can also be made for these results, but we can add some additional comments. To conduct the same experiment for the lifter Haar wavelets, we had to resize the grid to 1024×1024, as the scheme requires a power of two (while the LGT 5/3 and CDF 9/7 wavelets require a power of two plus one). While the two ways of summing (with of without the 1/2 on the borders) are mass-conserving for the LGT 5/3 and CDF 9/7 wavelets, they are not for the Haar wavelets. We, hence, observe experimentally both ways of summing are inherently different and that the LGT 5/3 and CDF 9/7 wavelets are coincidentally mass-conserving for both ways of summing. Finally, we observe that the Haar wavelets are only mass-conserving whithout the 1/2 on the borders (equation 7.38).

11.3.2 Proof of mass conservation for the LGT 5/3 and CDF 9/7 wavelets

In the following, we provide a proof of mass conservation with regards to equation 7.38 for the LGT 5/3 and CDF 9/7 wavelets. It should be noted that the mass conservation with regards to equation 7.39, is already already achieved because the wavelet coefficients (equation 7.18) have been chosen specifically to ensure this property. We rely on the following notations/assumptions:

- For the sake of simpliciy, we assume a size of N for the grid on all dimensions (the proof can easily be adapted to any size);
- The wavelet transform is an application T from $\mathcal{R}^N$ to $\mathcal{R}^N$;
- The wavelet transform exactly preserves the border values:

$$T(v_1) = v_1 \quad \text{and} \quad T(v_N) = v_N, \tag{11.8}$$

  where v is a vector of size N;
- There exists an inverse wavelet transform T^{-1} that does **not** necessarily ensure $T^{-1}(T(v)) = v$.
- The deviation of each value i of the wavelet transform is defined as

$$\delta_i = T^{-1}(T(v))_i - v_i; \tag{11.9}$$

- The inverse transform ensures

$$T^{-1}(v_1) = v_1 \quad \text{and} \quad T^{-1}(v_N) = v_N \tag{11.10}$$

  and

$$\sum_{i=1}^{N} \omega_i \delta_i = 0, \tag{11.11}$$

  which is derived from equation 7.39, with ω_i is $\frac{1}{2}$ for $i = 1$ and $i = N$ and 1 otherwise;

- The mass conservation, as defined in equation 7.38, is ensured if

$$\sum_{i=1}^{N} \delta_i = 0. \tag{11.12}$$

The goal of the proofs is to show that the mass conservation as defined in equation 11.12 is ensured.

Proof of mass conservation for one dimension without $1/2$ on the borders By combining equation 11.8 and equation 11.10, we know that

$$T^{-1}(T(v))_1 = v_1 \quad \text{and} \quad T^{-1}(T(v))_N = v_N. \tag{11.13}$$

Hence, equation 11.9 can be rewritten as

$$\delta_1 = T^{-1}(T(v))_1 - v_1 = v_1 - v_1 = 0 \quad \text{and} \quad \delta_N = T^{-1}(T(v))_N - v_N = v_N - v_N = 0. \tag{11.14}$$

Equation 11.11 can, therefore, be rewritten without the borders:

$$\sum_{i=2}^{N-1} \delta_i = 0, \tag{11.15}$$

notably without the ω_i coefficients which are equal to 1 for $i \neq 1$ and $i \neq N$. We can add δ_1 and δ_N, which we showed are equal to 0 in equation 11.14:

$$\delta_1 + \sum_{i=2}^{N-1} \delta_i + \delta_N = 0, \tag{11.16}$$

which can be rewritten as

$$\sum_{i=1}^{N} \delta_i = 0, \tag{11.17}$$

which is the mass conservation as defined in equation 11.12.

Proof of mass conservation for multi-d without $1/2$ on the borders We will now reason with tensors of dimension d rather than vectors. The proof for $d = 1$ (with a vector) has been provided in the previous paragraph. Let us assume that mass conservation (as defined in equation 11.12) is ensured for $d = d'$ and show this implies mass conservation for $d = d' + 1$.

Let $v^{(d')}$ be a tensor of dimension d' and let $V^{(d'+1)}$ be a tensor of dimension $d' + 1$. This tensor $V^{(d'+1)}$ can be viewed as a vector of tensors $v^{(d')}$. We refer to each of the $v^{(d')}$ tensors as a slice of $V^{(d'+1)}$. We assumed that conservation holds for $v^{(d')}$, i.e.,

$$\sum_{i_1, i_2, \ldots, i_{d'}} \delta_{i_1, i_2, \ldots, i_{d'}} = 0. \tag{11.18}$$

This property holds for each slice of $V^{(d'+1)}$.

As we stated in Section 7.3.1, the multi-dimensional wavelet transform can be described as a successive application of one-dimensional wavelet transforms along each dimension. In our case, this means an application along the first dimension, followed by an application along the second dimension, and so on up to the d'-th dimension. The addition of the $(d' + 1)$-th dimension involves performing the last wavelet transform on each line of the $d' + 1$-th dimension, which lets apply the same reasoning as in equation 11.13 and equation 11.14, namely that since the borders of the $(d' + 1)$-th dimension are preserved, they do not contribute to the mass deviation:

$$\delta_{i_1, i_2, \ldots, i_{d'}, 1} = 0 \quad \text{and} \quad \delta_{i_1, i_2, \ldots, i_{d'}, N} = 0. \tag{11.19}$$

Then, by summing equation 11.18 for each slice of the $(d'+1)$-th dimension apart from the borders, we obtain

$$\sum_{i_1=1,i_2=1,\ldots,i_{d'}=1,i_{d'+1}=2}^{N,N,\ldots,N,N-1} \delta^{\text{before}}_{i_1,i_2,\ldots,i_{d'},i_{d'+1}} = 0, \tag{11.20}$$

where $\delta^{\text{before}}_{i_1,i_2,\ldots,i_{d'},i_{d'+1}}$ is the deviation before the last wavelet transform. After applying the last wavelet transform along the $(d'+1)$-th dimension, the sum of the masses does not change. This is because the same arguments as in equation 11.15 (one-dimensional case) can be applied to each line of the $(d'+1)$-th dimension. We can, hence, write

$$\sum_{i_1=1,i_2=1,\ldots,i_{d'}=1,i_{d'+1}=2}^{N,N,\ldots,N,N-1} \delta_{i_1,i_2,\ldots,i_{d'},i_{d'+1}} = 0, \tag{11.21}$$

where $\delta_{i_1,i_2,\ldots,i_{d'},i_{d'+1}}$ is the deviation after the last wavelet transform.

Finally, by summing equation 11.19 and equation 11.21, we obtain

$$\sum_{i_1=1,i_2=1,\ldots,i_{d'}=1,i_{d'+1}=1}^{N,N,\ldots,N,N} \delta_{i_1,i_2,\ldots,i_{d'},i_{d'+1}} = 0, \tag{11.22}$$

which is the mass conservation as defined in equation 11.12 extended to multi-dimensional tensors.

In conclusion, we have shown that the mass conservation as defined in equation 11.12 is ensured for our implementation of the LGT 5/3 and CDF 9/7 wavelets. The key to this result is the exact preservation of the first and last value of the wavelet transform. Hence, more generally, we can state that any border-preserving wavelet transform that ensures mass conservation as defined in equation 11.11 will also ensure mass conservation as defined in equation 11.12. The lifted Haar wavelets that we used do not ensure border preservation, which is why we observe experimentally that the ponderation given to the border values does have an impact on the mass conservation (see tables 11.9 and 11.10).

11.4 Using machine learning for symbolic regression

In this section, we delve into a collaborative work that has been submitted at the CEMRACS 2023 edition. The team members that contributed to this work are Camilla Fiorini, Clément Flint, Louis Fostier, Emmanuel Franck, Reyhaneh Hashemi, Victor Michel-Dansac, and Wassim Tenachi. In this work, we explored the use of machine learning techniques for symbolic regression, focusing on the Sparse Identification of Nonlinear Dynamical Systems (SINDy) framework. The collective goal was to correctly identify governing equations from data, with a particular emphasis on complex systems that are challenging to describe analytically. In that regard, the SINDy framework appeared as a promising approach, leveraging sparse regression to identify symbolic expressions from data.

My personal interest in this work is slightly different from the main focus of the project. As a computer scientist, I immediately identified the potential of machine learning techniques for floating-point optimizations. In high-performance applications, a frequent bottleneck is the computing power, usually limited by the floating-point operations. Some routines are known to be relatively slow, such as the exponential function, the logarithm, or the square root. More broadly, the results of some functions are difficult to compute due to the lack of simple mathematical expressions. We can think of the Bessel functions, the hypergeometric functions, or the elliptic integrals. The natural approach to implement these functions is to rely on numerical methods, such as the Taylor series, the Chebyshev polynomials, or Newton-Raphson iterations. To optimize these computations, classical approaches include the use of look-up tables or polynomial approximations.

This is where machine learning can bring a new perspective. By training a model on a set of inputs and outputs, we can approximate the function with a neural network. We have realized that some complex functions can be automatically approximated with simple expressions thanks to gradient-based optimization. While this work has not been extended to floating-point optimizations yet, it is a promising direction that I am eager to explore in the future.

This section is structured as follows. First, we provide an overview of symbolic regression and the SINDy framework. Then, we introduce the Nested SINDy approach, which aims to increase the expressivity of the SINDy framework through a nested structure. We present the results of our study, demonstrating the ability of Nested SINDy to accurately find symbolic expressions for simple systems and sparse analytical representations for more complex systems. Finally, we discuss the challenges encountered during the optimization process and suggest future research directions to enhance the methodology.

11.4.1 Symbolic regression

Symbolic regression (SR) consists in the inference of a free-form symbolic analytical function $f : \mathbb{R}^{n_1} \longrightarrow \mathbb{R}^{n_2}$ that fits $\mathbf{y} = f(\mathbf{x})$ given data $(\mathbf{x}, \mathbf{y})$. It is distinct from regular numerical optimization procedures in that it consists in a search in the space of functional forms themselves by optimizing the arrangement of mathematical symbols (e.g., $+$, $-$, $\times$, $/$, sin, cos, exp, log, ...).

The rationale for employing SR can be broadly categorized into the following three core objectives.

1. SR can be used to produce models in the form of compact analytical expressions that are interpretable and intelligible. This objective is particularly vital in natural sciences, such as physics [295], where the capacity to explain phenomena is equally valuable as predictive prowess. This is typically probed by assessing the capability of a system to recover the exact symbolic functional form from its associated data. However, one should note that many SR approaches excelling in this metric are often bested in fit accuracy when exact symbolic recovery is unsuccessful [141]. In other contexts where the compactness and inherent intelligibility of expressions may not be as critical, significantly longer but more robustly accurate expressions ($> 10^3$ mathematical symbols) are desirable as SR still offers key advantages in such scenarios.

2. SR demonstrates the advantage of producing models that frequently exhibit superior generalization properties when compared to neural networks [225, 127, 128, 291, 277].

3. Another noteworthy advantage is the ability to create models that demand significantly fewer computational resources than extensive numerical models like neural networks. This efficiency becomes especially relevant in *multi-query* scenarios such as control loops [128, 277], optimization or uncertainty quantification, where models must be executed frequently, and thus computational efficiency is crucial.

SR has traditionally been approached through genetic programming, where a population of candidate mathematical expressions undergoes iterative refinement using operations inspired by natural evolution, such as natural selection, crossover, and mutation. This approach includes well-known tools like Eureqa software [237, 238], as well as more recent developments [285, 286, 253, 135]. Additionally, SR has been explored using a diverse array of probabilistic methods [27, 55, 125, 172, 270]. For recent SR reviews, refer to [141, 174, 16].

The rise of neural networks and auto-differentiation[1] has spurred significant efforts to incorporate these techniques into SR, challenging the dominance of Eureqa-like approaches [141, 180, 177]. Numerous methods for integrating neural networks into SR have been developed, ranging from advanced problem simplification schemes [276, 275] to end-to-end supervised symbolic regression approaches in which neural networks are trained in a supervised manner to map datasets to their corresponding symbolic functions [127, 39, 116, 40, 284, 144, 81, 183, 160, 161, 68]. Unsupervised approaches also exist, where recurrent neural networks are trained through trial-and-error using reinforcement learning to generate analytical expressions that fit a given dataset [207, 147, 146, 159, 307, 94, 112, 268, 186]. Furthermore, it should be noted that it has been a major focus of the SR community to facilitate the incorporation of prior knowledge to constrain the search for functional forms by leveraging domain-specific knowledge [262, 263, 264, 26, 208, 107, 132, 54, 184] and that SINDy-like frameworks as the one proposed here can accommodate such prior knowledge, as demonstrated in works like [211].

Supervised approaches offer rapid inference but lack a self-correction mechanism. If the generated expression is suboptimal, there are little means of correction. In contrast, unsupervised approaches enable iterative correction based on fit quality. However, they often rely on reinforcement learning frameworks to approximate gradients because direct optimization using auto-differentiation is infeasible due to the discrete nature of the problem, which involves discrete symbolic choices.

However, other unsupervised methods include neuro-symbolic approaches, wherein mathematical symbols are integrated into neural network frameworks. The goal being to sparsely fit the neural network to enable interpretability, generalization or even recover a compact mathematical expression. Prominent examples include SINDy [60], which is central to this study, and others such as [179, 244, 225, 282, 133, 203, 202].

SINDy-like approaches are the only type of unsupervised techniques capable of directly utilizing gradients from data to iteratively refine function expressions as they effectively render the discrete symbolic optimization problem continuous. Moreover, SINDy-like frameworks possess the advantage of being well-suited for exact symbolic recovery by enabling the creation of concise, intelligible analytical expressions through the promotion of sparse symbolic representations while yielding highly accurate and general expressions when exact symbolic recovery is unsuccessful or impossible. However, a limitation of the current SINDy framework is its inability to handle nested symbolic functions, which often results in suboptimal performances, especially in more complex problems as evidenced by comparative benchmarks (see for instance [262]). This is the primary motivation for our study, where we introduce a Nested SINDy approach.

11.4.2 The SINDy paradigm

In this section, we present the traditional SINDy approach, as introduced in [60].

Principle of the SINDy method

The Sparse Identification of Nonlinear Dynamical Systems (SINDy) approach extends previous work in SR, introducing innovations in sparse regression. The SINDy approach seeks to deduce the governing equations of a nonlinear dynamical system directly from observational data, doing so in a concise and sparse way. It is based on the essential assumption that these governing equations can be succinctly expressed by only a few significant terms, resulting in a sparse representation within the space of potential functions [60].

More specifically, the SINDy approach involves approximating a target function through a linear combination of (potentially nonlinear) basis functions, contained in a so-called library or dictionary $\mathcal{F}$. For instance, $\mathcal{F}$ might include constant, polynomial, or trigonometric functions:

$$\mathcal{F} = \begin{Bmatrix} x \mapsto 1, & x \mapsto x, & x \mapsto x^2, & x \mapsto x^4, & \dots \\ x \mapsto \sin(x), & x \mapsto \cos(x), & x \mapsto \sin(2x), & x \mapsto \cos(\pi x), & \dots \end{Bmatrix}. \tag{11.23}$$

[1] Leveraging the capabilities of deep learning libraries to meticulously track gradients associated with a set of parameters in relation to a numerical process, regardless of its intricacy.

In order to achieve expressiveness in the SINDy method, a large number of basis functions are selected. This extensive selection, however, poses a risk to interpretability, which is mitigated by imposing a sparsity constraint on the coefficients of the linear combination, as will be detailed in the subsequent section.

The SINDy method offers several key advantages. First, it utilizes underlying convex optimization algorithms, which ensures its applicability to large-scale problems [60]. Additionally, the method inherently leads to a sparse representation of the system. This sparsity generally results in a model that is more interpretable and generalizable compared to denser models. Unlike the Dynamic Mode Decomposition (DMD) approach, which requires prior assumptions about the structure of the system [223], SINDy automatically identifies relevant terms in the dynamical system using gradient descent, without such assumptions.

Despite these benefits, the method is not without limitations. The choice of the library $\mathcal{F}$ is crucial and requires a priori knowledge of the system. For instance, if the target function is a composition or a multiplication of simple functions, the SINDy approach will fail to identify the correct expression unless this specific composition/multiplication is in $\mathcal{F}$. Furthermore, the training process to determine which functions to retain in the dictionary is notably more sensitive to initial conditions than in other approaches.

The upcoming section aims to establish the mathematical framework and main notation associated with the SINDy method, which will be used to introduce the Nested SINDy approach.

Mathematical framework and notation

For the sake of simplicity, we present the method in the case where the target function is from $\mathbb{R}$ to $\mathbb{R}$, but the approach can be extended to functions from $\mathbb{R}^n$ to $\mathbb{R}^m$.

Given the data $(x_i, y_i)_{i=1,\dots,N}$, we aim to find a function f such that $f(x_i) \approx y_i$ (which is nothing but a regression problem) using the SINDy approach.

Let $\mathcal{F} = \{f_1, \dots, f_l\}$ be the aforementioned dictionary of basis functions. For instance, it could be the one given by Equation 11.23. We denote by $L(\mathcal{F}) = \text{Span}(\mathcal{F})$ the set of linear combinations of these basis functions, defined by

$$f \in L(\mathcal{F}) \iff \exists \theta \in \mathbb{R}^l \text{ such that } f = \sum_{i=1}^{l} \theta_i f_i. \tag{11.24}$$

The regression problem can then be formulated as the following least squares problem:

$$\min_{f \in L(\mathcal{F})} \|Y - f(X)\|_2^2,$$

where $X = (x_1, \dots, x_N)^T$ and $Y = (y_1, \dots, y_N)^T$. This problem can itself be reformulated in matrix form, using the definition of the vector space $L(\mathcal{F})$:

$$\min_{\theta \in \mathbb{R}^l} \|Y - \mathbb{F}(X)\theta\|_2^2 \tag{11.25}$$

where $\mathbb{F}(X) = (f_j(x_i))_{i,j} \in \mathcal{M}_{N,l}(\mathbb{R})$.

Numerous algorithms exist to solve this problem while promoting sparsity. Without being exhaustive, notable methods include the standard STLSQ (sequentially thresholded least squares) and the LARS (least-angle regression) methods. Another approach involves adding a regularization term on the coefficients of the linear combination to favor sparsity. The most popular is Lasso regularization, but others exist (SR3, SCAD, MCP, ...). In this work, we focus on the Lasso approach. Introducing a Lasso regularization term to promote sparsity, the optimization problem for the SINDy approach becomes, instead of Equation 11.25:

$$\min_{\theta \in \mathbb{R}^l} \|Y - \mathbb{F}(X)\theta\|_2^2 + \lambda \|\theta\|_1, \tag{11.26}$$

where $\lambda > 0$ is a hyperparameter, to be manually set when using the method. The values of λ will be reported when using the method in the following sections. Specialized optimization algorithms, such as ADMM (alternating direction method of multipliers), are effective in solving regression problems with regularization.

11.4.3 The Nested SINDy approach

As mentioned in Section 11.4.2, one of the main limitations of the SINDy approach is the choice of the nonlinear basis functions populating the dictionary $\mathcal{F}$. For instance, if the unknown function happens to be a composition or a multiplication of simple functions, we cannot find the correct expression unless this specific composition/multiplication is in $\mathcal{F}$. In this paper, we aim at relaxing this constraint by introducing a way of multiplying and composing simple functions, without having to manually add these compositions functions to the dictionary.

In the same spirit as the approach investigated in [179, 225], we will enlarge the set $L(\mathcal{F})$. We will proceed by analogy with a standard approach in machine learning, which involves considering models with multi-layer neural networks rather than a single broad layer. Here, instead of considering a single layer of nonlinear functions, we explore an augmented architecture, consisting of several such layers. We will refer to this approach as Nested SINDy.

The cost to bear is the increased complexity of the optimization landscape. Indeed, the optimization problem, used to be the linear least squares problem given by Equation 11.26. Now, it becomes a nonlinear problem, since the matrix $\mathbb{F}(X)$ is replaced with a composition of nonlinear functions. The new optimization problem (still with Lasso regularization) is formulated as follows:

$$\min_{\theta} \frac{1}{2}\|y - \mathcal{N}(x,\theta)\|_2^2 + \lambda\|\theta\|_1,$$

where $\mathcal{N}$ denotes our nested model, parameterized by θ. Consequently, the resolution of this nonlinear optimization problem may be significantly more challenging than the original convex optimization problem.

The method then primarily depends on the choice of the architecture $\mathcal{N}$. The goal is to introduce new layers that achieve favorable trade-offs between expressivity and optimization complexity. In this work, we propose two architectures, which are described in the following sections: the PR model (in Section 11.4.3) and the PRP model (in Section 11.4.3). For the sake of clarity, we will refer to the basic SINDy layer, given by a projection onto $L(\mathcal{F})$ (Equation 11.24), as the radial layer.

The PR Model

The PR (Polynomial-Radial) model augments the basic SINDy framework by introducing a polynomial layer that operates before the usual SINDy radial layer. This layer constructs a variety of monomials from the input variable x, represented as follows:

$$f_{\text{poly}}(x) = \sum_{i=0}^{d} \omega_i x^i, \tag{11.27}$$

where d represents the maximum allowed polynomial degree and ω_i are the weights. Note that ω_0 is the constant part of the layer, which corresponds to the bias in traditional neural networks. For inputs with multiple variables, the layer extends to a multivariate polynomial, facilitating complex combinations of the variables. For example, with two variables x and y, we obtain

$$f_{\text{poly}}(x,y) = \sum_{i=0}^{d}\sum_{j=0}^{d} \omega_{i,j} x^i y^j, \tag{11.28}$$

with $\omega_{0,0}$ acting as the constant term of the polynomial, effectively substituting the bias. Another choice is to limit the sum over $i+j$ to a maximum value d to reduce the number of terms in the polynomial, and thus obtain bivariate polynomials up to degree d.

As an example, for one input variable, if $\mathcal{F} = \{\sin, \cos\}$ and $d = 2$, then the PR model can learn all functions with expression

$$\lambda\cos(a_1 + b_1 x + c_1 x^2) + \mu\sin(a_2 + b_2 x + c_2 x^2).$$

This is way more expressive than standard SINDy, where functions such as $x \mapsto \cos(2x)$ or $x \mapsto \sin(1+x^2)$ would have to be manually added to the dictionary.

This PR layer can also be seen as a pure polynomial layer combined with a linear layer. We, hence, consider the PR model to have four layers: a polynomial layer, a linear layer, a radial layer, and a final linear layer. The last two layers are identical to the standard SINDy model, while the first two layers are new additions. Figure 11.4 illustrates this structure.

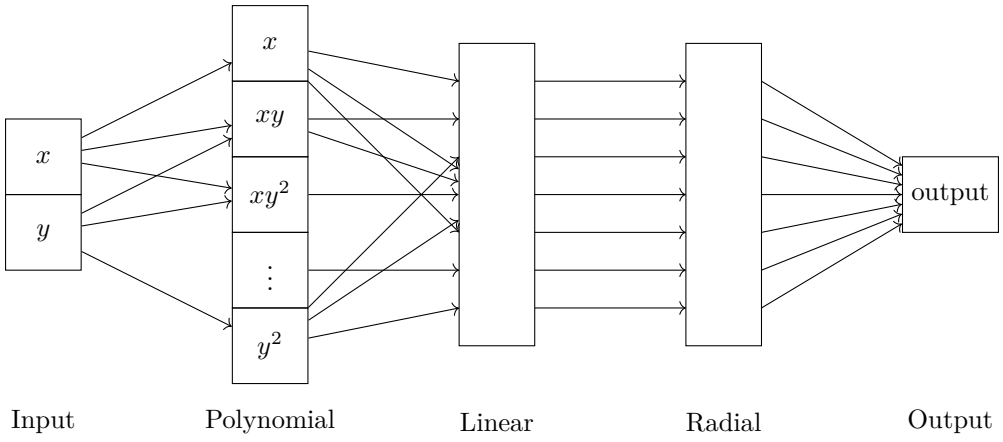


Figure 11.4: Structure of the PR model for $d = 2$ and 2 input variables.

The full expression of the model is:

$$f_{\theta,\mathrm{PR}}(x) = \sum_{j=1}^{l} c_j f_j \left(\sum_{i=0}^{d} \omega_{i,j} x^i \right) + B, \tag{11.29}$$

where θ includes all the trainable parameters of the model. The functions f_j are derived from the specified function set $\mathcal{F}$, while $\omega_{i,j}$ are the weights of the polynomial layer, and c_j and B are the weights and bias of the final linear layer, respectively. The parameters $\omega_{i,j}$ now depend on j since they are in the j-th radial layer. It is important to note that the radial and polynomial layers are not associated with any adjustable parameters.

The main advantage of the PR model is its enhanced expressivity, which facilitates the creation of linear combinations both before and after the radial layer. The polynomial layer allows the model to identify more complex functions and to integrate various inputs effectively in the case of multivariate data. Compared with the traditional SINDy approach, the model benefits from a reduced need for an extensive dictionary because it is capable of discovering linear combinations of the functions contained within the dictionary. Observations from our experiments suggest that the training process of the model is capable of converging to correct solutions even for non-trivial problems. This will be highlighted in Section 11.4.5.

The PRP Model

The PRP (Polynomial-Radial-Polynomial) model enhances the PR model by introducing an additional polynomial layer following the radial layer. This structure significantly improves the ability of the model to represent complex interactions within datasets. For example, the PRP model can express the function $f(x) = \arctan(x) \cdot \sin(x)$, (assuming arctangent and sine are in the dictionary), while the PR model cannot, as it is not a linear combination of the functions contained within the dictionary. In the PRP model, the outputs of the radial layer are first processed through a linear layer. This linear layer, typically fixed in size (set to 2 in our experiments), serves as an intermediate stage, transforming the outputs of the radial layer into a new set of variables. These variables are then fed into a subsequent polynomial layer, which allows for the formation of various monomial combinations of the outputs of the linear layers, such as squaring or multiplying them.

The mathematical expression of the PRP model is given as:

$$f_{\theta,\mathrm{PRP}}(x) = \sum_{1 \le |i| \le d} \omega^{\mathrm{PR}}_{i_1,i_2,\ldots,i_l} f_{\theta_1,\mathrm{PR}}(x)^{i_1} f_{\theta_2,\mathrm{PR}}(x)^{i_2} \ldots f_{\theta_l,\mathrm{PR}}(x)^{i_l} + B', \tag{11.30}$$

where $|i| = i_1 + i_2 + \ldots + i_l$ is the length of the multi-index $(i_1, \ldots, i_l)$, θ includes all the trainable parameters of the model, $\omega^{\text{PR}}_{i_1,i_2,\ldots,i_l}$ are the weights of the final linear layer, B' is the bias of the final linear layer, l is the size of the output chosen for the intermediate linear layer (set to 2 in our experiments), and $f_{\theta_1,\text{PR}}(x)$, $f_{\theta_2,\text{PR}}(x)$, $\ldots$, $f_{\theta_l,\text{PR}}(x)$ correspond to the output of the PR model given in Equation 11.29. The coefficients $\theta_1, \theta_2, \ldots, \theta_l$ correspond to the parameters of the PR models, which are the same as θ, except for the weights of the final linear layer (c_j and B in Equation 11.29). Figure 11.5 shows a graphical representation of this model.

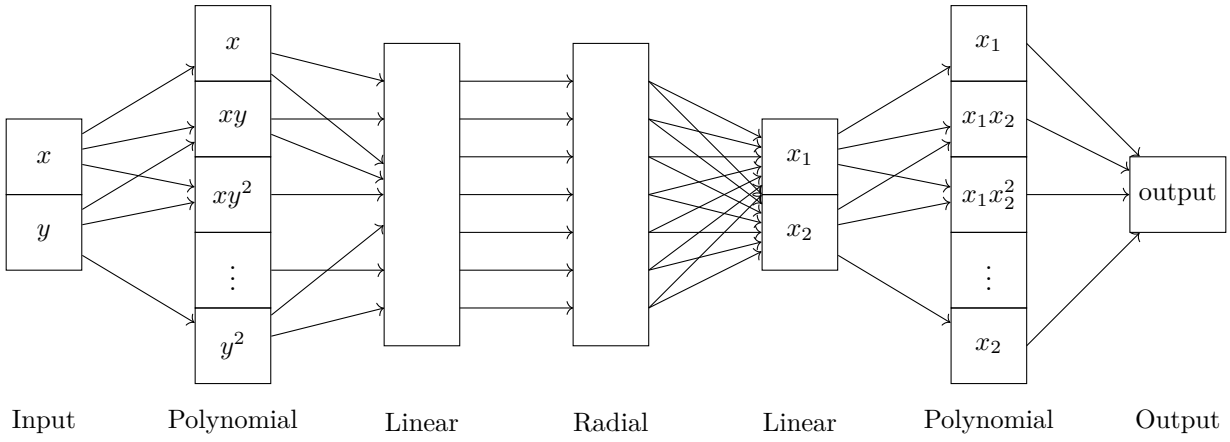


Figure 11.5: Structure of the PRP model for $l = 2$, $d = 2$, and 2 input dimensions.

The addition of a second polynomial layer in the PRP model markedly increases the expressivity of the model. It enables the model to capture more intricate relationships in the data, particularly beneficial for complex datasets where simpler models may fall short. Therefore, the PRP model is especially good at handling datasets with intricate variable interactions.

In summary, the PRP model, with its dual polynomial layers, offers a sophisticated extension of the SINDy approach. It provides a powerful framework for modeling complex systems, capable of capturing higher-order interactions and nonlinear relationships inherent in the data.

Downsides of the Nested SINDy approach

The Nested SINDy approach complexifies the optimization landscape, which destabilizes the training process. We can reasonably assume that the addition of linear layers creates local minima because of their composition with the nonlinear layers. Moreover, it adds a substantial amount of trainable parameters. This is because the number of parameters in the linear layers is quadratic, whereas those in the SINDy approach grow linearly with the number of functions in the dictionary. This deviates from the original SINDy approach, which is oriented towards function discovery, and assumes that the dictionary can become arbitrarily large. In the Nested SINDy approach, the number of functions in the dictionary should remain relatively small to avoid an explosion in the learning time. The next section addresses some training-related issues.

11.4.4 Training the nested SINDy model

We experimented and combined various strategies to best overcome challenges associated with nonlinear optimization. These strategies are given below, where we mention how we actually used them for training. Values of the hyperparameters introduced in this section will be given in Section 11.4.5.

Adding a regularization term to enforce sparsity

In our framework, the Lasso regularization term is added to the loss function to enforce sparsity in the model. We have tested different strategies to adapt the Lasso coefficient during training:

- A constant Lasso parameter throughout the training, which is the standard approach;

- A varying Lasso coefficient, initially set to zero for the early epochs, and then taken oscillating around some constant value, depending on the epoch. The intuition of this idea is that "shaking" the learning landscape helps to go out of local minima, in a direction that is still relevant to one of the two objectives (sparsity versus MSE);
- We also tried selecting neuron-dependent Lasso coefficients to promote specific functions or layers over others in the radial layer.

However, the approach that gave the most consistent results across the tested cases was to change the weight of the Lasso coefficient throughout the learning process. In the experiments, the Lasso coefficient is given by

$$\lambda(\text{epoch}) = \lambda_0 \cdot (1 + 0.4 \cdot \sin(\text{epoch}/10)),$$

where λ_0 is the initial Lasso coefficient, and where the sine function is used to oscillate the coefficient between $0.6\lambda_0$ and $1.4\lambda_0$.

Pruning to enforce sparsity

To enhance model sparsity, a complementary approach to Lasso regularization is to prune the neural network during training, reducing the number of parameters. In order to prune the neural network, we remove a parameter θ_i from the set of parameters $(\theta_i)_i$ if the following two conditions are met:

- The mean squared error is below a predefined threshold value $\text{MSE}_{\text{prune}}$, and
- $|\theta|$ is below a threshold value $\varepsilon_{\text{prune}}$ for a given number of epochs n_{prune}.

Choosing the optimization algorithm

We tested training our network with fairly standard optimization algorithms implemented in PyTorch but not specifically tailored to our problem: Adam, SGD, LBFGS. For instance, LBFGS [310] is uncommon in neural network training, as it was designed for optimizing constants in equations. However, it works well in our case, which is understandable given that our approach is close to classical regression problems. Moreover, it would be interesting to implement a more specific optimization algorithm that takes into account the form of our objective function (mean squared error plus regularization term), such as an ADMM (Alternating Direction Method of Multipliers) algorithm coupled with a standard PyTorch optimizer. Another promising avenue for SINDy-like approaches is the *basin-hopping* algorithm which combines LBFGS with global search techniques in order to avoid local minima as proposed in [244].

Adding noise to the gradient of the loss function

During a training step, we can add random noise to the gradient just before updating weights. This can be done at each training step, or only when the loss function does not vary sufficiently, e.g. when stuck in a local minimum.

The noise amplitude has to be well-tuned. It depends on the learning rate lr of the optimizer, the values of the parameters θ, and on the actual mean squared error L_{MSE}:

$$\nabla_\theta L \leftarrow \nabla_\theta L + \varepsilon(lr, L_{\text{MSE}}, \theta).$$

By making learning process less deterministic, we hope to more easily escape local minima. In the same spirit, we could also directly add noise to the parameters when the loss function is stuck in a local minimum.

Initializing the network parameters

The training process is very sensitive to the weight initialization, given the presence of multiple local minima in the objective function. For the moment, we have used a random normal initialization for the initial parameters of the network. To allow reproducibility, we rely on a fixed seed for the random number generator.

For the tested cases, this approach was sufficient, as attempting a limited number of training runs with different initializations was enough to find a good solution. However, for more complex problems, it would be interesting to implement a more sophisticated initialization strategy, for instance, by using reinforcement learning to discover important parameters and guide the initialization process accordingly.

11.4.5 Application to function discovery

In this work, we only consider function discovery. However, it is important to note that the nested SINDy approach can be extended to several other problems. In the following sections, we present four test cases, of increasing complexity.

Case 1: trigonometric function involving composition using the PR block

In Section 11.4.2, we recalled that the standard SINDy method encounters difficulties when the target is a function defined as the composition of several simpler functions, unless that specific composite function is included in the dictionary. To demonstrate the capability of the proposed PR nested SINDy method in such cases, the function $f(x) = \cos(x^2)$ is considered over the interval $[0, 3]$. With a dataset comprising 10^4 data points and utilizing a single input dimension, the PR-nested SINDy model attempts to replicate this function.

The architecture of the model is the PR model described in Section 11.4.3, with a maximum polynomial degree of 2 (in this case: x^2). The used dictionary of functions is:

$$\mathcal{F} = \left\{ x \mapsto x, x \mapsto x^2, \arctan, \sin, \cos, \exp, x \mapsto \log(|x| + 10^{-5}), x \mapsto \frac{1}{1+x^2} \right\}.$$

We use a Lasso regularization coefficient of 0.1, a batch size of 1 000, and a pruning threshold of 10^{-3}. The computations are performed on a CPU and all launched training eventually converged to an equivalent expression of the target function $cos(x^2)$. The training we use as a reference is shown in Figure 11.6 and converged to the expression $-0.985\cos(x^2 + 3.13)$. This expression is close to $-1\cos(x^2 + \pi)$, which equivalent to $cos(x^2)$. It is worth noting that the final expression can be further trained with a classical regression method to obtain a more accurate result.

In this example, the PR model successfully learns a composition of a function that is in the dictionary (cosine) with a polynomial function (that appears in the polynomial layer). Since the exact formula has been found, the generalization of the model is excellent. However, the learned function closely matches the structure of the model, since it conveniently includes the composition of a polynomial and a cosine function. In the next section, we attempt to learn the same function, but with an additional polynomial layer (PRP model). As the added polynomial layer is unnecessary, it will show that our example does not heavily rely on the structure of the model.

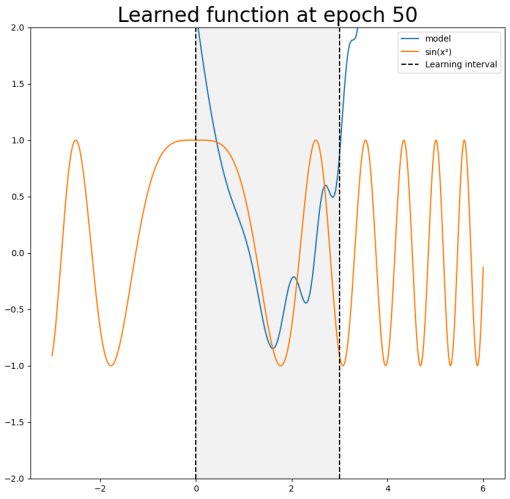
Case 2: trigonometric function involving composition using the PRP block

To further illustrate the capabilities of the introduced PRP-nested SINDy method, the same function $f(x) = \cos(x^2)$, considered over the interval $[0, 3]$, is examined. This time, we employ a dataset comprising 1 000 data points and a single input dimension, the PRP-nested SINDy model is tasked with replicating this function. The reason for using a smaller dataset is that that the added polynomial layer in the PRP model significantly slows down the training process, due to the increased number of parameters.

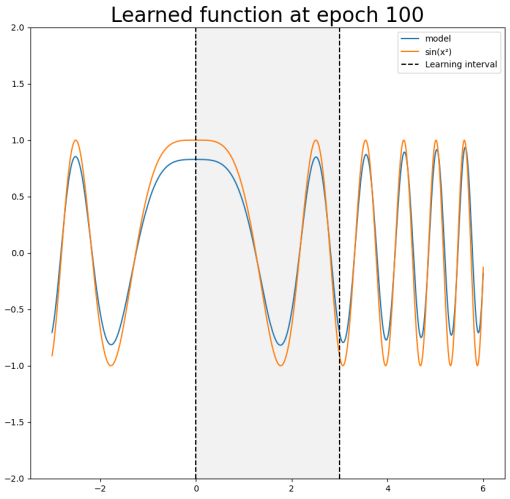
The structure of the model is the PRP model described in Section 11.4.3, with a maximum polynomial degree of 2 and $l = 2$, i.e., two intermediate variables after the radial layer. The used dictionary is the following:

$$\mathcal{F} = \left\{ x \mapsto x, x \mapsto x^2, \arctan, \sin, \cos, \exp, x \mapsto \sqrt{x}, x \mapsto e^{-x^2}, x \mapsto \log(1 + e^x) \right\}.$$

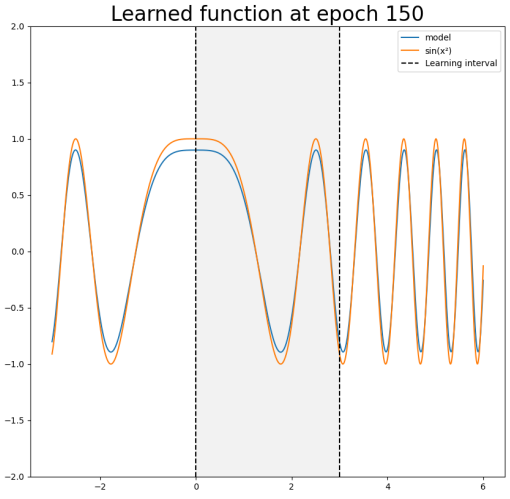
For this experiment, we use a Lasso regularization coefficient of 10^{-3}, a batch size of 1 000, and a pruning threshold of 0.05. This time, few training runs converged to the exact expression of the target function. Approximately 1 out of 10 runs converged to an expression equivalent to the target function, while the others converged to sparse, but incorrect, expressions. As a showcase, let us focus on a run that converged to the expression $\sin(x^2 + 1.57)$. This expression is close to $\sin(x^2 + \pi/2)$, which is equivalent to $\cos(x^2)$. Figure 11.7 shows the comparison between the learned function and the target function.



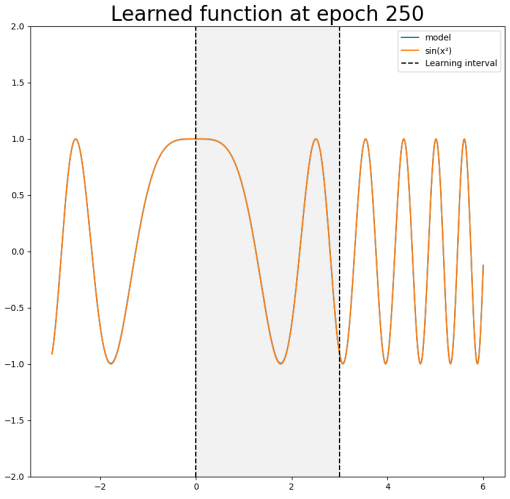
(a) Epoch 50: early model predictions versus target function, illustrating the initial learning phase.



(b) Epoch 100: model predictions show improved alignment with the target function as learning progresses.



(c) Epoch 150: further refined predictions, with the model beginning to capture periodicity of the target function.



(d) Epoch 250: near-convergent model predictions closely matching the target function.

Figure 11.6: Evolution of the learned function over successive training epochs. Each subfigure represents the predictions of the model (in blue) against the target function (in orange) at epochs 50, 100, 150, and 250, showcasing the progressive learning of the model and convergence towards the target function. We observe that the first epochs learn on the interval $[0, 3]$, while subsequent epochs are able to generalize.

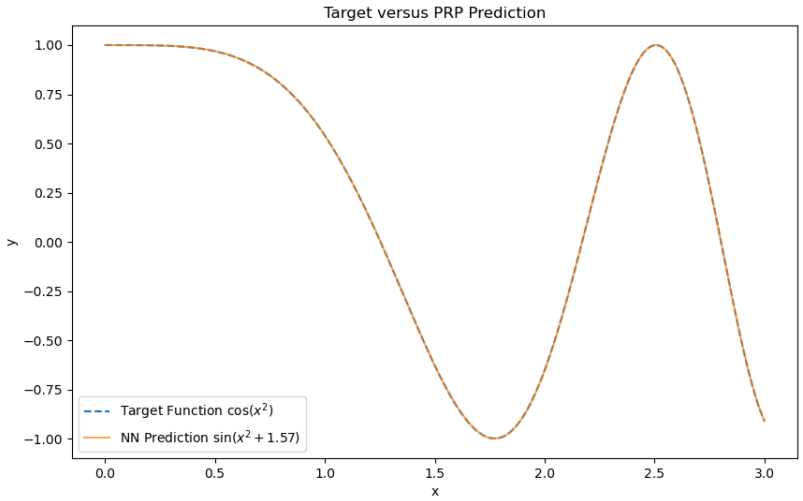


Figure 11.7: Comparison of the results from the PRP model ($\sin(x^2+1.57)$) with the target function ($\cos(x^2)$) over the interval [0, 3].

This test case demonstrates that $cos(x^2)$ can be learned both with the PR and PRP models. It is not clear why the PRP model is less likely to converge to the exact expression. Our interpretation is that the added polynomial layer in the PRP slows down the training process (both in terms of computation time and numerical convergence), which increases the average required time to converge to the correct expression. It could also be due to the fact that this additional layer make the problem "less convex" and, hence, less likely to converge to the exact expression.

In the next test case, we consider a more complex 2D function, which involves the multiplication of two trigonometric functions.

Case 3: Trigonometric Function Multiplication using the PRP Block

The following test case involves learning the two-dimensional function $(x, y) \mapsto 2\sin(x)\cos(y)$ over the space domain $[-2, 2]^2$ using the PRP model. This example is particularly interesting as it highlights ability of our approach to learn complex functions involving the multiplication of simple functions, whose product is not included in the dictionary. It demonstrates an increase of expressivity compared to the SINDy approach, which would not be capable of learning this function unless the product of sine and cosine functions was explicitly included in the dictionary.

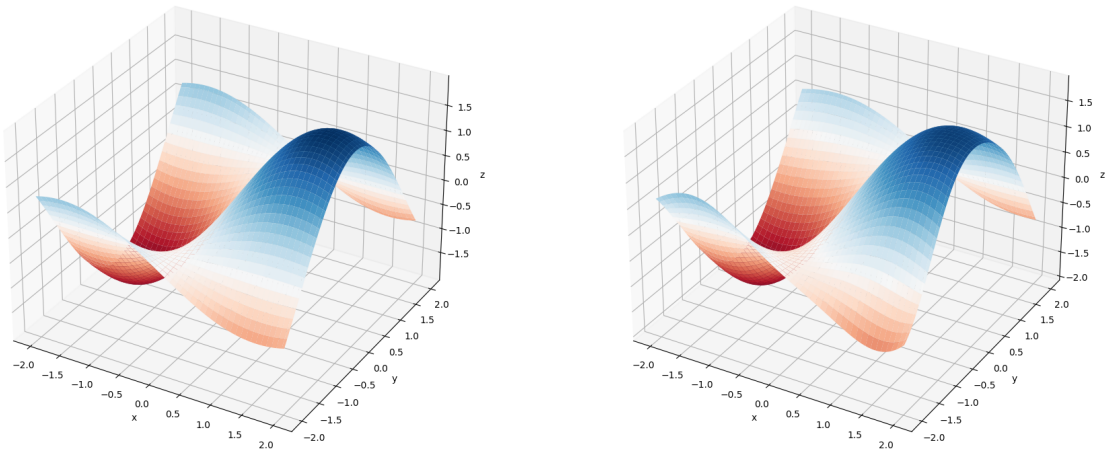
The used model is the same as the PRP model used in the previous test case, apart from the fact that the input dimension is now two and the dictionary of functions is extended to include the following:

$$\mathcal{F} = \left\{ \begin{array}{ccccc} x \mapsto \sqrt{|x| + 10^{-5}}, & x \mapsto x, & \sin, & \cos, & \tanh, \\ \exp, & x \mapsto \dfrac{1}{1+x^2}, & x \mapsto \log(|x| + 10^{-5}), & x \mapsto \exp\left(\dfrac{1}{1+x^2}\right), & x \mapsto \log(1+e^x) \end{array} \right\}.$$

This time, we were unable to find the exact expression of the target function. The sparsest expression found is given by:

$$\begin{array}{r} -0.77\left(1 - 0.612\sin\left(0.032x^2 - 0.162xy + 0.998x + 0.035y^2 - y - 0.14\right)\right)^2 + \\ 2.01\left(\cos\left(0.039xy - 0.499x - 0.497y + 0.809\right)\right)^2 \end{array}, \tag{11.31}$$

corresponding to a sparsity of 43.75% (14 nonzero coefficients out of the original 32 parameters). A comparison of this function with the target function is illustrated in Figure 11.8. The mean squared error (MSE) between the target function and its approximation is 5.69×10^{-2}.



(a) True model ($2\sin(x)\cos(y)$)

(b) Predicted model

Figure 11.8: Comparison of the true and predicted models: a visual representation of the target function $2\sin(x)\cos(y)$ and its learned approximation by the PRP model over the specified domain.

It is worth underlining that the small coefficients in the formula are not negligible. For instance, removing the term $0.039xy$ from the formula increases the MSE to 2.11×10^{-1}.

This example demonstrates the ability of our approach to find sparse expressions for data where the exact function cannot be found. This is promising for real-world applications where the exact function is unknown or too complex to be found. In the next test case, we aim to provide a concrete example where finding the exact function is impossible and assess the relevance of our approach compared to existing methods.

Case 4: Perimeter of an ellipse

Calculating the perimeter of an ellipse is a well-studied topic for which the solution cannot be expressed in terms of elementary functions. Let us first define an ellipse as the set of points (x, y) such that:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1, \tag{11.32}$$

which can be described with the parametric equations:

$$\begin{cases} x = a\cos(\theta), \\ y = b\sin(\theta). \end{cases} \tag{11.33}$$

The perimeter of an ellipse can then be expressed as:

$$P(a, b) = 4\int_0^{\frac{\pi}{2}} \sqrt{a^2\cos^2(\theta) + b^2\sin^2(\theta)}\, d\theta, \tag{11.34}$$

because the differential arc length of our parametric equation is $ds = -\sqrt{((-a\sin(\theta))^2 + (b\cos(\theta))^2)}d\theta$ and the four quadrants have the same length. Several approximations are known to approach the perimeter of an ellipse, such as the one proposed by Ramanujan:

$$P(a, b) \approx \pi(3(a + b) - \sqrt{(3a + b)(a + 3b)}). \tag{11.35}$$

Since any rescaled ellipse remains an ellipse, we assume without loss of generality that $b = 1$ for the rest of this experiment. To assess the relevance of our Nested SINDy technique, we will compare its performance

against the two reference solutions: Ramanujan's approximation, and a linear interpolation between the two endpoints of the interval $[1, 30]$. Indeed, using a linear interpolation makes sense for large values of a, since P is equivalent to $4a$ when $a \to \infty$, as can be seen by inspecting Equation 11.34.

To find formulas that approximate the perimeter of an ellipse, we train two PRP models. The first one is trained on the interval $[1, 5]$ and the second one on the interval $[1, 25]$. We run 50 learning sessions for each model and keep the model with the shortest associated formula. In its best run, the first model converged to a basic quadratic polynomial:

$$P_{\text{quadratic}}(a) = 0.061(a + 0.544)^2 + 3.28a + 2.72. \tag{11.36}$$

However, other relatively sparse solutions were found, such as a model with 9 nonzero parameters:

$$P_1(a) = 1.65a + 0.553(0.485a + 0.135\log(0.817|a^2|) + 1)^2 + 0.459\log(0.817|a^2|) + 3.4, \tag{11.37}$$

or one with 15 nonzero parameters:

$$\begin{aligned} P_2(a) = {} & 0.535a + 0.966(0.394a + 0.721\arctan(0.278a^2 + 0.393) + 1 \\ & + 0.111\exp(-0.063a^4))^2 + 0.978\arctan(0.278a^2 + 0.393) + 1.36 + 0.15\exp(-0.063a^4), \end{aligned} \tag{11.38}$$

thus demonstrating the ability of the model to find a variety of approximations. The MSE obtained at the end of the training are 2.26×10^{-3}, 2.69×10^{-3}, and 3.30×10^{-3} for the quadratic, P_1, and P_2 models, respectively.

The second model converged to a more complex expression which is too long to be displayed here. In this second model, 25 out of the 64 parameters are nonzero, corresponding to approximately 40% sparsity. The MSE obtained at the end of the training is 1.97. This metric, as the previous ones, is biased because the training is stopped at an arbitrary step and there is no final tuning step. We, hence, perform a final tuning step, by writing the final expression of the model and performing a gradient descent to only minimize the MSE. The later mentioned results include this final tuning step and show that the effective MSE can be significantly reduced.

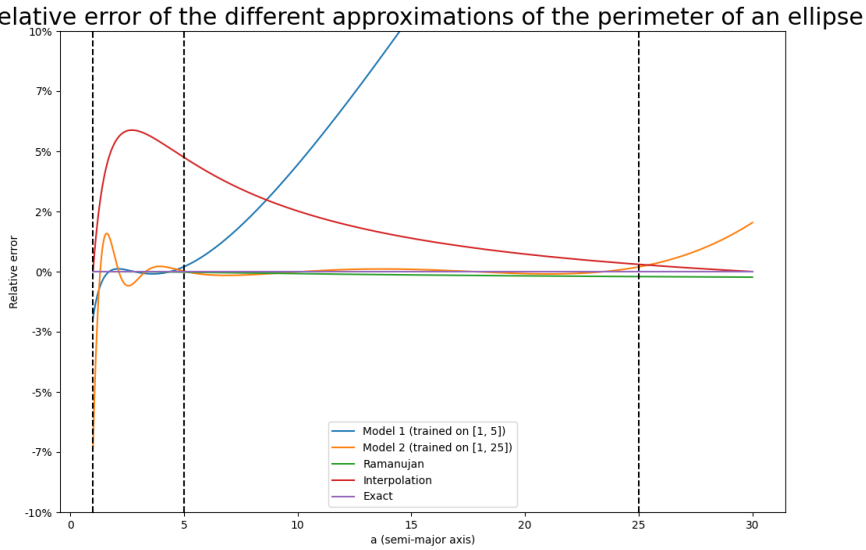


Figure 11.9: Relative error of the different approximations on the intervals $[1, 5]$, $[1, 25]$ and $[1, 30]$. The closest approximations are denoted in bold.

We display the relative error of each approximation on different intervals in Figure 11.9. Table 11.11 reports the MSE of each approximation on the intervals $[1, 5]$, $[1, 25]$ and $[1, 30]$. *Model 1* and *model 2*

Table 11.11: Mean squared error of the different approximations on the intervals $[1, 5]$, $[1, 25]$ and $[1, 30]$

	$[1, 5]$	$[1, 25]$	$[1, 30]$
Model 1	7.24×10^{-4}	1.05×10^{2}	2.61×10^{2}
Model 2	7.43×10^{-3}	$\mathbf{3.66 \times 10^{-3}}$	2.73×10^{-1}
Ramanujan	$\mathbf{2.78 \times 10^{-6}}$	9.84×10^{-3}	$\mathbf{1.82 \times 10^{-2}}$
Interpolation	4.70×10^{-2}	4.75×10^{-1}	5.46×10^{-1}

correspond to our two models (with *model 1* corresponding to Equation 11.36), *Ramanujan* corresponds to the approximation proposed by Ramanujan and *Interpolation* corresponds to the linear interpolation between the two endpoints of the interval $[1, 30]$. The two models perform well on the interval they were trained on. The first model performs better than the linear interpolation on $[1, 5]$, but worse than the Ramanujan approximation. The second model outperforms both the linear interpolation and the Ramanujan approximation on $[1, 25]$, but starts diverging on $[1, 30]$.

Overall, this experiment provides insight into the ability of the Nested SINDy approach to discover an approximation on a classical problem without knowledge other than the data points. It demonstrates that this method can be used to discover an approximation of a complex function with little effort. The main downside of this approach is the lack of guarantee regarding the convergence towards a sparse solution. The initial choice of coefficients of the model appears to have a significant impact on the final solution.

11.4.6 Conclusion

In this study, we explored the capabilities and limitations of the Nested SINDy approach in discovering symbolic representations of dynamical systems from data. Our investigation covered a range of test cases, from simple trigonometric functions to more complex functions.

The Nested SINDy approach extends the original SINDy methodology by incorporating nested structures and neural network architectures, allowing for the identification of complex symbolic expressions that are not directly accessible to traditional methods. This capability is necessary in cases involving compositions and multiplications of functions, where the Nested SINDy method accurately identifies symbolic representations in simple cases, or finds sparse symbolic representations in more complex cases. Our results confirm that Nested SINDy is a promising tool for symbolic regression and dynamical system discovery, with the potential to uncover the underlying physics of complex systems from data.

Future work could focus on several areas to enhance the Nested SINDy framework. First, exploring alternative optimization algorithms specifically designed for the unique challenges of nested symbolic regression could improve the efficiency and reliability of the method. Additionally, incorporating mechanisms for automatic selection of the dictionary of basis functions based on preliminary data analysis might streamline the model development process and improve the adaptability of the model for specific dynamical systems. Furthermore, one could envision integrating a supervised learning component, as in [244], in which a model is pre-trained to map the relationship between datasets and SINDy-like sparse patterns encoding analytical expressions. This effectively enables the automatic formulation of high quality initial solutions that can then be refined on a case-by-case basis.

For the moment, this work is not yet mature enough to be integrated into the research axes of this thesis. However, it is evident that the approach used in this work has the potential to be a powerful tool for floating-point optimization. In particular, finding sparse representations of complex functions could lead to compact and efficient implementations of mathematical functions. We have several ideas to improve the Nested SINDy approach and orient it towards floating-point optimization.

Chapter 12

French Summary

Cette thèse aborde les défis de l'optimisation des calculs de simulation de fluides sur des processeurs graphiques (GPU) dans le cadre d'un environnement de calcul haute performance. L'accent est mis sur la gestion sioux de la mémoire, qui est souvent un facteur limitant pour les simulations à grande échelle. Dans le contexte actuel, la gestion de la mémoire est souvent mise de côté, en raison des capacités matérielles, qui sont de plus en plus importantes. On suppose souvent que si les capacité mémoire du système sont dépassées, il suffit d'ajouter du matériel pour résoudre le problème.

Cependant, des défis pratiques sont souvent associés à l'ajout de matériel, tels que la consommation d'énergie, le coût, la complexité de la configuration, et la maintenance. De plus, le fait de lancer de manière abrupte un gros calcul sur un cluster crée souvent un goulot d'étranglement, car ce calcul réquisitionne d'un seul coup toutes, ou une grande proportion des ressources disponibles. C'est paradoxal, car en général, les clusters sont sous-utilisés, avec généralement moins de la moitié des processeurs actifs tandis que le reste reste inactif. Ainsi, la recherche tend à s'orienter de plus en plus vers des approches qui ne se concentrent pas uniquement sur le temps de calcul, mais également vers une utilisation plus judicieuse et équilibrée des ressources de calcul.

Cette thèse s'inscrit dans ce courant de recherche, en proposant des approches novatrices pour l'optimisation des calculs de simulation de fluides sur des processeurs graphiques. Notre objectif est de trouver des solutions qui permettent de réduire l'empreinte mémoire des simulations de CFD, tout en maintenant des performances élevées. En réduisant l'empreinte mémoire, nous pouvons améliorer de manière concrète l'utilisation des ressources de calcul, en permettant par exemple de lancer plus de simulations en parallèle sur un cluster donné.

Ce résumé est structuré de la manière suivante. La section 12.1 présente les fondements scientifiques et l'état de l'art des simulations de CFD. Ensuite, la section 12.2 établit les défis et objectifs de la thèse au regard de l'état de l'art actuel. Dans la section 12.3, nous discutons des différentes approches que nous avons abordées pour optimiser les calculs de simulation de fluides. La section 12.4 présente une contribution majeure de cette thèse, à savoir l'ordonnancement hétérogène automatique sur StarPU. Enfin, la section 12.5 présente une autre contribution majeure de cette thèse, à savoir l'utilisation de compression à la volée pour réduire l'empreinte mémoire des simulations de CFD. Nous conclurons ce résumé dans les sections 12.6 et 12.7, où nous discuterons des perspectives futures et récapitulerons les contributions de cette thèse.

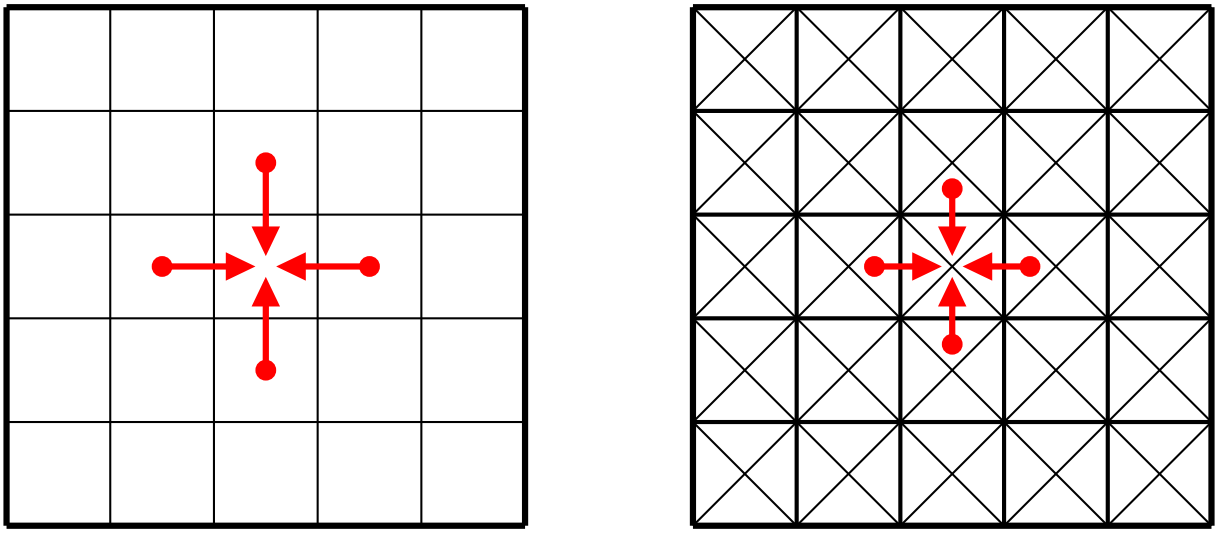
12.1 Fondements Scientifiques et État de l'Art

12.1.1 Équivalence entre la Simulation de Fluides et les Algorithmes de Stencil

Une façon très commune de simuler des écoulements de fluides est d'effectuer de la discrétisation numérique pour résoudre les équations de Navier-Stokes. Cette discrétisation peut se faire à plusieurs niveaux: discrétisation de l'espace, du temps, etc. Ce qui nous intéresse le plus dans notre approache, c'est la discrétisation de l'espace, qui est responsable de la plupart des besoins en mémoire. Il existe de nombreux schémas de discrétisation, tels que les schémas de volumes finis, les schémas de différences finies, les schémas de Galerkin, etc.

Ces schémas sont souvent associés à des maillages structurés ou non structurés, qui peuvent être de différentes natures: maillages cartésiens, maillages hexaédriques, maillages triangulaires, maillages tétraédriques, etc.

Dans notre aproche, nous nous concentrons sur les maillages cartésiens, qui sont pratiques en termes de programmation car ils correspondent généralement à des grilles régulières de valeurs flottantes. Les maillages cartésiens sont très utilisés en pratique par les numériciens, car ils sont simples à mettre en place et à manipuler. Une approche classique qui se base souvent sur des maillages cartésiens est la méthode de Lattice Boltzmann (LBM). Au cours de cette thèse, nous y ferons souvent référence, car c'est une méthode qui est très utilisée en pratique et qui est souvent associée à un coût mémoire élevé.



(a) Une valeur par cellule.

(b) 4 valeurs par cellule.

Figure 12.1: Exemples de types de stencils sur des maillages cartésiens. La figure 2.1a montre un stencil quiprend en compte les 4 voisins (au sens de Von Neumann) d'une cellule. La figure 2.1b montre un exemple avec les valeurs de 4 voisins et 4 valeurs par cellule. Les 4 valeurs par cellule peuvent correspondre à des valeurs cinétiques du fluide, par exemple.

Cependant, notre approche n'est pas vraiment dépendante du schéma numérique utilisé, mais plutôt de la structure en grille du maillage. En programmation, on regroupe généralement ce type d'algorithmes sous le terme de d'algorithmes de "stencil" (rarement traduit par "pochoir" en français). Dans ces algorithmes, chaque élément du maillage est mis à jour en fonction de ses voisins, en utilisant un "stencil" qui définit les opérations à effectuer. On peut voir ça comme l'application d'une fonction φ de type $f : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^m$ sur chaque élément du maillage, où n est la taille du stencil (nombre de voisins lus) et m est le nombre de valeurs dans une cellule du maillage (en général: les valeurs macroscopiques du fluide). Deux exemples de stencils sont donnés dans la figure 12.1. Tant que le maillage utilisé est une grille régulière, l'approche stencil peut être utilisée, ce qui en fait une approche très générale. Nous ferons donc systématiquement l'hypothèse que notre simulation est implémentable en utilisant un algorithme de stencil, et nous concentrerons sur la formulation "stencil".

12.1.2 Travaux Connexes

Il existe de nombreux travaux connexes qui abordent des problématiques similaires à celles que nous abordons dans cette thèse. Tout d'abord, certains travaux sont dédiés à l'optimisation des calculs de simulation de fluides sur des processeurs graphiques. Bien qu'ils ne soient pas directement utilisés dans cette thèse, ils ont été une source d'inspiration pour s'assurer que nos travaux sont en phase avec les dernières avancées dans le domaine.

Ensuite, il y a les travaux axés sur la gestion efficiente de la mémoire. Nous avons choisi de les regrouper en trois catégories:

- **Optimisation "légères":** Cette catégorie inclut les travaux portant sur des optimisations jugées "légères". Une partie de ces travaux se concent sur des optimisations algorithmiques, tandis que d'autres se basent sur des changements de précision flottante. Ces optimisations sont à la fois "légères" en temps de calcul, mais aussi en termes de gain de mémoire. Dans le cas de calculs à précision flottante multiple, le gain de mémoire est, en principe, compensé par une perte d'accuité numérique. Nous verrons au cours de cette thèse que notre approche offre généralement un meilleur compromis entre gain de mémoire et perte d'accuité numérique;

- **Raffinement de Maillage Adaptatif (AMR):** Bien que l'AMR s'éloigne de l'approche "stencil", il est essentiel d'y faire référence car c'est une méthode très utilisée en pratique. Dans cette approche, le maillage est raffiné localement en fonction de la complexité du problème, amenant à diviser l'espace sous forme d'arbre (quadtree en 2D, octree en 3D). Dans son essence, cette approche est une façon de mieux répartir les capacités de calcul et de mémoire, en les concentrant aux endroits où elles sont les plus nécessaires. Il existe différentes limitations à cette approche, notamment le coût de gestion de l'arbre, la difficulté de parallélisation, et la nécessité pour les numériciens de gérer les interfaces entre les différents niveaux de raffinement;

- **Compression (explicite) de Données:** Enfin, certains travaux s'intéressent à la compression explicite de données. Notre approche s'inscrit dans cette catégorie, mais avec une approche novatrice qui n'a pas été explorée dans la littérature. Notre méthode de compression est basée sur les ondelettes, un outil mathématique bien connu pour sa capacité à compresser des signaux. Les ondelettes sont connues pour offrir des taux de compression élevés, mais sont en général considérées comme trop coûteuses en termes de calcul pour être utilisées comme un moyen de compression "à la volée". Nous avons cependant montré qu'en utilisant de manière appropriée les processeurs graphiques modernes, il est possible d'obtenir des vitesses de compression satisfaisantes.

Au vu de la littérature actuelle, il est clair que notre approche est novatrice et qu'elle offre un compromis intéressant entre gain de mémoire et perte d'accuité numérique. Dans la prochaine section, allons définir les objectifs de cette thèse.

12.2 Objectifs de la Thèse

Cette thèse aborde les défis de l'optimisation des calculs de simulation de fluides sur des processeurs graphiques, en se concentrant sur la gestion de la mémoire. Une partie des contributions est dédiée à l'accélération des calculs de simulation de fluides sur processeur graphique. Bien que cet aspect soit secondaire par rapport à l'objectif principal de cette thèse, il est nécessaire de s'assurer que les calculs de simulation de fluides sont effectués de manière efficace. Pour cela, nous nous focalisons sur le calcul distribué de stencils, qui permet de faire des simulations de fluides à grande échelle.

Une façon moderne de distribuer les calculs est de faire appel à des bibliothèques de calcul distribué. Deux outils puissants existent et sont très utilisés par la communauté calcul haute performance: StarPU et PaRSEC. Ces deux moteurs sont basés sur la parallélisation à base de tâches. Nous avons effectué diverses contributions avec ces deux moteurs, dans l'objectif de fournir une base solide pour intégrer nos travaux dans des environnements de calcul haute performance.

Enfin, une partie importante de cette thèse est dédiée à la compression de données en tant que telle. Au cours de divers travaux, nous avons utilisé les ondelettes pour créer un paradigme de compression adapté aux simulations de fluides. L'objectif final est d'aboutir à un gain effectif de mémoire sans compromettre la précision des simulations. La balance se fait entre trois aspects: le taux de compression, la perte d'accuité numérique, et la vitesse de compression.

Dans la prochaine section, nous discuterons des différentes approches que nous avons abordées pour optimiser les calculs de simulation de fluides.

12.3 Optimisation des Calculs de Simulation de Fluides sur processeur graphique

Au cours de cette thèse nous avons abordé plusieurs aspects de l'optimisation des calculs de simulation de fluides sur processeur graphique Tout d'abord, nous nous sommes focalisés sur l'implémentation d'un schéma de magnétohydrodynamique (MHD) sur processeur graphiques. Pour cela, nous nous sommes basés sur les travaux de Baty *et al.* [29], qui se sont eux-même basés sur la méthode MHD-DC (MHD, Divergence-Cleaning). En observant une convergence d'ordre 2, nous pouvons avoir confiance en le fait que notre implémentation est correcte, et pouvons nous concentrer sur l'optimisation de la performance.

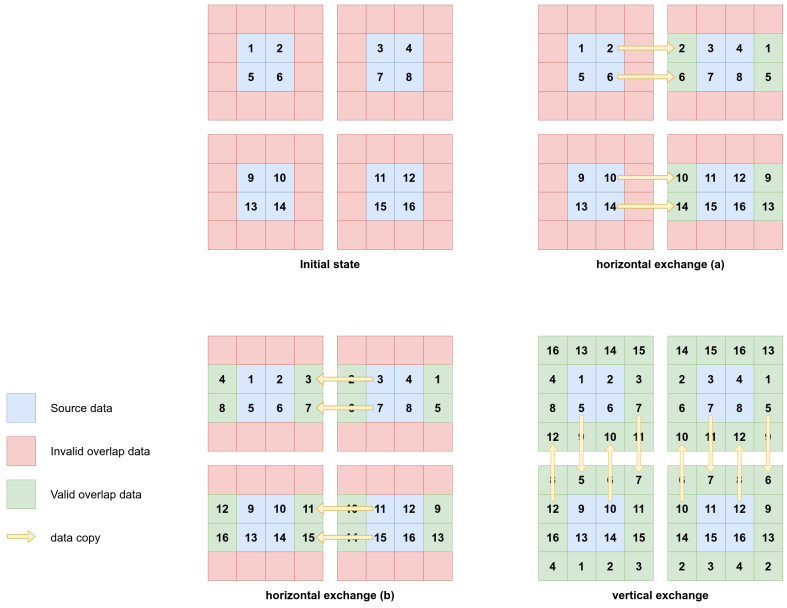


Figure 12.2: Ce schéma montre comment les "faces" peuvent être échangées entre les sous-grilles dans le cas 2D. La phase de copie horizontale est divisée en deux parties pour mieux afficher le processus. Pour des raisons de clarté, toutes les copies de données nécessaires (flèches jaunes) ne sont pas affichées. Dans un scénario réel, les données sont des vecteurs de valeurs flottantes, plutôt que des entiers, qui sont ici utilisés pour marquer les différentes cellules.

Dans le cadre de cette thèse, nous pensons que la partie la plus importante de l'optimisation de simulations de fluides est la gestion du passage à l'échelle. Ainsi, nous avons abordé la question de la distribution de calculs de stencils sur des processeurs graphiques. Il existe de nombreux moteurs de calcul distribué, certains spécialisés dans le calcul de stencils. Il n'est donc pas nécessairement pertinent d'en développer un nouveau en se basant sur les mêmes paradigmes. Nous avons donc étudié certains moteurs de calcul de stencil et remarqué que leur approche était généralement "rigide", c'est-à-dire qu'ils ne permettent pas de s'adapter à des changements de besoins au fil de l'exécution. La figure 12.2 nous montre l'approche classique de distribution de calculs de stencils avec 2×2 sous-grilles. L'idée est de diviser la grille en sous-grilles, et de distribuer les calculs de manière équilibrée entre les différents processeurs. Les sous-grilles incluent des "bords fantômes" qui permettent de répliquer les valeurs des cellules voisines et ainsi de garantir l'indépendance des calculs. Ce paradigme, si appliqué tel quel, peut mener à des problèmes de famine de tâches, où certains processeurs n'ont plus de travail à effectuer.

Il apparaît donc qu'utiliser des outils plus flexibles, tels que StarPU et PaRSEC, pourrait être une solution plus adaptée. Concrètement, ces outils pourraient par exemple permettre de détecter des phénomènes de famine de tâches, et de les résoudre en répartissant le travail de manière plus équilibrée. StarPU et PaRSEC s'inscrivent dans une approche de parallélisation à base de tâches. Cette approche est généralement appréciée des développeurs, car elle est relativement intuitive et généralement suffisante pour obtenir de

bonnes performances.

Dans ce cadre, nous avons exploré le potentiel du modèle de programmation PTG (Parameterized Task Graphs) offert par PaRSEC pour une implémentation efficace d'un schéma D2Q9. Nous avons étendu ce modèle pour mieux gérer les calculs de stencils complexes en introduisant des flux paramétriques qui simplifient l'écriture de la relation de dépendance entre les tâches. Cette innovation permet de réduire la complexité du code et d'améliorer l'expressivité du modèle de programmation, facilitant ainsi le développement et la maintenance du code. En implémentant le schéma numérique, nous avons démontré que les performances sont préservées, c'est-à-dire que l'utilisation de flux paramétriques est équivalent à une implémentation manuelle des dépendances entre les tâches. Enfin, nous avons proposé une discussion sur la façon optimal d'effectuer la synchronisation des sous-grilles. L'implémentation actuelle est limitée à une exécution sur un seul nœud, mais une exécution multi-nœuds ne devrait pas poser de problèmes majeurs.

Enfin, nous avons implémenté un moteur générique de calcul de stencil basé sur StarPU. L'idée est de fournir une base de code solide pour calculer des stencils dans différents contextes. Grâce au modèle de StarPU, il est possible de distribuer les calculs de manière efficace sur différents types de processeurs (dont graphiques). Nous avons validé notre implémentation (multi-GPU) en la comparant à une implémentation (mono-GPU) du schéma de magnétohydrodynamique par Baty *et al.* [29]. Ce moteur est conçu pour être facilement adaptable à différents types de stencils, et sera utilisé comme base pour des travaux futurs.

Bien que ce moteur pourrait être amélioré pour être plus compétitif, nous avons focalisé l'étude suivante sur améliorer l'ordonnancement des tâches dans StarPU. En effet, l'utilisation de StarPU abstrait de nombreux aspects de la programmation parallèle, dont l'ordonnancement des tâches, qui est pourtant un aspect critique pour obtenir de bonnes performances. Dans la prochaine section, nous discuterons de notre contribution majeure dans ce domaine.

12.4 Ordonnancement Hétérogène Automatique sur StarPU

Cette section détaille l'introduction d'une stratégie d'ordonnancement avancée pour StarPU, un système d'exécution orienté vers l'exécution optimisée de tâches sur des architectures hétérogènes. L'ordonnanceur développé, nommé *AutoHeteroprio*, étend les fonctionnalités de *Heteroprio*, un ordonnanceur existant dans StarPU spécialement conçu pour les machines hétérogènes.

Heteroprio est une des stratégies d'ordonnancement disponibles par défaut dans StarPU, qui est connue pour ses performances élevées sur des architectures hétérogènes. Heteroprio, nécessite une assignation manuelle des priorités aux différentes tâches pour fonctionner efficacement, ce qui peut s'avérer être une charge considérable pour les développeurs. Cette assignation manuelle demande souvent une connaissance approfondie de la dynamique des tâches et une série de tests de performance, rendant le processus laborieux et sujet à erreurs.

Pour surmonter ces défis, nous proposons AutoHeteroprio, une version automatisée de Heteroprio. L'objectif principal de cette étude est de simplifier l'utilisation de StarPU en automatisant l'assignation des priorités, permettant ainsi aux développeurs de se concentrer davantage sur d'autres aspects de leurs applications sans sacrifier la performance. AutoHeteroprio utilise des heuristiques pour calculer de manière dynamique les priorités des tâches durant l'exécution, s'adaptant ainsi aux variations de l'environnement d'exécution et aux spécificités des tâches. Ces heuristiques ont été développées en se basant sur des exécutions simulées de faux programmes, permettant ainsi d'avoir une forme de "vérité terrain" pour guider les choix de conception. Nous avons déterminé plusieurs métriques d'intérêt sur lesquelles se basent les heuristiques, telles que le temps d'exécution (prédit), le taux d'utilisation du processeur, et le nombre de successeurs d'une tâche.

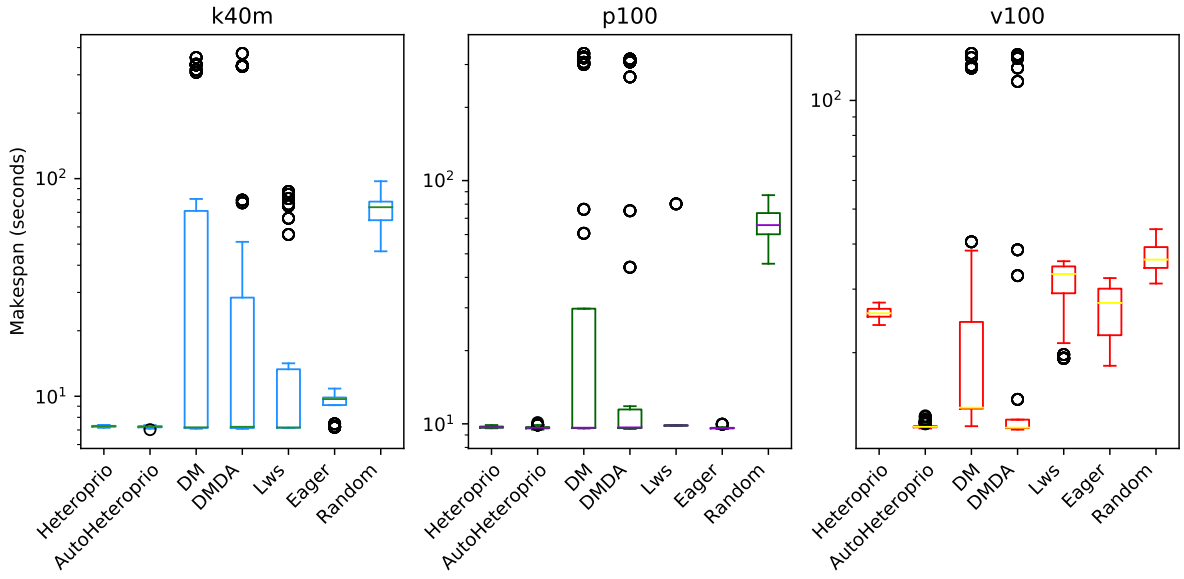


Figure 12.3: Temps d'exécution pour le test *testBlockedRotationCuda* de ScalFMM sur les trois configurations matérielles (k40m, p100 et v100). Les paramètres par défaut sont utilisés, avec 10 millions de particules. L'échelle de l'axe des ordonnées est logarithmique. Les boîtes à moustaches montrent la distribution des 32 temps d'exécution (896 pour AutoHeteroprio) pour chaque ordonnanceur.

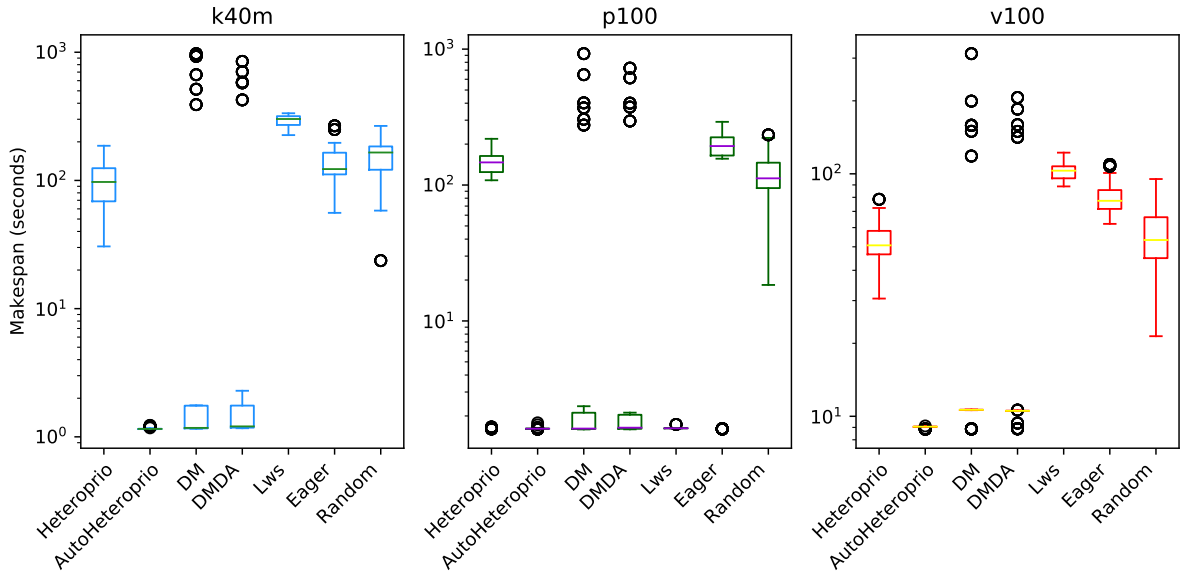


Figure 12.4: Temps d'exécution pour le test *testBlockedRotationCuda* de ScalFMM sur les trois configurations matérielles (k40m, p100 et v100). Une taille de block de 2000 est utilisée, avec une hauteur d'arbre de 7 et 60 millions de particules. L'échelle de l'axe des ordonnées est logarithmique. Les boîtes à moustaches montrent la distribution des 32 temps d'exécution (896 pour AutoHeteroprio) pour chaque ordonnanceur.

Pour valider ces heuristiques, des tests extensifs ont été menés, mettant en évidence que AutoHeteroprio peut souvent correspondre ou surpasser les performances de l'ordonnanceur Heteroprio (avec priorités définies

manuellement), tout en éliminant la nécessité pour les développeurs de spéculer sur les priorités optimales sur leurs tâches. Prenons par exemple l'application ScalFMM [10], qui est un code de calcul intensif en termes de calculs de type N-corps. L'algorithme, basé sur la méthode multipôle rapide (FMM), est constitué d'un calcul d'interactions entre particules de manière directe et indirecte, en utilisant une structure d'arbre. Cette application est un cas favorable pour nous, car Heteroprio figure déjà parmi les meilleurs ordonnanceurs pour cette application. Nous avons comparé les performances de différents ordonnanceurs sur ScalFMM, en utilisant deux configurations logicielles différentes. La première configuration (figure 12.3) utilise les paramètres par défaut de ScalFMM, tandis que la deuxième configuration (figure 12.4) utilise des paramètres différents.

Il est intéressant de remarquer que dans ce cas favorable, AutoHeteroprio surpasse tous les autres ordonnanceurs, y compris Heteroprio (version non-automatisée). On peut par exemple s'intéresser aux résultats du cas *v100* du premier cas. Dans ce cas, on voit que Heteroprio est de manière constante plus lent que AutoHeteroprio et DMDA (variation de HEFT), qui sont les meilleurs ordonnanceurs sur cette configuration. Il est cependant légitime de dire que AutoHeteroprio est meilleur que DMDA, car DMDA inclut de nombreuses valeurs extrêmes (élevées) qui montrent que ce dernier offre un ordonnancement moins "stable".

La capacité d'AutoHeteroprio à dépasser Heteroprio peut s'expliquer de deux façons différentes. Soit les priorités calculées par AutoHeteroprio sont meilleures que celles définies manuellement, soit AutoHeteroprio est capable d'adapter de manière pertinente les priorités en fonction de l'environnement d'exécution. L'exemple du premier cas sur v100 pourrait par exemple s'expliquer par la lenteur d'un kernel GPU sur cette configuration, qui n'est pas prise en compte par Heteroprio, dont les priorités sont pré-définies.

D'autre part, d'autres tests effectués montrent que le choix de l'heuristique a une importance relativement limitée sur les performances, typiquement de l'ordre de quelques pourcents de temps d'exécution. Cela montre que l'approche d'Heteroprio est moins dépendante du choix de priorités que ce que l'on pourrait penser.

Ainsi, nos travaux ont résolu une critique majeure que l'on pouvait faire à l'ordonanceur Heteroprio, à savoir la nécessité de définir manuellement les priorités des tâches. En automatisant ce processus, nous avons rendu l'utilisation de StarPU plus accessible aux développeurs, tout en conservant des performances élevées.

Cette section marque la fin de la discussion sur l'optimisation des calculs de simulation de fluides sur processeur graphique. Dans la prochaine section, le thème de la compression de données sera abordé. Nous y décrirons comment les ondelettes peuvent être utilisées pour compresser les données de simulation, et comment cette approche peut être intégrée dans des simulations de fluides pour mener à des gains effectifs de mémoire.

12.5 Intégration de compression de données grâce aux ondelettes

La Transformation en Ondelettes Discrètes (TOD) est un outil essentiel pour le traitement et l'analyse des signaux dans de nombreux domaines, y compris la compression de données. Ce processus décompose un signal en une série de coefficients, permettant de capturer des informations précises dans le temps et la fréquence. Des travaux fondamentaux ont été menés par Haar, Gabor, Grossmann, Morlet, et Daubechies, entre autres, jetant les bases théoriques et pratiques pour des applications comme la compression d'images JPEG2000.

Pour intégrer la TOD dans de la simulation de fluides, il est nécessaire de concevoir des ondelettes qui tiennent compte des besoins liés à la conservation des propriétés physiques, telles que la conservation de la masse et la gestion des discontinuités. C'est pourquoi nous avons accordé une attention particulière à la conception d'ondelettes biorthogonales qui répondent à ces exigences. Nous nous sommes intéressés à trois types d'ondelettes: LGT5/3, CDF9/7, et une variation des ondelettes de Haar. Nous avons modifié la conception de chaque ondelette pour atteindre différentes propriétés. Ces ondelettes possèdent chacune des caractéristiques uniques, ce qui permet d'adapter le choix de la TOD en fonction de la situation. La différence la plus notable entre ces ondelettes est le degré de filtrage qu'elles appliquent. Les ondelettes LGT5/3 filtrent les polynômes de degré 1, les ondelettes de Haar (modifiées) filtrent les polynômes de degré 2, et les ondelettes CDF9/7 filtrent les polynômes de degré 3. Le degré du filtrage est corrélé avec le potentiel de compression; plus le degré est élevé, plus la compression est (théoriquement) efficace.

Pour obtenir une compression effective, la TOD doit être combinée avec d'autres processus, tels que le seuillage (mettre à zéro les coefficients proches de zéro) et une compression sans perte. Le résultat final est

une méthodologie qui permet de compresser des données 2D d'un facteur de 10 à 100, tout en conservant une précision relativement élevée. La question est de savoir si cette méthodologie est adaptée à la simulation de fluides, notamment sur des simulations à grande échelle avec utilisation de processeurs graphiques. On peut imaginer deux problèmes majeurs: le coût de la compression et la perte d'accuité numérique.

Dans une première étude, nous évaluons la perte de précision due à la compression à l'échelle de la simulation. L'idée est de savoir si l'application répétée de cycles de compression/décompression peut mener à une perte de précision significative, qui n'est pas décelable sur un seul cycle. Les résultats semblent montrer qu'en pratique, on peut généralement trouver un seuil en dessous duquel la perte de précision est acceptable. Ainsi, on peut avoir confiance en le fait que cette méthodologie peut être utilisée pour réduire significativement les besoins en mémoire des simulations de fluides. Cependant, il reste à voir si le coût en termes de temps de calcul est acceptable.

Dans une deuxième étude, nous avons évalué le coût de la compression en utilisant des processeurs graphiques. Cette fois-ci, nous avons mis l'accent sur le débit de compression et sur les détails pratiques qui permettent d'observer un gain réel de mémoire. Nous avons vu qu'en tirant profit de la mémoire partagée des processeurs graphiques, il est possible de réduire significativement le nombre d'accès à la mémoire globale de notre algorithme de compression. Notre algorithme se base sur des TODs faites directement sur la mémoire globale, suivies d'une compression de type COO (format de matrice creuse) implémenté à la main.

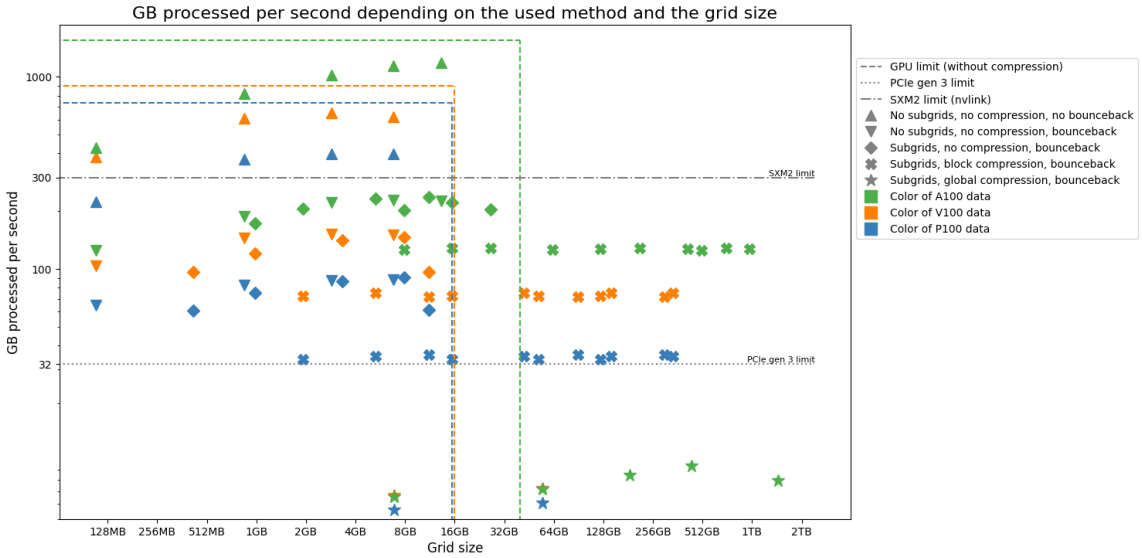


Figure 12.5: Évaluation des performances d'une simulation D3Q27 avec différentes configurations. L'axe des abscisses montre la taille totale de la grille (en GB) si elle était décompressée, tandis que l'axe des ordonnées affiche la vitesse de traitement (en GB/s). Le matériel (P100, V100, ou A100) est représenté par la couleur, tandis que le marqueur représente la méthode utilisée. Les marqueurs représentent le résultat pour une seule exécution. Les lignes pointillées colorées représentent les limites théoriques de chaque GPU testé en fonction de leurs spécifications, en supposant aucune compression.

La figure 12.5 montre les résultats de cette étude. On y évalue différentes méthodes: avec ou sans sous-grilles, avec ou sans compression, et deux méthodes de compression différentes. La méthode *block* correspond à notre nouvelle implémentation par blocs "shared", tandis que la méthode *global* correspond à une implémentation "naïve" où chaque étape de la TOD est faite directement sur la mémoire globale. Le graphique représente le débit de traitement en fonction de la taille de la grille (si complètement décompressée). Le débit de traitement, mesuré en GB/s, est une mesure théorique du nombre d'accès mémoires nécessaires pour le stencil, à supposer que l'on fait une lecture et une écriture par cellule. En principe, ce débit de traitement doit nécessairement être inférieur à la bande passante mémoire de la carte graphique, et idéalement, il devrait s'en rapprocher le plus possible. Or remarque entre autres que les débits de compression sont supérieurs

aux débits des bus PCI-Express, ce qui signifie que la compression à la volée est plus rapide que de faire transiter les données non compressées sur le bus PCI-Express. On a donc à la fois un intérêt en termes de gains mémoire et en termes de temps de calcul, parce que la compression nous permet d'éviter le goulot d'étranglement du bus PCI-Express.

En conclusion, nous avons montré que l'utilisation de la TOD pour compresser les données des simulations de fluides est une approche viable pour réduire les besoins en mémoire dans le contexte de simulations haute performance. Auparavant, la TOD était essentiellement utilisée pour du stockage de résultats ou bien de manière implicite dans des algorithmes de résolution de systèmes linéaires. Nous avons montré que les processeurs graphiques récents sont capables de gérer la compression à la volée de manière efficace, permettant ainsi d'obtenir des gains de mémoire significatifs.

12.6 Perspectives Futures

Les travaux présentés dans cette thèse ouvrent plusieurs pistes pour des recherches futures.

Premièrement, il existe plusieurs domaines d'amélioration pour le schéma de compression basé sur les ondelettes. La TOD elle-même peut être optimisée pour réduire le nombre d'accès à la mémoire et améliorer la localité des données. Actuellement, la TOD est effectuée successivement sur chaque dimension, ce qui n'est pas optimal en termes de localité des données. Nous sommes conscients que les algorithmes TOD multidimensionnels pourraient théoriquement résoudre ce problème, mais nous n'avons pas trouvé d'implémentation qui s'intégrerait correctement dans le paradigme de programmation des processeurs graphiques.

La méthode de compression sans perte peut également être améliorée. L'implémentation actuelle basée sur le format COO semble rapide mais repose sur la sparsité des données. Cela a posé problème dans notre dernière expérience, où les données étaient moins creuses que prévu, entraînant un taux de compression plus faible. Cela est probablement dû à la nature turbulente du flux testé, qui nécessite un seuil bas pour préserver la précision de la simulation. Il existe plusieurs autres méthodes de compression sans perte qui seraient moins dépendantes de la sparsité des données, telles que le codage entropique ou les méthodes basées sur des dictionnaires. Cependant, comme la compression sans perte doit être intégrée dans notre noyau de compression, nous devons être capables de fournir des implémentations rapides, ce que nous n'avons pas eu le temps de faire.

Une autre idée pour améliorer les performances est d'imbriquer la compression et la simulation dans un même noyau. Actuellement, les noyaux de décompression écrivent les données décompressées dans la mémoire globale, où les noyaux de simulation les lisent et les écrivent. Étant donné que les données décompressées apparaissent à un moment donné dans la mémoire partagée, il pourrait être possible d'effectuer les calculs physiques directement dans la mémoire partagée puis de compresser immédiatement les données pour les renvoyer à la mémoire globale. Cela réduirait considérablement le nombre d'accès à la mémoire globale, car seules les données compressées (moins volumineuses) donneraient lieu à des accès à la mémoire globale. Cependant, il n'est pas évident de trouver une nouvelle conception de simulation qui fonctionnerait avec cette idée, en particulier à cause du motif d'accès aux données, qui nécessite d'accéder aux données des blocs voisins, qui ne sont pas nécessairement disponibles.

Enfin, un autre domaine majeur d'amélioration serait une mise en œuvre flexible d'une simulation de fluides à grande échelle intégrant notre schéma de compression. Actuellement, notre implémentation multi-GPU StarPU ne permet pas l'utilisation de la compression, la rendant comparable à la plupart des solveurs stencil de pointe. Cependant, dans un travail futur, nous prévoyons d'intégrer le schéma de compression dans le solveur et d'évaluer l'impact sur les performances de différentes stratégies. Nous pensons qu'une approche entièrement dynamique, où les sous-grilles peuvent être remodelées en fonction des facteurs d'exécution, serait la plus prometteuse. Contrairement aux approches plus populaires, qui intègrent rarement la compression, notre approche pourrait être plus performante en termes d'équilibrage de charge en raison du coût réduit de l'envoi de données compressées. L'utilisation de blocage temporel serait également facilitée, car le coût de synchronisation accru serait compensé par la réduction du coût de transfert de données.

En conclusion, le travail présenté dans cette thèse fournit une base solide pour l'intégration de schémas de compression dans des simulations de fluides à grande échelle. Les résultats obtenus jusqu'à présent sont prometteurs, et nous croyons qu'avec des recherches et des développements supplémentaires, il sera possible d'atteindre des améliorations significatives pour le passage à l'échelle de ce type d'algorithme.

12.7 Remarques et Conclusion

Les travaux réalisés au cours de cette thèse ont mis en lumière des stratégies innovantes et efficaces pour l'optimisation des simulations de dynamique des fluides sur des architectures parallèles et distribuées, en mettant un accent particulier sur la compression des données. Grâce à l'adoption de la TOD et des techniques de compression avancées, nous avons pu obtenir une réduction significative de l'empreinte mémoire sur des simulations tout en préservant, dans une large mesure, l'intégrité et la précision des calculs.

Ces améliorations ouvrent la porte à des simulations plus vastes et plus complexes en optimisant l'utilisation des ressources disponibles. Cette thèse propose donc non seulement des solutions techniques à des défis spécifiques mais suggère également une manière plus durable et économiquement viable de mener des simulations avancées.

Cependant, plusieurs défis demeurent et doivent être abordés dans des travaux futurs. L'intégration de la compression dans le flot de travail de simulation en temps réel nécessite une attention particulière pour équilibrer de manière optimale le compromis entre la compression, la perte de précision et le coût en performances. De plus, les architectures de calcul évoluent rapidement, introduisant continuellement de nouvelles optimisations matérielles qui peuvent être exploitées pour améliorer encore les performances et l'efficacité des algorithmes proposés.

En résumé, bien que des progrès significatifs aient été réalisés, nous estimons qu'il est nécessaire d'adapter et d'affiner les stratégies utilisées pour être capables de se comparer aux moteurs de simulation de fluides les plus avancés. La voie est désormais tracée pour des innovations futures qui, espérons-le, continueront à repousser les limites de ce qui est possible dans le domaine de la simulation numérique.

Clément FLINT

Compression de données efficace pour les solveurs PDE haute performance

Résumé

Cette thèse se concentre sur l'optimisation des simulations de fluides dans des environnements à mémoire limitée via des ondelettes pour compresser les données, réduisant les besoins en mémoire tout en préservant la précision. Des ondelettes biorthogonales spécifiques assurent la conservation de la masse et un filtrage polynomial. Nous avons intégré ces méthodes dans les flux de travail des simulations, développant des stratégies pour les processeurs graphiques et ajustant les taux de compression selon les besoins en mémoire. Les résultats montrent une réduction significative de l'empreinte mémoire pour un coût de calcul faible. Des améliorations potentielles sont encore possible, ouvrant la voie à l'intégration dans des applications industrielles et de recherche en mécanique des fluides.

Résumé en anglais

This thesis explores optimizing fluid simulations in memory-constrained environments using advanced wavelet transforms to compress data, significantly reducing memory needs while maintaining accuracy. Tailored biorthogonal wavelets like LGT5/3, CDF9/7, and modified Haar wavelets ensure mass conservation and effective polynomial filtering. We integrated these methods into simulation workflows with attention to performance, developing strategies for graphics processors and balancing compression rates with performance trade-offs. Results demonstrate reduced memory footprints and potential performance boosts in scenarios where PCIe bus bandwidth limits speed. This research could significantly benefit industrial and research applications in fluid mechanics by managing large data efficiently, improving performance, and reducing energy consumption.