POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Boosting Performance of Compressible CFD on Exascale Systems

Tesi di Laurea Magistrale in
High-Performance Computing Engineering - Ingegneria del Calcolo ad Alte Prestazioni

Author: **Tommaso Trabacchin**

Student ID: 249478
Advisor: Prof. Fabrizio Ferrandi
Co-advisors: Prof. Diego Rossinelli, Prof. Patrick Zulian
Academic Year: 2025-26

Abstract

The resolving power of Computational Fluid Dynamics (CFD) is ultimately bounded by the fraction of nominal peak performance that can be extracted from the underlying computing hardware. We present a solver development approach that sustains 50% of nominal peak performance over more than half of the computational duty cycle, yielding over 30% of peak performance overall. Relative to Hypersonic Task-based research (HTR) solver, a state of the art solver featuring high order numerical schemes for supersonic and turbulent flows, this corresponds to an approximately 10X reduction in time to solution for the same underlying numerical schemes, implemented through a substantially different software and kernel design. These results hold across multiple exascale systems, both CPU based and GPU based. The techniques that enabled this result require rethinking how HPC software is built, from performance modeling and kernel synthesis to how design decisions are explored.

Keywords: Computational Fluid Dynamics, Performance modelling, Program synthesis, Exascale HPC performance

Abstract in lingua italiana

Il potere risolutivo della Fluidodinamica Computazionale (CFD) è in ultima analisi limitato dalla frazione della performance nominale di picco che può essere effettivamente estratta dall'hardware di calcolo sottostante. Presentiamo un approccio allo sviluppo di solver che mantiene il 50% della performance nominale di picco per oltre metà del ciclo computazionale, raggiungendo complessivamente oltre il 30% della performance di picco. Rispetto al solver Hypersonic Task-based Research (HTR), uno solver all'avanguardia che utilizza schemi numerici di alto ordine per flussi supersonici e turbolenti, questo corrisponde a una riduzione del tempo di soluzione di circa 10 volte per gli stessi schemi numerici sottostanti, implementati tramite un design software e dei kernel sostanzialmente diversi. Questi risultati sono validi su più sistemi exascale, sia basati su CPU sia su GPU. Le tecniche che hanno reso possibile questo risultato richiedono di ripensare il modo in cui il software HPC viene costruito, dalla modellizzazione delle prestazioni e sintesi dei kernel fino all'esplorazione delle decisioni di progettazione.

Parole chiave: Fluidodinamica Computazionale, Modellazione delle prestazioni, Sintesi di programmi, Prestazioni HPC exascale

Contents

Introduction

From aerodynamics to biological systems, fluid flow phenomena are ubiquitous in science and engineering. Understanding, predicting and controlling their dynamics is therefore crucial. In practice, fluid behavior is studied through two complementary approaches [7].

Through experimentation and observation, flow phenomena can be investigated in laboratories. Observables can be measured through an acquisition pipeline, with finite resolution and finite field of view. In practice, quantitative access is limited to a projection of the flow field or to a very limited region of interest, such as a cross-section. In addition, many volumetric or derived quantities are not measured directly but inferred, and the uncertainty behind such inference is governed by the measurement model accuracy, calibration, and optical access. Moreover, experiments can be expensive and slow to modify because of external constraints imposed by facilities and diagnostics, and they may experience severe run-to-run variability. Safety constraints further restrict the feasible operating envelope in reactive, high pressure, or high temperature systems, which can force test conditions that are less representative of the target configuration.

Computational Fluid Dynamics (CFD) is a complementary approach to alleviate the limitations of experimental investigations. Computer simulations can probe the modelled fluid state directly and without disturbing the flow. This is a simple operation that is impossible to carry out in the lab. The major shortcoming of CFD is arguably the ability to resolve spatiotemporal scales present in the experiments [7]. In simulations, flow fields are represented by a finite set of degrees of freedom, and these evolve through finite precision arithmetic computation. Therefore, only a limited and distorted band of the spatiotemporal spectrum can be accessed. In turbulent flows, the range of dynamically active scales grows rapidly with the Reynolds number, which makes resolving all relevant scales prohibitively expensive in three dimensions, even on modern supercomputers.

Despite the introduction of turbulence models, spatiotemporal resolution remains the single key player for simulation quality [7]. Closures can represent the net influence of unresolved scales on the resolved ones, but they cannot reconstruct missing spatiotemporal content or “invent” frequencies that are not captured by explicit sampling. For a fixed

wall clock budget on a computing system, the portion of spatiotemporal scales that we can capture is determined by how much of the system's nominal peak performance and memory bandwidth we can extract. High order discretizations are a cornerstone of high fidelity CFD because they attenuate distortions of the represented spectrum, but they lead to more expensive and non-trivial instruction streams, making the goal of achieving a high fraction a grand challenge. These constraints motivate three conflicting software goals in large scale CFD: sustain high fractions of peak performance (i), maintain performance portability across diverse architectures with essentially one code base (ii), and (iii), preserve software productivity so that new models, discretizations, and workflows can be introduced without rewriting the full performance critical stack.

This thesis proposes and assesses an approach to achieve all three goals.

0.1. Supercomputing for CFD: benefits and challenges

Computational resources have an indirect but decisive impact on the quality outcome of CFD investigations. Supercomputing enables ever-growing simulation fidelity and has reached exascale capabilities. However, effectively leveraging such systems for CFD is a non-trivial task, as elaborated in the following section.

Achieving high fraction of peak compute performance. Simulation quality is primarily determined by the ability to correctly resolve as many spatiotemporal scales as possible within a practical time frame. In turn, this depends on the simulation software's ability to leverage the underlying computing infrastructure within a given wall-clock time budget. Several metrics exist to quantify this ability.

For our work, we rely on the achieved Fraction of Peak, that is:

$$\text{Fraction of Peak} = \frac{\text{Achieved Performance}}{\text{Nominal Performance}}, \tag{1}$$

where Nominal Performance is the theoretical nominal compute peak, and Achieved Performance is defined as follows:

$$\text{Achieved Performance} = \frac{\text{TOTAL FLOPs}}{\text{TTS}}, \tag{2}$$

where TTS is the simulation's Time-To-Solution and TOTAL FLOPs refers to the number of FLOPs executed. Such operation count must only account for "algorithmic" FLOPs, excluding any redundant calculation. Moreover, we choose a metric that considers the en-

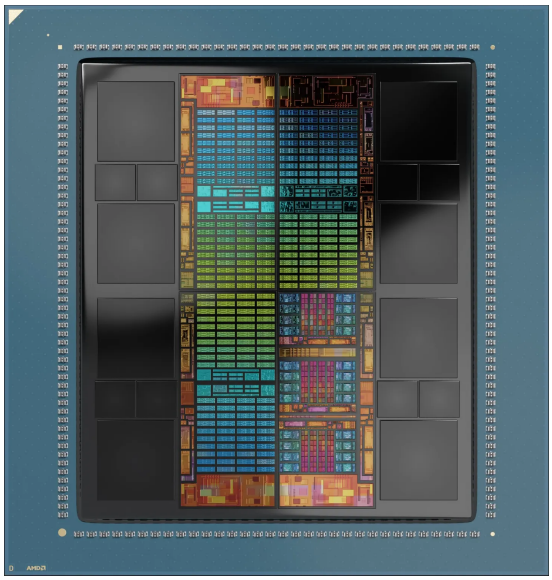
tire simulation execution, avoiding the ambiguities associated to per-kernel performance assessment, due to task-scheduling complexities.
Fraction of Peak is a very informative metric, as it represents how efficiently the computational resources are utilized, and its inverse sets a hard upper-bound on the ultimate windows of spatiotemporal scales that are accessible within a certain TTS and compute allocation.

However, achieving a high fraction of the peak, for high-order CFD software, represents a formidable challenge, and it is not unusual for such programs to reach no more than 5% [6, 11, 13, 17, 21]. The reasons behind this difficulty lies in part to the deep understanding of the underlying microarchitecture required, and involves non-trivial trade-offs choices in software development.

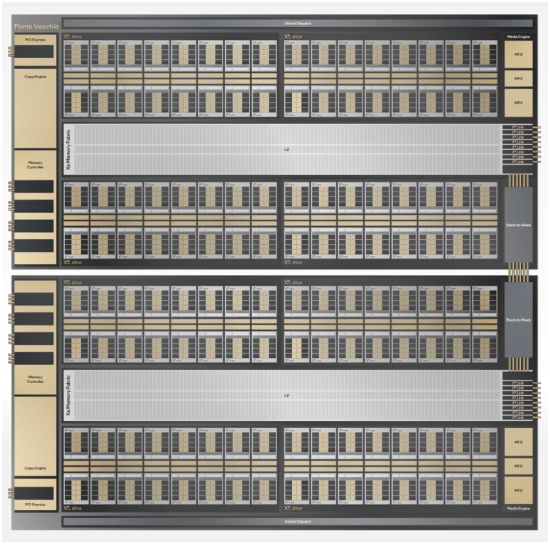
Performance portability. Performance portability of a software is its ability to achieve consistent performance on several different platforms. In general, this is a difficult aim, and it becomes even harder when the target platforms significantly differ in their microarchitectural details. To address (at least partially) this challenge, it is often needed to implement source code-level optimizations. Unfortunately, the current exascale landscape, despite comprising only three systems, is characterized by a wide variety of CPUs and GPUs microarchitectures. While a detailed discussion of their details is beyond the scope of this thesis, as an example, we can cite the differences in the GPU architectures. The AMD MI300A and the AMD MI250X GPUs (Figure 1a) (mounted respectively on the *El Capitan* and the *Frontier* supercomputers), and the Intel GPU Max 1550 (Figure 1b) (mounted on the *Aurora* supercomputer) feature different execution models, which reflects the different nature of the underlying hardware. In particular, the former adopt the most common GPU execution model, that is the Single Instruction, Multiple Threads (SIMT) one. It consists of a fusion of Thread Level Parallelism (TLP) and Data Level Parallelism (DLP), while the Intel GPU relies on Simultaneous Multi Threading (SMT), which clearly separates TLP from DLP.

Software productivity. By software productivity we mean the ability to extend and adapt the software to support new and unforeseen features. Scientific computing has always been affected by a low software productivity, which has resulted in a considerable reduction in advancement pace in many scientific and engineering fields. The reason for such an issue has been found to lie in the following factors:

- Ensuring code correctness,
- Optimization and tuning of serial code version,



(a) System architecture of the AMD MI300A GPU



(b) System architecture of the Intel Max 1550 GPU

- Optimization and tuning of parallel code version,
- Porting and optimization of existing codebase for different platforms.

Correctness of scientific software is of paramount importance, and, in this regard, complexities in mathematical models and algorithms employed represent a significant challenge. Moreover, as performance is often crucial in high-performance computing, extensive optimization and tuning of source code is needed. In particular, given that parallel architecture are at the core of supercomputing, scientific software needs to be accordingly adapted, which involves significant effort. Lastly, considering the available computing platform are continuously evolving, scientific software needs to be accordingly updated, to continue to deliver state-of-the-art performance. However, as mentioned in the previous paragraph, this often demands code specializations, which makes any new feature implementation a complex and expensive task.

Conventional Approaches. A single unified road to address the previous challenges has not emerged yet. However, a variety of strategies have been proposed in the recent past. They can be summarized as:

- Hand-written kernels,
- Compiler directives,
- Domain-specific languages.

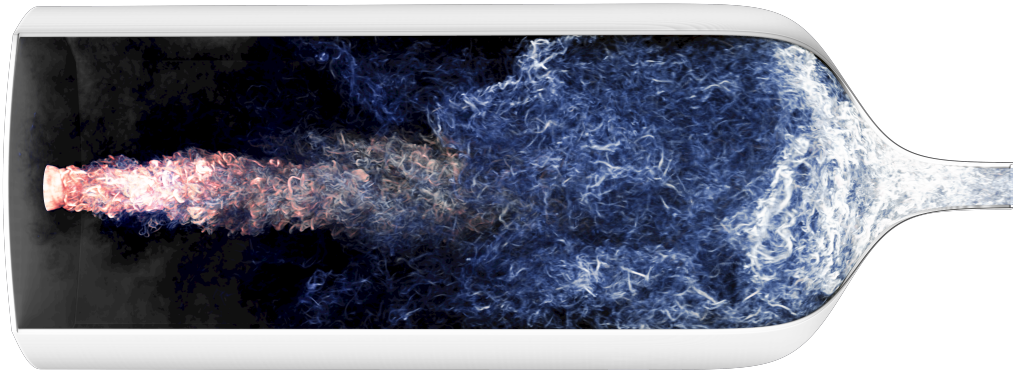
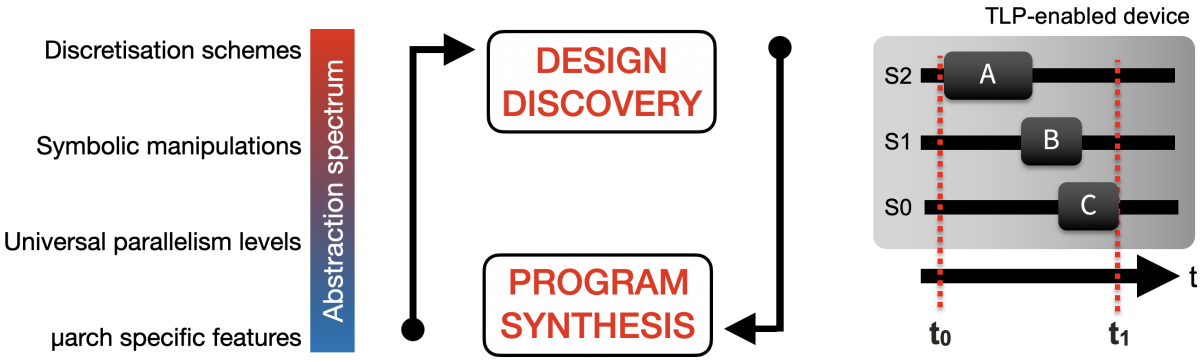


Figure 2: Laser ignition process of gaseous methane/oxygen combustion, conducted in [30]

Kernel hand-writing [3, 39] is the approach that seems to deliver the best performance, as it ensures target platform features (even undisclosed ones) can be fully leveraged. However, this tends to aggravate both performance portability and software productivity. A different strategy is based on the use of compiler directive-based tools, such as OpenACC and OpenMP [33]. They allow offloading compute kernels to specific devices (e.g., GPUs and FPGAs) and expressing parallel patterns by annotating the source code with concise compiler directive tools. Software productivity is favoured, as the resulting code is rather platform agnostic, at the expense of performance portability and achieved fraction of peak performance. Lastly, Domain Specific Languages [20, 35, 36, 40] are programming languages that allows expressing software at the appropriate abstraction level. Prominent examples of DSLs in scientific computing includes Litsz [12] (for the solution of Partial Differential Equations) and Spiral [32] (for Digital Signal processing). This approach seems to be the strategy finding the best tradeoffs, as they express computational patterns at a level that allow for algorithmic and performance optimizations. A DSL seem better positioned to navigate the optimization decision space because it has access to a wider portion of the abstraction spectrum.

0.1.1. This Thesis

Motivation. This thesis work originated in the context of the INSIEME project under PSAAP III, founded by the US Department of Energy and the National Nuclear Security



(a) (b)

Figure 3: Bottom-up quantitative approach to discover successful HPC software designs, and subsequent top-down program synthesis to expand the software scope (left). Timeline view of an oversubscribed, threaded device (right). Kernels A, B, and C occupy logical streams S2, S1, and S0 and run between t_0 and t_1. Consequently, performance measurement for an isolated kernel does not directly relate to the overall performance.

Administration. Its main objective is to exploit Exascale systems in order to predict complex multi-phisics phenomena. An example of the simulations carried out as part of the project is shown in Figure 2. Currently, the main CFD solver used within the project is HTR [13], whose performance, as they align with the current state-of-the-art, represents a limitation on the resolvable scales. This work aims at providing an alternative solver, which we call *Lean Vertical Rewrite* (LVR), that can achieve significantly higher performance [37], thus enabling conducting higher quality simulation.

Contributions.

- We present a compressible flow solver (LVR) able to achieve 35% of the nominal peak performance
- 35% of peak is measured across diverse computing platforms
- The software was developed in 3 months by 4 researchers.

We are not aware of any related work that achieved similar performance in a comparable amount of time.

Moreover, following the ideas proposed in [37], we propose a novel and robust approach for performance analysis and reporting (see Figure 3b), based on the fraction of achieved floating-point performance. This overcomes the limitations of other metrics (e.g. per-iteration wall time, normalized execution time or strong/weak scaling efficiency) that are

more commonly used in CFD, but that fail to capture whether the targeted systems are being efficiently exploited.

We also show how a range of performance modelling and analysis techniques can be leveraged to guide the development approach from its early stage (see Figure 3a). Since we do not take advantage of any specific feature of CFD, we believe our method could be extended to other fields. However, this remains as future work.

Structure. This thesis is structured as follows: in chapter 1, prior works relevant to this thesis are reported. In chapter 2, we review some fundamental HPC concepts, following the ideas in [37]. In chapter 3, the governing equations and numerical schemes employed for LVR are described. In chapter 4, a detailed description of LVR is provided, and in chapter 6 our code generation pipeline is presented. Lastly, in chapter 7, LVR is validated, and its performance are assessed. Additionally, an example of the simulations conducted through it is reported.

My Contributions. Under the guidance of Diego Rossinelli, I had the possibility of contribuiting significantly to the development of *LVR*. Specifically, I developed all the necessary plugins and code base to maintain the initial very competitive performance obtained on a single MI300A, to the system level (LLNL Tuolumne), and then developed the tooling and additional plugins to preserve the performance level to Intel CPU and GPU architectures. Tooling that I developed also involved testing and implementing of non-trivial boundary conditions (so called NSCBC), and contributed to the computational modeling discussed chapter 3.

In terms of performance modeling and analysis (chapter 5), I studied the impact of register pressure arising from code generated by our program synthesis pipeline on Intel Sapphire Rapids and ARM-based Grace CPU, and discovered performance issues due to microarchitecture limitations, I improved the software performance of the base software. I also implemented an MPI communication scheme that allows overlapping network communication with ongoing computation, which contributed to reduce the network impact on the Time-To-Solution. This is particularly difficult to accomplish on GPU-accelerated systems. On NVIDIA-based systems, AMD-based systems, and Intel based systems I leveraged my contributions to conduct computational studies with LVR including simulation prototyping. I conducted large-scale investigations of the Triple Point Shock problem on both the Alps and Aurora supercomputers.

Publications. This thesis builds largely on two publications: "Crafting Software in HPC: A Quantitative Approach", published in the Center for Turbulence Research 2025 Annual Research Briefs [37], and "Boosting CFD to 30% of Peak on Exascale Systems".

The first publication presents a set of guidelines aimed at supporting the development of high-performance computing (HPC) software and includes a performance analysis of LVR, benchmarked against the HTR solver, to which I contributed substantially and which I wrote to a large extent. The report is publicly available at this link.

The second publication is a paper currently in preparation, for which I am the first author, and which I plan to submit shortly to the Computer Methods in Applied Mechanics and Engineering journal. This work places greater emphasis on the proposed performance modeling methodology and on the program synthesis pipeline. The manuscript and related materials are available at this repository.

This thesis additionally provides further details on LVR—its design, development process, and experimental results—which could not be fully included in the aforementioned publications due to space and scope limitations.

1 | Prior work

In this chapter, a summary of prior works related to our work is provided.

1.1. Compressible flow solvers targeting GPUs

In [9, 10, 33], the authors port the MFC multiphase compressible flow solver [4] to NVIDIA GPUs with OpenACC, achieving 84% strong and 97% weak scaling efficiency. There, the WENO and Riemann solver kernels achieve about 40% and less than 20% of the peak compute throughput, respectively. Metaprogramming is used to enable compile-time optimizations such as treating user inputs as constant and automatic subroutine inlining.

In [3, 39], the authors present STREAmS, a DNS compressible flow software for heterogeneous architectures. The STREAmS kernel development is described as a mixed of hand-written and directive-based strategy using the directive CUDA Fortran tool *cuf*. For targeting AMD GPUs and HIP framework they use HIP's standard code translation tools. STREAmS achieves between 75% and 79% of the peak memory bandwidth, while the WENO kernel achieves between 10% and 30% of the peak compute throughput, with weak scaling efficiency up to 98% on hundreds of nodes.

In [9, 10], the authors employ OpenACC for portability on NVIDIA and AMD GPUs, and report 20% fraction of peak as the highest achieved by their kernels on any of the platforms.

HTR (Hypersonic Task-based Research) [13], is a multicomponent compressible flow solver, capable of modelling thermochemical phenomena induced by high temperatures, such as vibrational excitation and chemical dissociation. HTR is written using the Regent domain-specific language [34] based on the Legion runtime [23], a task-based alternative to OpenACC to target heterogeneous CPU/ GPU (NVIDIA) architectures. Only performance in terms of weak and strong scaling efficiency is reported.

OpenSBLI [20] is a framework for code-generation in the context of compressible fluid dynamics on heterogeneous architectures. The framework is able to generate a complete

solver, starting only from the problem governing equations, discretization strategies, and boundary/initial conditions It leverages the SymPy Python library for symbolic manipulation and targets the Oxford Parallel Structured (OPS) [35] domain-specific language, showing how automatic code-generation allows optimizations that would be difficult to achieve in handwritten code. Only relative performance metrics to TTS are reported. However, a successive work [28] demonstrates the use of the framework to generate a solver with a reduced memory traffic, showing a 10% fraction of peak performance on the NVIDIA V100 GPU, and a 22% fraction of peak on the Intel Xeon Silver CPU, respectively a 10-folds and 3.5 folds improvement with respect to the baseline.

1.2. Compressible flow solvers targeting CPUs

A compressible flow solver for turbo-machinery simulation is presented in [15]. The authors report an outstanding achieved overall fraction of peak compute performance of 13%. In [36], authors describe a the optimization, through the OPS DSL (discussed below), of the CloverLeaf compressible CFD solver [29]. Results show an achieved 5% fraction of peak performance for the optimized version, while the original baseline is almost two times slower.

Other software focus on friendly programming APIs [16] or fine-grained software components promoting separation of concerns [4].

1.3. Domain specific languages, Code generators and autotuners

Domain specific languages (DSL) enable the navigation of the complete abstraction spectrum by driving automated kernel synthesis. The DSL describes high-level computations that a dedicated domain specific compiler uses to automatically generate instruction streams tailored to specific microarchitectures.

OPS (Oxford Parallel library for Structured mesh solvers) [35] is an embedded domain-specific language for writing multi-block structured mesh algorithms. Such algorithms are described by means of simple coding abstractions, and executed by highly parallel and automatically optimized workflows.

ExaStencils [40] is a code generator framework for multigrid solvers, based on the ExaSlang external multi-layer DSL. Each layer sits at a different level of the abstraction spectrum, from high-level definition of mathematical equations to low-level program specification.

Code generation is complemented with code optimization techniques, such as polyhedral optimizations and redundancy elimination [22].

HyTeg is a code generation framework for finite element matrix-free solvers. Its kernel synthesis component features several advanced code optimization, including loop invariant identification and common subexpression elimination. Per-kernel fraction of compute peak achieves 62% of nominal performance.

While most parallel frameworks require the programmer to explicitly formulate parallel strategies, task-based frameworks like Cilk(++), Intel's Thread Building Blocks, Charm++, and Regent [34] programming models are based on tasks and their mutual data dependences. Such dependencies are described by graphs across tasks, and the runtime will dynamically discover and dynamically exploit available parallelism, e.g., with work-stealing.

The correct tuning of performance relevant parameters is a necessary step to achieve the full potential of any parallel code. Such tuning is often performed manually, but mature computational software frameworks automate this process as part of their development cycles. Autotuning strategies include both brute-force as well as sophisticated search algorithms. In [32], autotuning guides formula transformation and code generation. Authors remark the convenience of relying on compiler options to control the autotuning process, reporting more performant code at the expense of performance portability.

In [24], the autotuning limits the maximum number of per-thread registers to increase occupancy of the GPU, and achieve an average 10% performance improvement on various benchmarks.

1.4. Roofline Model

The Roofline Model [45] is a performance model that provides insights concerning the compute or memory boundness of a program, based on the platform compute and memory bandwidth peak. It was introduced to address the arising complexity of multi-core CPUs at that time. Since then, in HPC, other factors have become relevant and increasingly difficult to model, such as the network or the CPU-GPU interconnects (PCI Express). The Roofline model alone fails to account for all these aspects, and may prove inadequate to remain as insightful on current hardware as it used to be when it was first introduced.

1.5. Symbolic performance modelling

In [1], a performance model framework is proposed. It is based on in-code assertions, which are used to annotate functions or single code-block. Through assertions, users can specify the FLOP count, memory, and network footprint for each annotated code region. However, manual FLOP counting is needed. Importantly, an actual implementation of the application is not necessary. From such assertions, a performance model, (in the form of MATLAB/Octave code) is produced.

1.6. Fraction of Peak

As achieved overall fraction of peak performance is rarely reported for compressible flow solvers, we decided to evaluate this metric for some of the publicly available solvers described above. In particular, we performed such analysis for the STREAmS-2 and Uranos solvers. For both of them, the most recent version available at the time of writing was used. They were compiled with the 12.3.0 version of GNU Fortran and C++ compilers, and by passing the `-Ofast -march=native` options. The solvers were run on a single socket of a NVIDIA Grace Superchip, launching 72 MPI ranks. Performance (i.e., the FLOP count and the TTS) were measured with the Linux perf tools. To the best of our abilities, we configured both solvers to ensure the same numerical scheme as LVR were used. We also ensured to disable any form of I/O to the file system. The STREAmS-2 solver achieved 5.44% of the compute peak performance, while Uranos attained 3.6% of the same metric. The aforementioned solvers have been developed to support a wide range of applications, hence prioritizing generality and flexibility over optimal performance for specific problems. Nevertheless, we consider such results as indicative of the average performance usually achieved in the field of compressible flow solvers.

In this context, our work aims at proving that achieving a high fraction of peak in compressible flow simulations is feasible, and that such a performance can be reached on several exascale architectures.

2 | HPC Fundamentals

In this chapter, we discuss some high-performance computing guidelines that, directly or indirectly, influenced the LVR's development. We believe their applicability extends beyond this particular use case.

In section 2.1 we focus on techniques that can serve as guidance for HPC software development. In section 2.2, we discuss how to leverage computing platforms to extract as much performance as possible.

2.1. Software

The principle that guided LVR's development is that successful HPC software is discovered through measurements, rather than through design. This requires exploring the abstraction spectrum bottom-up (Figure 3a), quantitatively, starting from microbenchmarks that run near hardware limits, and adding application features until performance collapses. We bisect the performance space, understand the cause, address it, and refactor the code to clean up. This stands in contrast to top-down designs, which can proceed without ever seeing throughput close to nominal levels.

While bottom-up design is our approach of choice, it is human engineered and manual. Each layer can, in principle, be improved and tailored to a specific target platform with autotuners. Moreover, one may wish to solve slightly different equations that benefit from the schemes discovered in the bottom-up phase. Given the intensive effort required for the bottom-up pass, we wish to avoid repeating the process. After the bottom-up traversal, we instead go back down again (see again Figure 3a) with program synthesis. If we can capture the execution patterns that we have shown to run exceptionally well, we can reuse the resulting framework to target different governing equations where applicable, and deploy rapidly for cases we have not anticipated to cover. Program synthesis framework therefore is our way to expand the application scope of what we have learned vertically, bottom-up.

2.1.1. High performance

Our primary goal is to enable software execution to reach high performance, i.e., find ways to minimize TTS, approximated as

$$\text{TTS} = \frac{\text{IC}}{\text{IPC} \times f}, \tag{2.1}$$

where f is the effective clock frequency, IC is the effective instruction count of the instruction stream implementing the selected numerical schemes , and IPC is the average number of instructions executed per clock cycle. This minimization problem is hard to solve, as it involves conflicting minimization subgoals. The diversified presence of computing units across a system results in multiple frequencies, instruction counts, and IPCs, make the minimization problem ambiguous and even more challenging.

2.1.2. Introspection.

Inspect assembly

Within a programming language, source code can be considered a message to the compiler. To verify receipt from the compiler as intended, a disassembler (e.g., objdump -S or cuobjdump -sass) inspects the actual produced machine code of the object file. Thereafter, we are able to inspect assembly listings and search for patterns (e.g., heavy-duty loops, presence of SIMD instructions, absence of control flow, and unit-stride memory access). We accelerate our skills in effectively communicating to the compilers with online learning tools like Godbolt. When switching off some components in a debugging session in release mode, the compiler sometimes removes instruction streams that depend on the disabled code, defying the debugging purpose. Without checking the assembly listing, this issue would remain undetected.

Synthetic microbenchmarks

The building blocks of HPC software are synthetic microbenchmarks. These minimalistic kernels target specific hardware features and measure how close we can get to nominal rates. Microbenchmarking is also used to detect skill issues, that is, our inability to extract the target throughput even in a synthetic setting. In such a case, how can we then expect to leverage it in the field?

Since they are at a low abstraction layer, microbenchmarks are straightforward to main-

tain, and are reusable between target projects. Once validated, microbenchmarks may be deployed to characterize new hardware, with STREAM [25, 26] and mpiBench [18]being the most frequently used, in addition to internal ones.

Microbenchmarking can lose predictive accuracy in a proper application, as different hardware features may contend the same resources. We can make an analogy to a Taylor expansion of performance as a 2D function. Linear terms in the two directions represent microbenchmarking of two separate features. The non-linear terms then represent the interaction between the features, such as resource contention, uncaptured by individual microbenchmarks. However, they still remain crucial for providing upper bounds that contextualize observed results.

Symbolic layer

Ideally, software design choices would be postponed until microbenchmarking is conclusive enough. But sometimes we want to extend a previously developed software to support new physics, or we have to make decisions on the primary memory layout. The strategy we choose is consequential, as it constrains the freedom of the lower layers of abstraction.

A symbolic framework (such as SymPy) provides an overview across the abstraction spectrum. The lowest layers are represented by data rate curves (Figure 2.1a); the middle layers, by symbolic expressions of what kernels compute; and the top layers, by the composition of such expressions. With this representation, payload sizes and operation counts become trivial queries, in turn enabling us to model the attainable performance of different scenarios. For given problem and simulation configurations, we can predict performance by comparing competing timescales (Figure 2.1b): computation, data transfer, and network communication, etc., and we can configure our problem such that no nominal timescale drastically dominates the others. Our approach differs from the roofline model and its advice [46]. In addition to timescales of system memory bandwidth and computing throughput, we consider network and multiple I/O timescales representing contemporary compute nodes at higher fidelity. Also, roofline-based advice encourages us to have higher operational intensities, and to stop after reaching the ridge point. If operational intensity is above the ridge, we actively decrease it, by either decreasing instruction count or trading computation for more data access, in order to reach overall higher utilization of the system components, pruning TTS further. Besides providing a comprehensive vertical performance model, a symbolic framework is the enabler of code synthesis.

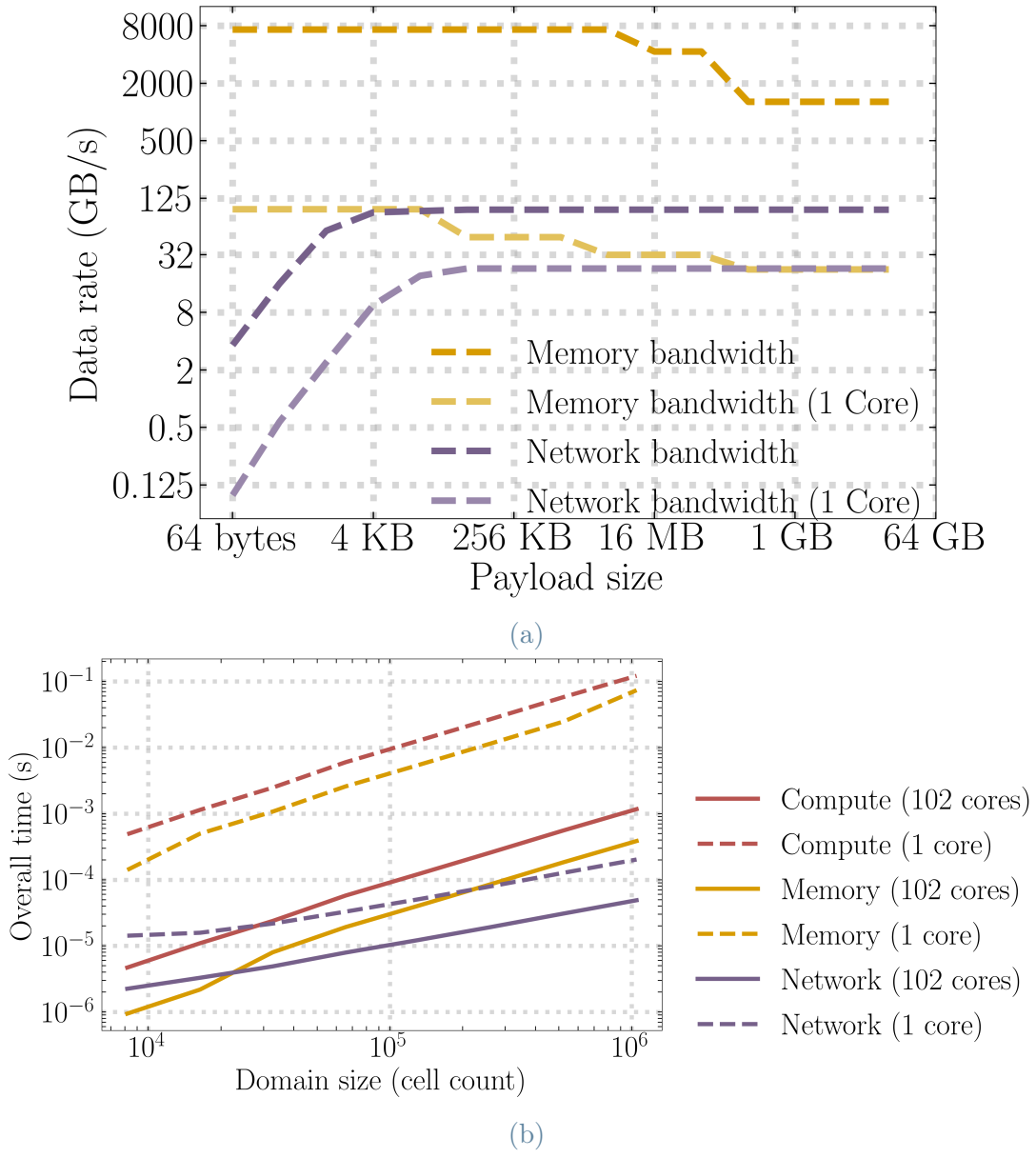


Figure 2.1: (a) Data rates profiles extracted from synthetic microbenchmarking on the ANL Aurora supercomputer, showing curves absorbed in the symbolic framework; (b) coarse-grained view of the abstraction spectrum, which allows for accurate performance modeling in terms of the timescales involved in the simulation.

2.2. Hardware

2.2.1. Throughput

We distinguish among three main forms of parallelism: instruction-level (ILP), data-level (DLP), and thread-level parallelism (TLP) [31].

Explicit forms of parallelism, including DLP and TLP, have grown considerably in the past two decades.

Implicit ILP

CPU cores discover ILP on the fly; they reorder and dynamically schedule instructions to hide hazard latencies. From the 1980s to the 2000s, architectural advances in ILP roughly doubled the performance rate of RISC microprocessors, yielding about 52% annual improvement versus about 25% from transistor scaling alone [19]. This enhanced extraction of ILP has led a major convenience to programmers, with the peak point arguably reached by out-of-order execution models [41], used today across most CPUs from Intel, AMD, as well as some ARM and RISC-V architectures. Combined with architecture-aware compilers, ILP is no longer a programmer's problem: we can simply write code as a linear and logical sequence of expressions. GPUs, by contrast, execute in order. Although SIMT, the execution model used on GPUs, is the primary latency-hiding mechanism for hazards, ILP can still be a limiting factor. This forces the programmer to rely on explicit source-level techniques to expose ILP (e.g., [43, 44] or software pipelining), which remains tedious even with contemporary macro preprocessors.

Deceiving DLP

On CPUs, DLP is implemented and exposed through SIMD instruction sets. Despite recent progress in auto-vectorizers, DLP is where communication with the compiler still tends to break down. Aside from choosing a correct memory layout, discussed later, persuading the compiler to emit DLP is a laborious exercise.

C compilers are extremely capable at mapping processing schemes on multicomponent one-dimensional arrays to SIMD instructions, but they struggle to capture multidimensional access patterns. We extend the DLP to multidimensional arrays via a simple compiler deception: we present the data as one-dimensional representations of multichannel arrays, and the same multidimensional algorithm starts leveraging DLP.

Deception techniques are workarounds targeting scalar programming models. Other lan-

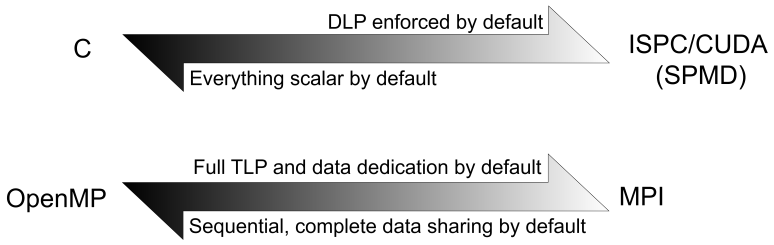


Figure 2.2: Assumption spectrum of programming models for DLP (top) and TLP (bottom).

guages do not require such workarounds because they operate on opposite assumptions, as depicted in Figure 2.2. Inherited from shading programming/SPMD practice, these include CUDA and HIP on GPU, and ISPC on CPUs/GPUs. DLP-friendly programming models can target both CPUs and GPUs. However, SIMT, is considerably more flexible, as it seamlessly fuses DLP and TLP.

Troubleshooting TLP

TLP is a critical performance lever in hardware designs, which often feature tens to thousands of processing elements. It not only scales computational throughput, but also significantly impacts data rates by driving and saturating DRAM and network bandwidth. Effective placement, considering hierarchies like sockets, NUMA nodes, and chiplets, is crucial; ignoring this can result in a 20% or more reduction in aggregate data rate. Individual cores are limited by line fill buffers, extracting only a small fraction of DRAM bandwidth. Therefore, multiple cores are employed to sustain memory requests in flight, with specialized cores dedicated to data movement to ensure communication keeps pace with computation. Interprocess communication via shared memory facilitates coordination among these roles. On GPU nodes, core specialization and TLP are particularly vital, as primary processes manage work dispatch to devices (Figure 2.3).

TLP manifests as threads and processes (both scheduled equivalently on Linux via the clone family; see Figure 2.2). While OpenMP threads initially appear attractive due to shared address space and simple pragmas, complex data flows introduce significant debugging challenges (e.g., deadlocks, races, and memory inconsistency). Optimal patterns often involve non-interacting workers operating on dedicated data. Furthermore, thread creation/destruction, though rapid, lacks direct hardware counterparts. Job schedulers manage processes with clear policies for placement and oversubscription, whereas threads necessitate their own binding semantics and are prone to oversubscription.

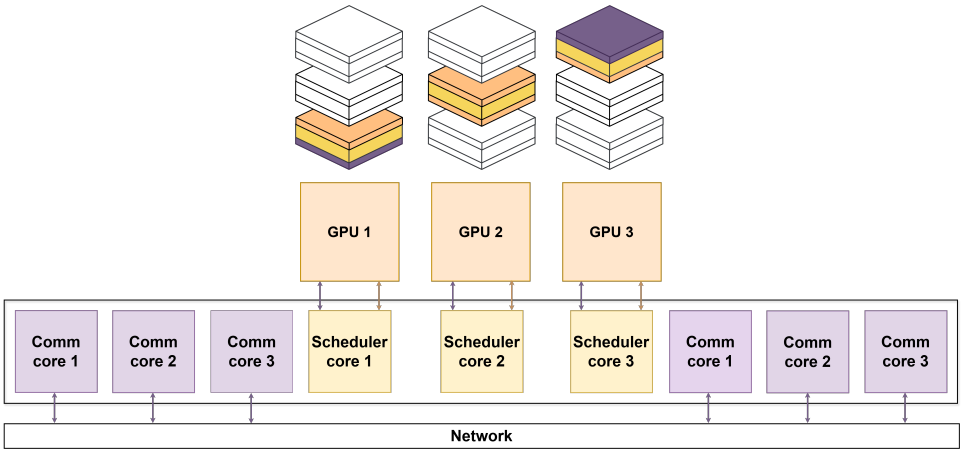


Figure 2.3: Core specialization is essential for maximizing DRAM and network bandwidth at the node level.

Synchronous programming

All major components—cores, memory interfaces, buses, and NICs—are clocked. At the timescales that matter on HPC nodes, we can treat the system as synchronous, with deterministic delays. Asynchronous software for HPC has always been a subject of academic interest and research. Unless one uses coroutines or nonstandard programming languages, source code of asynchronous software implies the separation of program execution from state transitions. The instruction stream no longer dictates when and in what order execution state changes, which is error prone. The resulting blooming of dependencies and event handling makes reasoning and debugging much harder. Our practice is to rely on blocking calls and explicit local barriers because they are easy to reason about and debug. We then avoid penalties by building explicit latency-hiding mechanisms with double buffering within well-marked phases.

System-level awareness

Computing performance is unlocked when software stack and runtime are aligned with the system architecture. Effective subscription levels and accurate TLP placement, which reflect underlying hardware structures (e.g., sockets, NUMA nodes, and chiplets), can significantly amplify the benefits of combined TLP and DLP. We verify runtime binding using lightweight tools such as `xthi` and inspect PCIe fabric and GPU links to ensure correct process-to-GPU affinity. System-level awareness is an interplay among the scheduler, software stack, and problem configuration, rather than solely an internal software property. Source code must be cognizant of external directives and avoid conflicts.

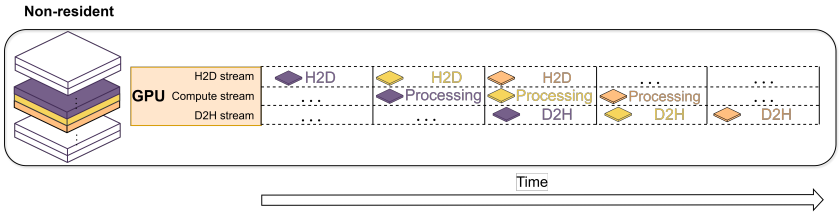


Figure 2.4: Hiding I/O latencies when data do not reside on GPU. Each domain slab undergoes three pipeline stages (upload, compute, and download), each carried out by a task-specific CUDA stream.

Data layout and object-disoriented programming

The memory subsystem is central to HPC due to its deep parallelism and the cross-abstraction impact of data layout decisions. Skepticism regarding object-oriented programming (OOP) in HPC stems from several compounding factors. Fine-grained encapsulation often impedes DLP. While array of structures (AoS) hinders SIMD instruction utilization, structure of arrays (SoA) facilitates it by exposing homogeneous data streams. We employ AoS judiciously, only when it offers clear locality benefits and the computation disallows DLP, frequently preferring composite layouts like AoSoA.

OOP's separation of state management from execution tends to break the sequential nature of how computational flows are expressed with temporal coupling exemplifying this issue, and in turn may disorient compilers. This often leads to instruction streams with excessive data-dependent control flow, increasing instruction count and diminishing instructions per cycle.

Metaprogramming, particularly template recursion, substitutes control flow for code replication, a practice that, among other issues, negates the benefits of loop stream detectors (LSD). In conjunction with header-only libraries, this typically results in a few large translation units and protracted rebuild times. Template diagnostics remain notoriously opaque. Instead of accepting the C++ standard's ok preprocessor, we advocate for true code synthesis that generates clear and concise kernel sources, thereby simplifying execution paths for both human comprehension and compilers.

Sharing data

MPI-only software is often memory-consuming. For intranode communication, we favor interprocess communication via shared memory over MPI extensions, given their historical brittleness and slow performance across platforms and versions.

Two primary approaches are considered: POSIX and System V shared memory. On Cray/HPE systems, `xpmem` is preferred, though not universally available. POSIX shared memory, typically backed by `tmpfs` at `/dev/shm`, is usually limited to 50% of RAM. System V shared memory circumvents this cap but imposes per-segment limits, necessitating chunking for multi-GB regions. Both methods involve reserving address space and allocating once at startup, publishing names or keys, and allowing processes to attach. Data sharing among MPI processes not only enhances memory efficiency but also eliminates redundant `MPI_Send` and `MPI_Recv` calls.

The argument that MPI processes with shared memory should be threads overlooks the selective sharing of simulation degrees of freedom, while maintaining private memory allocations that better reflect hardware architecture. This approach largely mitigates false sharing, cache pollution, and memory inconsistencies. Furthermore, processes, unlike threads, are first-class citizens in job schedulers.

2.2.2. Latency

Higher throughput mandates larger workloads, in turn extending latencies. However, a few long latencies are preferred over numerous shorter ones, as the former are more amenable to explicit latency-hiding schemes via double buffering. Despite repeated proposals, such techniques lack widespread adoption. Here, we rediscover two GPU latency-hiding techniques. The first addresses non-resident degrees of freedom, concealing off-chip I/O with three streams per device: one for host-to-device uploads, one for computation, and one for device-to-host downloads. Workloads are finely sliced (see Figure 2.4). The runtime maps transfer streams to DMA engines, with overlap verified on profiler timelines. This method requires page-locked host buffers, is inspired by dated software pipelining techniques, and pairs effectively with MPI plus interprocess communication via shared memory.

The second technique assumes that degrees of freedom are GPU-resident. GPU-aware MPI looks attractive but has been brittle across drivers and MPI stacks and tends to hide complexity. We prefer a TLP-assisted scheme. We split the grid into three launches: two halo slabs and an inner region that touches only on device data. The inner kernel runs while the upload stream moves the next halos and the download stream drains results from the previous step (Figure 2.5). This approach has outperformed a blocking GPU-aware MPI path in our tests. The achievable fraction of peak network bandwidth by GPU-aware MPI remains unclear. As shown in Figure 2.1a, throughput depends on TLP-assisted communication, and TLP subscription level control appears implementation-specific. While unified virtual memory (UVM) initially seems attractive for CPU-to-GPU transfers, its

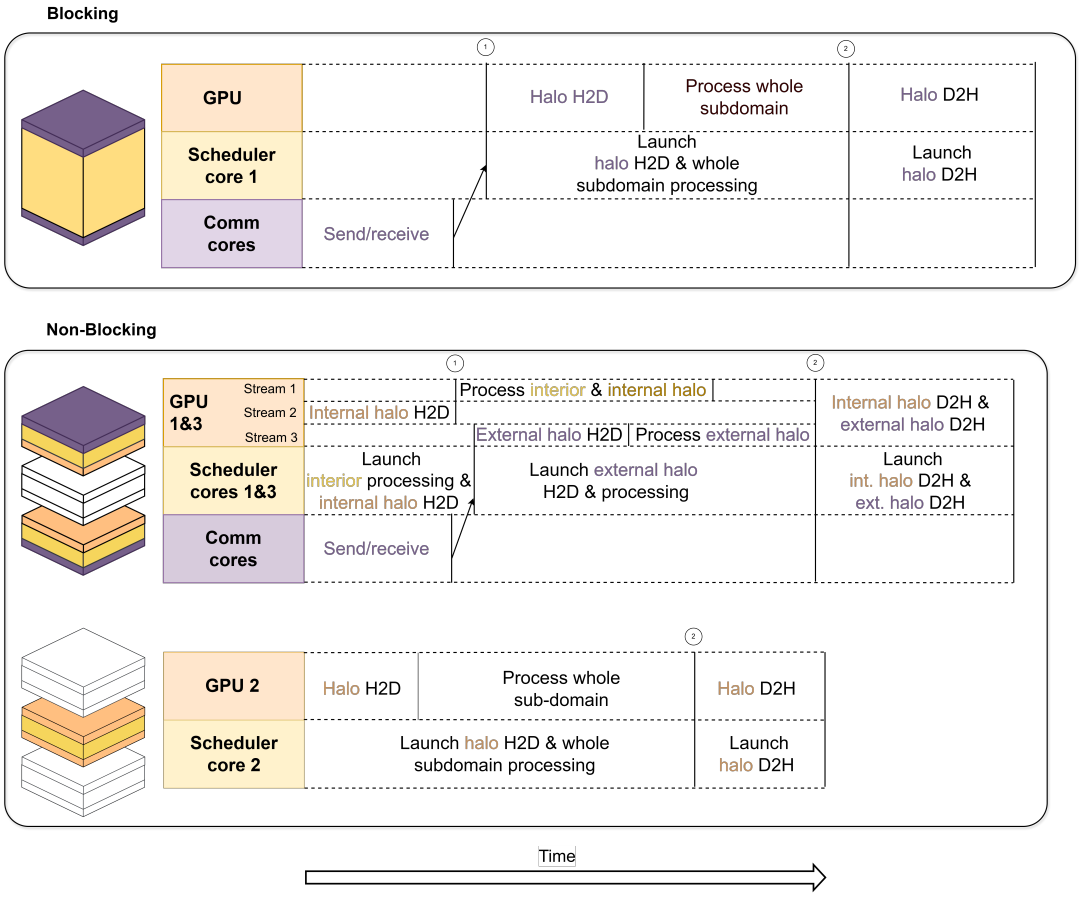


Figure 2.5: Sequence diagram of the computation–communication overlap compared to a non-blocking approach.

page-fault-driven on-demand migration introduces fine-grained, difficult-to-hide latencies.

Alternatively, for governing equations exhibiting locality, we employ overlapping subdomains, an aggressive technique suppressing communication latency. This involves domain decomposition with overlapping adjacent subdomains, creating thicker halos. Exchanges are deferred during substeps, gradually corrupting boundary data. Before corruption reaches the interior, halos are exchanged to restore consistency. Thus, latency is traded for redundant computation and enhanced effective bandwidth through larger payloads.

Certain irregular computational schemes encounter the worst-case scenario of numerous short latencies, preventing close-to-peak data rates. The strategy there focuses on improving fine-grained spatiotemporal locality, as it directly impacts the memory subsystem, particularly first-level data caches. This strategy, implemented by data- and computation-reordering techniques including kernel fusion, is beyond the scope of this discussion.

3 | Numerical Schemes

3.1. Governing equations

The fields of a compressible flow problem can be represented by $\rho, \boldsymbol{u}, p, E, T$, namely its density, speed vector, pressure, energy and absolute temperature. The evolution of the flow field is governed by the Euler equations:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{3.1a}$$

$$\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u}^T + p\mathbb{I}) = \boldsymbol{0} \tag{3.1b}$$

$$\frac{\partial (\rho E)}{\partial t} + \nabla \cdot ((\rho E + p)\boldsymbol{u}) = 0. \tag{3.1c}$$

In the present work, the above system of equations is closed by the equation of state:

$$p = \rho R T \tag{3.2}$$

where R is the gas constant. This system can be expressed in flux-divergence form as:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}}{\partial x} + \frac{\partial \boldsymbol{G}}{\partial y} + \frac{\partial \boldsymbol{H}}{\partial z} = \boldsymbol{0} \tag{3.3}$$

where $\boldsymbol{U}(\boldsymbol{x}, t) = (\rho, \rho\boldsymbol{u}, \rho E)^T$ is the vector of conservative variables and with initial condition $\boldsymbol{q}(\boldsymbol{x}, t) = \boldsymbol{q}_0(\boldsymbol{x}, 0)$, and $\boldsymbol{F}, \boldsymbol{G}, \boldsymbol{H}$ are flux functions:

$$\boldsymbol{F} = \begin{pmatrix} \rho u_x \\ \rho u_x^2 + p \\ \rho u_y u_x \\ \rho u_z u_x \\ (E+p)u_x \end{pmatrix}, \boldsymbol{G} = \begin{pmatrix} \rho u_y \\ \rho u_x u_y \\ \rho u_y^2 + p \\ \rho u_z u_y \\ (E+p)u_y \end{pmatrix}, \boldsymbol{H} = \begin{pmatrix} \rho u_z \\ \rho u_x u_z \\ \rho u_y u_z \\ \rho u_z^2 + p \\ (E+p)u_z \end{pmatrix} \tag{3.4}$$

Equation 3.3 is semi-discretized according to the Finite Volume Method, employing a

structured Cartesian 3D grid, and it can be reduced to a system of Ordinary Differential Equations:

$$\frac{d\boldsymbol{U}(t)}{dt} = L(\boldsymbol{U}(t)) \tag{3.5}$$

where $\boldsymbol{U}(t)$ is a representation of cell-averaged $\boldsymbol{U}$ variables. Time is uniformly discretized with (adaptive) time step Δt_n, so that $t = \sum_n \Delta t_n$. Solution of (3.5) is advanced in time according to a low-storage third-order accurate *Total Variation Diminishing Runge-Kutta* method, as follows:

$$\boldsymbol{w}_j = a_j \boldsymbol{w}_{j-1} + \Delta t_n \boldsymbol{L}(\boldsymbol{y}_{j-1}) \tag{3.6a}$$

$$\boldsymbol{y}_j = \boldsymbol{y}_{j-1} + b_j \boldsymbol{w}_j \tag{3.6b}$$

for $j = 1, 2, 3$, $\boldsymbol{w}_0 = 0$, $\boldsymbol{y}_0 = \boldsymbol{U}^n$, $a_j = \{0, -\frac{17}{32}, \frac{32}{27}\}$, $b_j = \{\frac{1}{4}, \frac{8}{9}, \frac{3}{4}\}$, and $\boldsymbol{U}^{n+1} = y_3$.

For each time-step it is necessary to evaluate three times $\boldsymbol{L}(\boldsymbol{U}(t))$, which is the most computationally expensive aspect of the integration process. Its computation can be split along the three spatial dimensions, as follows:

$$\boldsymbol{L} = \boldsymbol{L}_x + \boldsymbol{L}_y + \boldsymbol{L}_z \tag{3.7}$$

The contribution of each cell i, j, k to L_x, at time-step n is:

$$L^{x,n}_{i,j,k} = -\frac{1}{h}\left(F^n_{i+1/2,j,k} - F^n_{i-1/2,j,k}\right) \tag{3.8}$$

where $F^n_{i+1/2,j,k}$ is the numerical approximation of intercell flux $\boldsymbol{F}$ at face location $i+1/2$, and h is the uniform grid spacing. Computing numerical fluxes requires solving a Riemann problem at each cell interface, for which we employ the HLLC approximated solver [42]. Therefore, $F^n_{i+1/2,j,k}$ can be computed as:

$$\begin{aligned} F^n_{i+1/2,j,k} = &\frac{1+\text{sign}(s^*)}{2}(\boldsymbol{F}(\boldsymbol{U}^*_L) + s^-(\boldsymbol{U}^*_L - \boldsymbol{U}_L)) \\ &+\frac{1-\text{sign}(s^*)}{2}(\boldsymbol{F}(\boldsymbol{U}^*_R) + s^+(\boldsymbol{U}^*_R - \boldsymbol{U}_R)) \end{aligned} \tag{3.9}$$

where

$$s^- = \min(s_L, 0),\ s^+ = \max(0, s_R),\ \text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Wave speed estimates s_L and s_R, for the lowest and fastest moving waves, respectively, are obtained from Einfeldt [14]. The wave speed estimate for the contact wave is computed

by following Batten [2],

$$s^* = \frac{\rho_R u_R(s_R - u_R) - \rho_L u_L(s_L - u_L) + p_L - p_R}{\rho_R(s_R - u_R) - \rho_L(s_L - u_L)}. \tag{3.10}$$

Approximation of the intermediate states U_L^*, U_R^* are computed according to [42]. Initial states U_L, U_R are obtained from a fifth-order WENO reconstruction. In order to avoid non-physical results near shocks, we do not directly reconstruct conserved quantities. Instead, we reconstruct a projection of the corresponding primitive variables $\boldsymbol{W} = (T, \boldsymbol{u}, p)$. In particular, Equation 3.3 can be rewritten as:

$$\frac{\partial \boldsymbol{W}}{\partial t} + \boldsymbol{A}\frac{\partial \boldsymbol{W}}{\partial x} + \boldsymbol{B}\frac{\partial \boldsymbol{W}}{\partial y} + \boldsymbol{C}\frac{\partial \boldsymbol{W}}{\partial z} = 0, \tag{3.11}$$

where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ are the Jacobian of the corresponding fluxes with respect to $\boldsymbol{W}$. While it would be possible to apply the WENO reconstruction directly on each primitive variable, such an approach is suitable only when the order of accuracy is low (second or third order). We note that $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ are diagonalizable (as the system of ODEs is hyperbolic), and thus have a standard closed-form formula $\boldsymbol{A} = \boldsymbol{S}_A \Lambda_A \boldsymbol{S}_A^{-1}$, and similarly for $\boldsymbol{B}$ and $\boldsymbol{C}$. Such a decomposition defines a characteristic space to which the primitive variables can be mapped to. As a result, the original problem is decomposed into five uncoupled advection problems. Then, WENO reconstruction is applied on each characteristic variable. Finally, reconstructed variables are projected back to the original space. More details regarding the employed numerical schemes can be found in [8].
The numerical schemes just described lead to a second-order convergence rate, for smooth flow fields. In fact, while a fifth-order WENO reconstruction scheme is employed, two other choices fundamentally limit the achieved convergence order to 2. First, fluxes are cell-face averaged using a one-point quadrature rule, with a second-order convergence rate. Second, while we evolve in time primitive variables $\boldsymbol{W}$, fluxes computation requires their conserved counterparts, which are computed based on Equation 3.2. However, such conversion reduces the convergence order to two.

3.2. Boundary conditions

LVR supports a wide range of boundary conditions, including periodic boundary conditions, reflecting walls, and non-reflecting outflow. In this section their mathematical details are presented. As LVR supports Cartesian grids only, in the rest of this chapter a grid with shape $L_x \times L_y \times L_z$ is assumed. It is worth noticing that assuming a Carte-

sian grid allows making considerable simplifications in the treatment of such boundary conditions.

3.2.1. Periodic boundary conditions

Imposing periodic boundary conditions on a direction of the domain means enforcing:

$$\boldsymbol{U}(\boldsymbol{x}) = \boldsymbol{U}(\boldsymbol{x} + L\boldsymbol{e}),$$

where L is the length of the chosen direction (either L_x, L_y or L_z), and $\boldsymbol{e}$ is the unit vector normal to the periodic boundary.

3.2.2. Reflecting wall boundary conditions

Imposing reflecting wall boundary conditions on a wall of the domain means enforcing zero normal mass and normal energy flues on that boundary. Therefore, it must hold that:

$$\rho u_n = 0$$

$$(E + p)u_n = 0$$

where u_n is the component of the velocity vector normal to the boundary. When such boundary conditions are implemented by means of ghost cells (as it is done in LVR), they translate to enforcing that fluid fields on the ghost cells mirror the interior of the domain, with the sign of the velocity normal to the boundary flipped.

3.2.3. Characteristic boundary conditions (NSCBC)

In order to impose non-reflecting outflow boundary conditions, we rely on the concept of Navier-Stokes Characteristic boundary conditions (NSCBC). Even if this work is concerned with Euler equations (that is, the inviscid case), such a method remains well suited. In particular, as the method is based upon the concept of characteristic waves, the method itself is actually rigorous only for the inviscid case.

We start by performing a characteristic analysis of Equation 3.3 along one direction, for example the x one. Equation 3.3 can be recast as:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}}{\partial x} + \boldsymbol{C} = \boldsymbol{0}, \tag{3.12}$$

where

$$\boldsymbol{C} = \frac{\partial \boldsymbol{G}}{\partial y} + \frac{\partial \boldsymbol{H}}{\partial z}.$$

If we define matrix $\boldsymbol{A}$ so that its entries are:

$$a_{ij} = \frac{\partial \boldsymbol{F_i}}{\partial \boldsymbol{U}_j}$$

, Equation 3.12 can be rewritten as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{A}\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}} + \boldsymbol{C} = \boldsymbol{0}, \tag{3.13}$$

Given the hyperbolic nature of Equation 3.12, A is guaranteed to diagonalizable:

$$\boldsymbol{A} = \boldsymbol{S}\boldsymbol{\Lambda}\boldsymbol{S}^{-1} \tag{3.14}$$

where $\boldsymbol{S}$ is a matrix such that its columns are $\boldsymbol{A}$'s right eigenvectors, and $\boldsymbol{\Lambda}$ is the diagonal matrix whose non-zero entries are the corresponding eigenvalues. By multiplying Equation 3.13 by $\boldsymbol{S}^{-1}$ we get:

$$\boldsymbol{S}^{-1}\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{\Lambda}\boldsymbol{S}^{-1}\frac{\partial \boldsymbol{U}}{\partial x} + \boldsymbol{C} = \boldsymbol{0}. \tag{3.15}$$

Finally, by defining

$$\boldsymbol{\mathcal{L}} = \boldsymbol{\Lambda}\boldsymbol{S}^{-1}\frac{\partial \boldsymbol{U}}{\partial x} \tag{3.16}$$

we rewrite the original system as:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{S}\boldsymbol{\mathcal{L}} + \boldsymbol{C} = \boldsymbol{0}. \tag{3.17}$$

It is possible to see how the original system has been recast so that the hyperbolical terms along thr x direction have been modified, while the ones along the other directions have been left unchanged. In order to better understand the meaning of the $\boldsymbol{S}\boldsymbol{\Lambda}$ term we shall start by considering the meaning of the diagonalization of matrix $\boldsymbol{A}$. Any hyperbolic system can be recast as a system of independent ODEs, expressed in terms of *characteristic waves*, which correspond to the $\boldsymbol{S}^{-1}\frac{\partial \boldsymbol{U}}{\partial x}$ term. Therefore, each $\boldsymbol{A}$'s eigenvector is associated to one characteristic wave, which travels at a speed determined by the corresponding eigenvalue. In particular, in this case the eigenvalues λ_i are:

$$\lambda_1 = u - c$$

$$\lambda_2 = \lambda_3 = \lambda_4 = u$$

$$\lambda_5 = u + c$$

where c is the speed of sound:

$$c^2 = \frac{\lambda p}{.}\rho$$

The $\boldsymbol{\mathcal{L}}$s are:

$$\mathcal{L}_1 = \lambda_1 \left(\frac{\partial p}{\partial x} - \rho c \frac{\partial u}{\partial x} \right),$$

$$\mathcal{L}_2 = \lambda_2 \left(c^2 \frac{\partial \rho}{\partial x} - \frac{\partial p}{\partial x} \right),$$

$$\mathcal{L}_3 = \lambda_3 \frac{\partial v}{\partial y},$$

$$\mathcal{L}_4 = \lambda_3 \frac{\partial w}{\partial z},$$

$$\mathcal{L}_5 = \lambda_5 \left(\frac{\partial p}{\partial x} + \rho c \frac{\partial u}{\partial x} \right).$$

$\mathcal{L}_1$ and $\mathcal{L}_5$ are associated to the acoustic waves, travelling in the negative and positive directions, respectively, and transport pressure perturbations. L_2 refers to the entropy wave, while $L3$ and $L4$ refer to wave that transport tangential velocities perturbations. It is worth noticing that each $-\mathcal{L}$ is equal to time variation of the associated perturbation at any given location (including, the boundary).

For each of these wave, by considering its speed, it is possible to determine wether it is moving from the inside of the domain to the outside, or vice versa. In the first case, the associated $\mathcal{L}$ can be computed from the interior of the domain, and therefore does not require to be prescribed. On the other hand, in case of an incoming wave, the corresponding $\mathcal{L}$ can not be determined from the interior, and needs to be somehow prescribed.

At this point, it is possible to rewrite the system in Equation 3.12 in terms of $\mathcal{L}$s:

$$\frac{\partial \rho}{\partial t} + d_1 + C_1 = 0 \tag{3.18}$$

$$\frac{\partial \rho E}{\partial t} + \frac{1}{2}\left(u^2 + v^2 + w^2\right) d_1 + \frac{d2}{\gamma - 1} + \rho u d_3 + \rho v d_4 + \rho w d_5 + C_2 = 0 \tag{3.19}$$

$$\frac{\partial \rho u}{\partial t} + u d_1 + \rho d_3 + C_3 = 0 \tag{3.20}$$

$$\frac{\partial \rho v}{\partial t} + v d_1 + \rho d_4 + C_4 = 0 \tag{3.21}$$

$$\frac{\partial \rho w}{\partial t} + w d_1 + \rho d_5 + C_5 = 0 \tag{3.22}$$

$$\tag{3.23}$$

where C_is are the component of the $\boldsymbol{C}$ vector (which accounts for the transverse terms), and d_i are defined as:

$$d_1 = \frac{1}{c^2}\left(\mathcal{L}_2 + \frac{\mathcal{L}_5 + \mathcal{L}_1}{2}\right) = \frac{\partial \rho u}{\partial x} \tag{3.24}$$

$$d_2 = \frac{1}{2}\left(\mathcal{L}_5 + \mathcal{L}_1\right) = \frac{\partial c^2 \rho u}{\partial x} + (1-\gamma)\mu\frac{\partial p}{\partial x} \tag{3.25}$$

$$d_3 = \frac{1}{2\rho c}\left(\mathcal{L}_5 - \mathcal{L}_1\right) = u\frac{\partial u}{\partial x} + \frac{1}{\rho}\frac{\partial p}{\partial x} \tag{3.26}$$

$$d_4 = \mathcal{L}_3 = u\frac{\partial v}{\partial x} \tag{3.27}$$

$$d_5 = \mathcal{L}_4 = w\frac{\partial w}{\partial x} \tag{3.28}$$

This resulting system can now be used to advance the solution on the boundary. In order to compute $\mathcal{L}_i$s we rely on fourth-order accurate finite differences, while for C_i we employ the same numerical schemes used for the interior of the domain (WENO reconstruction + HLLC solver).

Non-Reflecting Outflow boundary conditions

Non-reflecting outflow boundary conditions are essential to model scenarios like the one (the shock tube) we simulate in this thesis. In particular, they allow shocks to leave the domain, without being reflected back. If we assume that such boundary conditions are applied on the left face of a Cartesian domain (see Figure 3.1), it follows that u must be negative. Therefore, four characteristic waves are leaving the domain ($\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4$), independently of u, and one ($\mathcal{L}_5$) may be leaving or entering the domain, depending on wether the outflow is supersonic ($u < -c$), or subsonic ($u > -c$). If the outflow

is supersonic, no incoming characteristic waves exist, and so all $\mathcal{L}_i$s can be computed from the interior of the domain. If the outflow is subsonic, $\mathcal{L}_5$ must be prescribed. As mentioned above, that wave is associated with pressure perturbations on the boundary. If we set $\mathcal{L}_5$ to 0, no wave would cross the boundary, resulting in perfectly non-reflecting boundary conditions. However, this way, the problem may result to be ill-posed. A different approach would be to impose a far-field pressure p_∞, meaning that, after all waves have left the domain, the outlet pressure is effectively p_∞, while in the transient period it is allowed to assume different values. This is analogous to having the domain enclosed in a larger environment, whose static pressure is p_∞. However, such a method would involve solving the governing equations on a much larger domain (the original domain plus the environment), resulting in a potentially computationally expensive problem. Instead, it is possible to act on $\mathcal{L}_5$ to gently drive the outlet pressure towards p_∞. In the literature, it is suggested to adopt the following relaxation formula:

$$\mathcal{L}_5 = K(p - p_\infty), \tag{3.29}$$

where p is the current outlet pressure, and K is defined as $\sigma\,(1 - \mathcal{M}^2)\,\frac{c}{L}$, with σ a constant, $\mathcal{M}$ the Mach number, c the speed of sound, and L the domain characteristic length. As suggested in [38] we set σ to 0.25.

For simplicity reasons, we rely on the above outflow boundary conditions even when $u > 0$, that is, when the outflow is effectively an inflow. We found that such a choice does not compromise the simulation numerical stability. However, to handle such a case, a modification to Equation 3.29 was needed. In fact, it is required that K is strictly greater than zero. Otherwise, the pressure would result to be driven to values far from p_∞. However, in case $u > c$ (which, technically, is still considered to be a subsonic outflow), the $1 - \mathcal{M}^2$ factor would result to be negative, making K negative as well. Therefore, we modify Equation 3.29 as follows:

$$\mathcal{L}_5 = \sigma\ \min\left(|1 - \mathcal{M}^2|, 1\right) \frac{c}{L}(p - p_\infty). \tag{3.30}$$

The minimum function is needed to ensure that the $|1 - \mathcal{M}^2|$ term remains lower than 1, as it happens when $u < 0$.

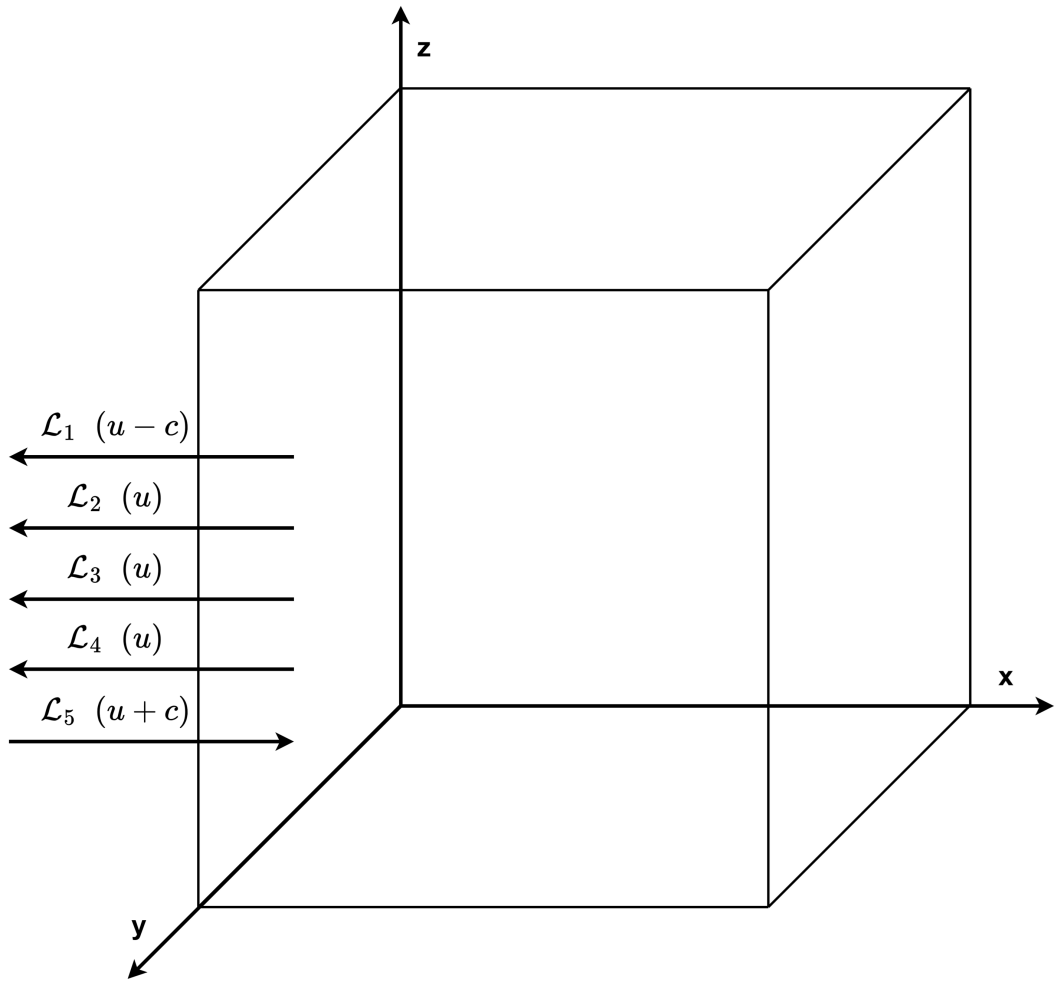


Figure 3.1: Characteristic waves direction and their speed in the case of a subsonic outflow.

4 | Solver structure

In this chapter a detailed description of LVR is provided

4.1. Data structures

LVR data structures consist mainly in linearized arrays, organized in a Structure of Arrays (SoA) fashion. The main quantities that need to be stored are the following.

Degrees of freedom. In LVR, per each cell, five degrees of freedom exist: the temperature, the pressure and the three velocity components. In order to support boundary conditions, we rely on 3 layers of ghost cells on each boundary of the inner domain. This justifies why for a $N_x \times N_y \times N_z$ domain, $(N_x+6) \times (N_y+6) \times (N_z+6)$ cells are allocated.

Convective fluxes and right-hand sides. The number of fluxes to be computed along each direction, depends on the direction itself. In fact, while fluxes should be computed at each cell interface, degrees of freedom on ghost cells are not evolved relying on convective fluxes. Therefore, along the x direction, $(N_x + 1) \times N_y \times N_z$ x-fluxes are needed, and similarly along the y and y directions. In addition to fluxes, also Equation 3.5 right-hand sides must be stored. Since such ODEs system must be solved for each cell of the inner domain, 5 arrays with $N_x \times N_y \times N_z$ are needed.

NSCBC fluxes and right-hand sides. For characteristics boundary conditions, additional data structures are needed. For each degree of freedom on the first ghost layer (excluding the corners), a NSCBC flux and right-hand side are needed.

The choice of the SoA layout lies in the fact that it enables auto-vectorization by compilers. When the somewhat more flexible Array of Structures (AoS) layout is instead used, compilers may struggle to exploit the available SIMD units.

4.2. Kernels

LVR main computational kernels are the following ones.

FLUX The FLUX kernel implements the flux computation. As explained in chapter 3 the fluxes computation can be logically divided into four steps: 1) primitive variables projection into characteristic space, 2) reconstruction of the characteristic variables through the WENO scheme, 3) back-projection of the characteristic variables into the original space, 4) numerical fluxes computation using the HLLC solver. All these stages are implemented in an unified kernel, as explained in Equation 7.1. The same kernel is reused to perform the flux computation along all three directions, by just changing the order of the velocity components of the inputs. For instance, if FLUX(P, T, V_x, V_y, V_z) computes x-fluxes, then FLUX(P, T, V_y, V_x, V_z) computes y-fluxes

DUC After fluxes have been obtained, it is necessary to compute their divergence, and then to apply the Runge-Kutta scheme described in chapter 3. However, it must be noticed that, fluxes are computed with respect to conserved variables, which are different from the stored degrees of freedom. Therefore, while right-hand sides can be stored in conserved form, the updated degrees of freedom (which are initially obtained in conserved form) must be converted to primitive form. Also in this case, following the considerations made in Equation 7.1, all these three stages are implemented in a single kernel.

DT At each time step, the time step length Δ_t is adaptively chosen. This involves determining the fastest moving wave in the domain. To this end, it is necessary to perform a reduction over all the degrees of freedom. Among the three, this is the least compute intensive one.

The simplified LVR algorithm is reported in the following pseudocode.

for all $ti \leq TCMAX$ **do**
  $\Delta t \leftarrow DT(P, T, V_x, V_y, V_z)$
  for all $k < 3$ **do**
    $F_{x,P}, F_{x,T}, F_{x,V_x}, F_{x,V_y}, F_{x,V_z} \leftarrow FLUX(P, T, V_x, V_y, V_z)$
    $F_{y,P}, F_{y,T}, F_{y,V_y}, F_{y,V_x}, F_{y,V_z} \leftarrow FLUX(P, T, V_y, V_x, V_z)$
    $F_{x,P}, F_{x,T}, F_{x,V_z}, F_{x,V_y}, F_{x,V_x} \leftarrow FLUX(P, T, V_z, V_y, V_x)$
    $\boldsymbol{R} \leftarrow R_P, R_T, R_{V_x}, R_{V_y}, R_{V_z}$
    $\boldsymbol{F}_x \leftarrow F_{x,P}, F_{x,T}, F_{x,V_x}, F_{x,V_y}, F_{x,V_z}$
    $\boldsymbol{F}_y \leftarrow F_{y,P}, F_{y,T}, F_{y,V_x}, F_{y,V_y}, F_{y,V_z}$
    $\boldsymbol{F}_z \leftarrow F_{z,P}, F_{z,T}, F_{z,V_x}, F_{z,V_y}, F_{z,V_z}$

$$P, T, V_x, V_y, V_z, \boldsymbol{R} \leftarrow DUC(\Delta t, P, T, V_x, V_y, V_z, \boldsymbol{R}, \boldsymbol{F}_x, \boldsymbol{F}_y, \boldsymbol{F}_z)$$

end for

end for

4.3. Plugin architecture

LVR is designed as a modular system composed of dynamically loaded plugins. Each plugin is built as a shared object that the runtime loads based on properly configured environment variables. A lightweight driver then invokes the plugin's exposed functions.

The plugin model provides benefits analogous to Object-Oriented Programming: encapsulation (each plugin is an isolated compilation unit), inheritance (plugins can load and build upon other plugins), and polymorphism (the driver makes no compile-time assumptions about plugin interfaces).

In LVR, the plugin architecture is leveraged for several purposes.

Horizontal decoupling. Plugins are used to decouple the implementation of LVR's components, whose respective logics operate independently. For example, the simulation logic is delegated to specialized plugins that encapsulate all required platform-specific code. In contrast, tasks such as data dumping or applying initial conditions—both independent of the simulation logic—are handled by separate platform-agnostic plugins. In particular, managing data dumping through a plugin allows saving to disk only the necessary data, thus reducing the burden on the file system, and the I/O time.

This design allows components that are not immediately required to be postponed during development. The high degree of modularity also promotes code reuse, as plugins can be developed without being tied to a specific solver. Additionally, at compile time, only the necessary plugins must be built, while others (e.g., those targeting different platforms) can be safely excluded. Moreover, given the neat separation into independent plugins, LVR enjoys fast compilation times: each plugin constitutes an independent compilation units, allowing Thread-Level parallelism to be leveraged during the building process.

Bisection. The plugin architecture also simplifies debugging. When a new, potentially unstable plugin is being tested, previously validated plugins can serve as reliable fallbacks. By replacing suspected plugins with correct ones, it is possible to quickly identifying the faulty ones, with a simple trial-and-error process.

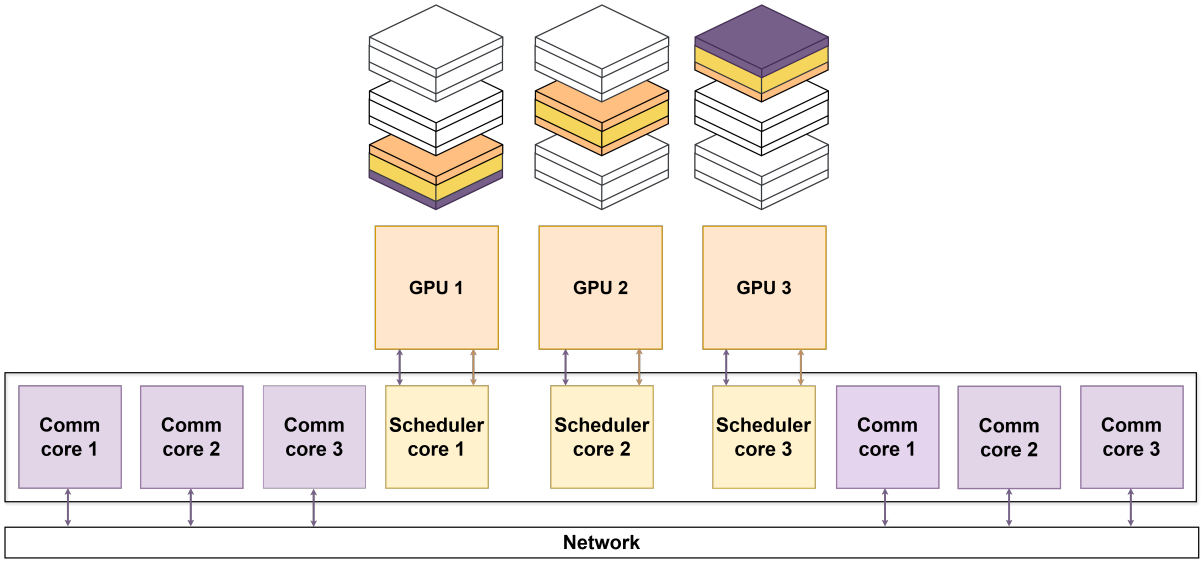


Figure 4.1: Core specialization is essential for maximizing DRAM and network bandwidth at the node level.

4.4. Parallelization strategies

In order to leverage high-performance computing infrastructure, several parallelization strategies have been devised for LVR. In the rest of this section, they are described in details.

4.4.1. OpenMP

OpenMP is used to take advantage of multi-threading capabilities within a single compute node. We divide the workload along the z axis, that is, each thread is assigned a slab of the computational domain, as depicted in Figure 5.2a, and takes care of all associated computation. Threads need to be synchronized before and after the DUC and the DT kernel.

4.4.2. Leveraging the exascale building blocks

When GPUs are the target platforms, a different parallelization strategy is warranted. In particular, we adopt a much finer parallelism granularity. We launch 1D grids composed of 1D threads blocks. The grid size is chosen according to the domain size, while thread blocks' size is fixed, and can be set by the user. Each thread takes care of processing a single cell, for all the LVR's kernels. Except for the DT kernel, no inter-thread cooperation is required. For what concerns CPU-GPU data transfers, they happen only when strictly

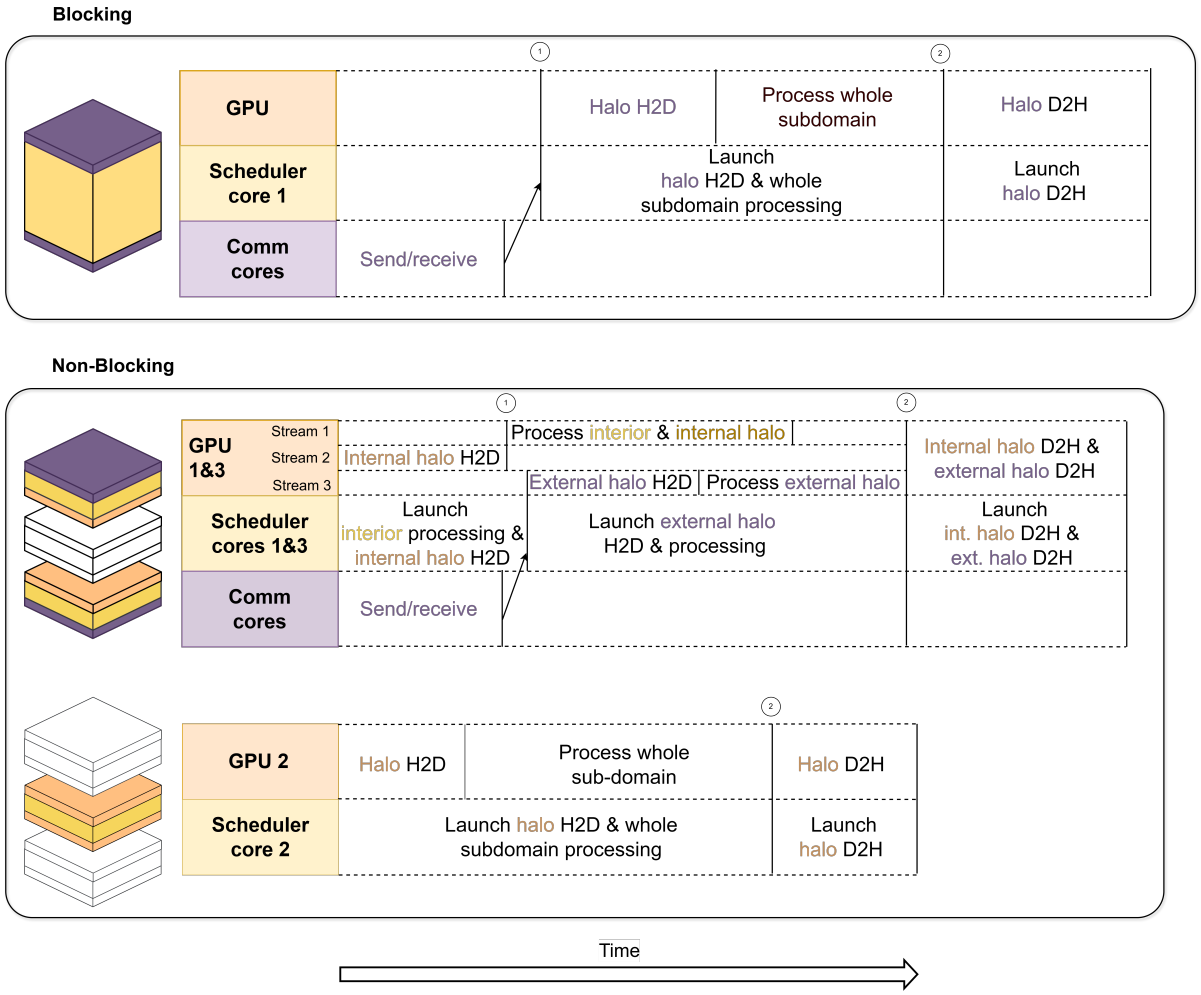


Figure 4.2: Sequence diagram of the computation–communication overlap compared to a blocking approach.

needed. At the begin of the simulation, degrees of freedom are initialized as required by the chosen initial conditions. Then, they are copied to device memory, after which the computation can start. Throughout the computation, only the top and bottom ghost layers needs to be copied from the CPU to the GPU, which come from the MPI data exchange. Symmetrically, the top and bottom outer regions of the inner domain need to be copied back to the CPU, so that MPI data exchange can take place. Degrees of freedom on the GPU are fully copied to CPU only either at the end of the simulation, or when a data dump to file is required. It is worth noticing that CUDA/OpenCL parallelization in LVR can only be coupled with MPI parallelization.

4.4.3. Distributed workload on exascale systems

Performance modelling results discussed in subsection 5.2.1 show that the slab domain decomposition scheme leads to negligible network timescales, in weak scaling regimes, that is, when each node is assigned a substantial amount of work. On the contrary, in strong-scaling regimes, its network time scale degrades significantly. This underlines how communication over the network may represent a bottleneck. In order to reduce its impact, we devised a strategy to overlap, when targeting GPU platforms, on-device computation with communication over the network and, additionally, over GPU-CPU interconnects. In LVR, each device is assigned a domain slab and, as explained in subsection 5.2.1, data transfer must happen to preserve consistency across all subdomains. In a non-overlapped communication scheme, at each iteration, each device has to wait for incoming data to be received and outgoing ones to be delivered. However, this way, the TTS is effectively bound by the sum of the computation time and network time. To alleviate this issue, one can notice that each subdomain can be divided into three regions: an interior and two halos, as depicted in Figure 4.1, and in Figure 4.2. Only the computation of the latter depends on incoming data, while the interior computation can proceed without any delay. Therefore, network communication can be overlapped with interior computation, while halos computation can take place immediately after. To ensure consistency, interior computation results must be written out-of-place, so that halos computation does not depend on data of mixed iterations. Such considerations have been followed for the LVR cluster-level parallelization strategy. Cores are split into two groups: compute cores and communication cores. Compute cores are assigned to a GPU device, and issue all data transfers and grid launches. Communication cores, instead, are responsible for performing all necessary MPI calls for inter-node communication. We divide the communication workload among several cores because, as shown in Figure 2.1a, the peak network bandwidth can only be achieved when several cores are utilized. For intra-

node communication (between compute and communication cores, and between compute cores), we rely on inter-process communication via shared memory.

5 | Performance modelling

5.1. Time Scales Model

In the development of LVR, we aimed at taking performance-oriented design decisions, without iterating through several tentative versions before satisfactory performance was achieved. Instead, we wanted to model our solver's performance ahead of time, even before an instruction stream existed. In order to do so, we developed the *Time Scales Model* performance model. Its main objective is not to offer accurate performance predictions, but rather to provide informative insights regarding the LVR's bottlenecks and performance upper-bounds. The model considers the main contributors to LVR's total execution time:

- floating-point operations,
- memory transfers,
- network transfers,
- CPU-GPU transfers.

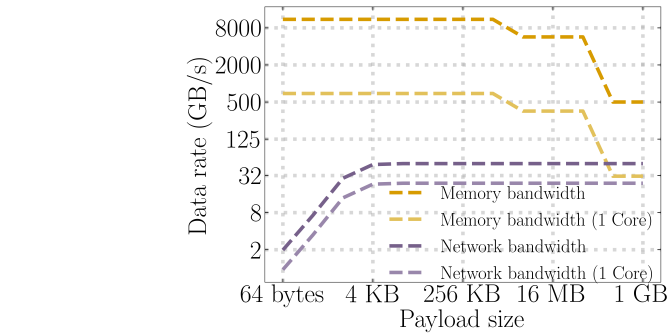
We estimate the time spent by the solver on each of them, as follows:

$$t_{\text{compute}} = \frac{\text{FLOP count}}{\text{Peak Compute Performance}}, \tag{5.1}$$

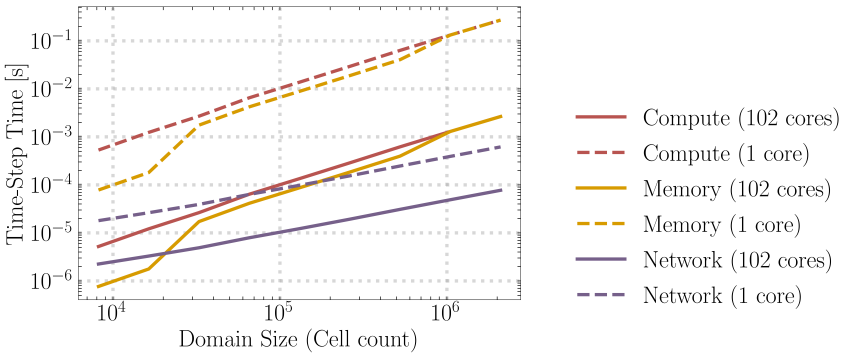
$$t_{\text{transfer}} = \frac{\text{Payload size}}{\text{Rate(Payload size)}} \tag{5.2}$$

Here, t_{compute} represents the time spent on floating-point computations, while t_{transfer} corresponds to the time required for a given data transfer type. We compute one transfer time per transfer class listed above.

The FLOP count denotes the total number of floating-point operations required by the algorithm. *Peak Compute Performance* is the FP64 theoretical peak of the target platform. Payload size is the total volume of data moved through memory, the network, or the CPU–GPU interconnect (e.g., PCIe). Finally, Rate refers to the effective bandwidth



(a) Rates



(b) Time scales

Figure 5.1: Synthetic micro-benchmarks are used to profile data rates (left). The curves are absorbed in the symbolic framework. By accessing the coarse-grained view of the abstraction spectrum, the symbolic framework allows us to have accurate performance modelling in terms of the timescales involved in the simulation (right).

for that transfer, expressed as a function of the payload size.

Estimating each of these quantities requires its own methodology, which we detail next.

Peak Compute Performance. This quantity can be computed theoretically by accounting for the hardware characteristics of the target platform. For CPUs, a commonly used expression is:

$$\text{P. Perf.} = \text{Clock} \cdot \text{Core count} \cdot \text{SIMD width} \cdot \text{FMA units per core} \cdot 2,$$

where the clock frequency is the expected operating frequency of the CPU[1], the core count refers to the number of *physical* cores, the SIMD width denotes the number of double-precision values processed per vector instruction, the number of FMA units indicates how many fused multiply–add units each core contains, and the final factor of 2 accounts for the two floating-point operations performed by each FMA per cycle.

For GPUs, the architectural diversity across vendors and generations prevents the use of an equally simple expression.

Nevertheless, under ideal conditions—namely instruction streams rich in dependency-free FMA operations—performance approaching the theoretical peak can be achieved. Consequently, our model relies on theoretical peak values and does not require empirical benchmarking to obtain them.

Rates. Estimating data transfer rates is challenging for two main reasons. First, theoretical bandwidth values are rarely attained in practice, even under ideal conditions. For example, sustained memory bandwidth often reaches no more than about 80% of the theoretical peak. Second, as noted earlier, effective bandwidth strongly depends on the payload size, as illustrated in Figure 5.1a.

Memory bandwidth generally decreases as the payload size increases. This behavior is largely due to the presence of caches: small payloads may fit entirely within cache levels, yielding higher observed bandwidth, whereas larger transfers increasingly access slower levels of the memory hierarchy.

In contrast, network and PCI Express bandwidths exhibit the opposite trend. Their effective throughput improves with larger payload sizes because fixed latency and protocol overhead dominate small transfers but become amortized for larger payloads.

[1]When multiple cores are active simultaneously, the sustained frequency is often lower than the advertised peak. In such cases, the base frequency typically provides a more realistic estimate.

To account for these effects, we determine data rates through micro benchmarks, small programs designed to measure specific transfer behaviours in isolation. While micro-benchmarks may produce optimistic results compared to full applications, where many hardware components interact in nontrivial ways, they nonetheless provide meaningful upper bounds suitable for our performance model. Moreover, by varying the payload sizes used in these benchmarks, we obtain rate estimates across a broad range of transfer sizes.

FLOP count. For the numerical schemes used in LVR (in particular, Equation 3.9, for fluxes' computation), manually counting FLOPs is both tedious and prone to mistakes. To avoid this, we rely on an ad-hoc symbolic framework built using SymPy [27], which allows us to express the numerical schemes directly as mathematical expressions. At this level of abstraction, FLOP counting can be fully automated, yielding accurate and efficient estimates.

Because not all floating-point operations have the same throughput, each operation is assigned a weight: 1 for addition/multiplications, 8 for divisions and 12 for square roots. Once the FLOP count of each kernel is computed, the solver's total FLOP count is derived from the problem size, i.e., the dimensions of the computational domain.

Payload sizes. In LVR, payload sizes can be derived directly from the domain size, so no symbolic tool is required. For memory traffic, we assume a write-back, write-allocate cache model. Under this model: 1) for each kernel, every input value is fetched from main memory only once, with all subsequent accesses served from cache; and 2) each write operation incurs an additional read, meaning that the number of written bytes must be counted twice when computing the payload size.

Given the complexities involved in this performance model—such as relating the domain size to the corresponding payload sizes and, in turn, to the appropriate bandwidths—the computation of each of the aforementioned times (which we refer to as the problem's *Time Scales*) is non-trivial and cannot be reliably performed by hand. Furthermore, because the time scales depend on the problem size, effective visualization is also essential.

For these reasons, we developed a lightweight framework that automates these tasks, greatly simplifying the application of the performance model.
Once the time scales have been computed, interpreting them is straightforward. The largest time scale ($t_{\max}$) represents the dominant performance bottleneck. Consequently, the Time To Solution (TTS) cannot be smaller than this value.

Such time scale can in principle achieve its peak performance. Instead, for any other time scale t_i, its maximum achievable fraction of peak performance f_i can be computed as :

$$f_i = \frac{t_i}{t_{\max}}, \tag{5.3}$$

This can be understood as follows: assuming peak performance is continuously delivered for t_i, when averaged over the entire TTS, its contribution would reduce to $\frac{t_i}{TTS}$. Remembering $t_{max} < TTS$, this reasoning leads to Equation 5.3.

By identifying both the principal limiting factor and the maximum attainable fraction of peak for each contributing component, the Time Scales Model (TSM) can be viewed as a generalization of the Roofline model. In particular, TSM incorporates performance aspects beyond compute and memory throughput alone and explicitly accounts for the dependence of data rates on the payload size. An example of TSM in action is shown in Figure 5.1b. LVR's compute, memory and network time scales (with respect to the Intel Sapphire Rapid CPU) are reported, referred to a single time-step. Time scales are shown in two scenarios: 1) 1 core, 102 cores. In the case with 102 cores, it can be seen how LVR is compute bound from $3 \cdot 10^4$ up to $3 \cdot 10^4$ cells. Afterward, compute and memory time scales are nearly identical. This is because, as the domain size increases, it does not fit into the cache, resulting in a lower bandwidth, and in turn a higher memory time scale. LVR is predicted to be network bound only for domains smaller than $3 \cdot 10^3$ cells. Such an analysis would not have been entirely possible with the Roofline Model, as it fails to capture the dependency of the memory bandwidth on the footprint, and also does not account for the network.

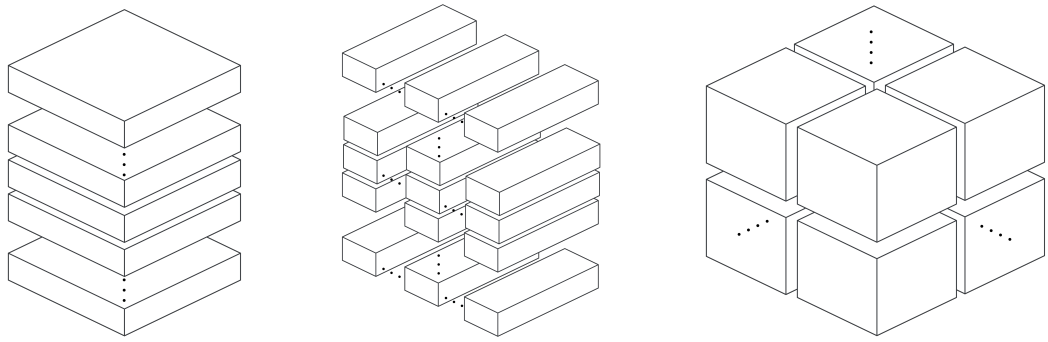
5.2. Performance model-driven development

For the development of our compressible flow solver LVR, the TSM has been extensively used to take design decisions.

In the next sections, we use our tools for calculating FLOP count and memory traffic from the numerical scheme, kernel variants and chosen layouts.

5.2.1. Domain Decomposition

In LVR, domain decomposition is leveraged for enabling simulations on multi-node systems. Here, the domain is subdivided among nodes to split the computational burden. At the interface between subdomains, data is shared to ensure a globally consistent pro-



(a) (b) (c)

Figure 5.2: *Slabs* domain decomposition (left), *Pencil* domain decomposition (centre), and *Cubic* domain decomposition (right).

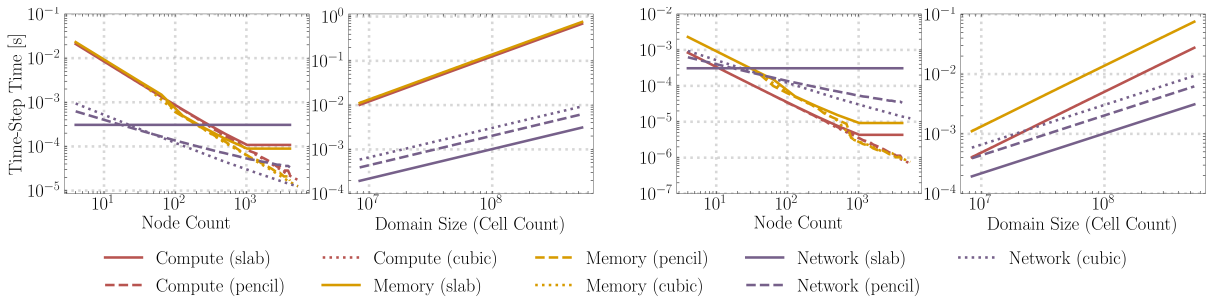


Figure 5.3: Plots (left → right): (1) Intel Sapphire Rapids CPU — strong scaling, (2) Intel Sapphire Rapids CPU — weak scaling, (3) Intel Max 1550 GPU — strong scaling, (4) Intel Max 1550 GPU — weak scaling. Each panel reports scaling performance for cubic, slab, and pencil domain decomposition schemes on a $256 \times 256 \times 1024$domain.

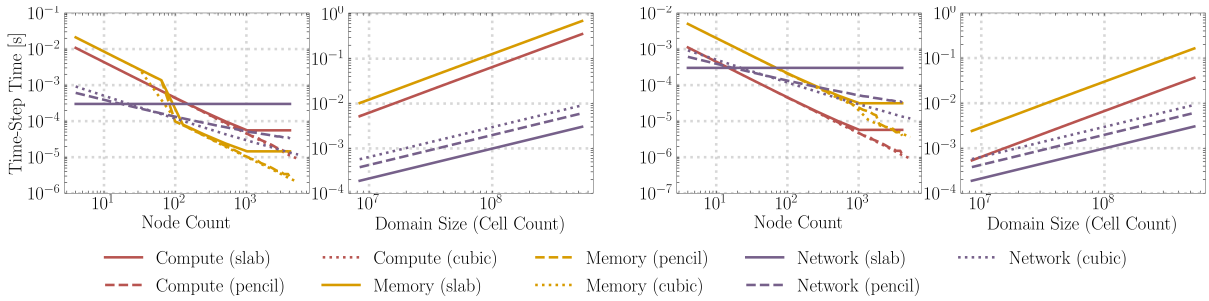


Figure 5.4: Plots (left → right): (1) NVIDIA Grace Superchip — strong scaling, (2) NVIDIA Grace Superchip — weak scaling, (3) NVIDIA H100 GPU — strong scaling, (4) NVIDIA H100 GPU — weak scaling. Each panel reports scaling performance for cubic, slab, and pencil domain decomposition schemes on a $256 \times 256 \times 1024$ domain.

cedure.

The choice of the numerical scheme determines how much data must be transferred. In the LVR case, given the 5^{th} order WENO scheme, for each boundary degree of freedom, 3 ghost degrees of freedom need to be retrieved from neighbour compute nodes.

Before explaining the domain decomposition schemes considered for LVR in more details, it is convenient to describe the data layout adopted in LVR for storing the degrees of freedom (which are the data that must be shared across boundaries). In particular, we adopt a Structure of Arrays (SoA) layout, where each field in Equation 7.1 is stored in its own separate array. The arrays are linearized in row-major order, with the x coordinate corresponding to the fastest-varying index, the y coordinate to the next-fastest, and the z coordinate to the slowest-varying index.

The three domain decomposition schemes are: *Slabs*, *Pencil*, and *Cubic*. All of them, are based on splitting the global domain along either one, two or three axes. For each axis along which the domain is split, each node must exchange three layers of ghost degrees of freedom on each side with its neighbouring nodes.

Slabs (Figure 5.2a) decompose the domain along the slowest-varying axis (z). Nodes are logically arranged in a linear topology, with two neighbors per node. In this scheme, the total number of nodes does not affect the amount of data communicated.

Pencil (Figure 5.2b) splits the domain along the two fastest-varying axes (y, z), arranging nodes in a 2D logical grid. Each node has four neighbors.

Cubic (Figure 5.2c) partitions the domain along all three axes, with nodes arranged in a 3D logical grid. Each node communicates with six neighbors. In Figure 5.4 and Figure 5.3 we compare LVR under these schemes using the TSM, on two different supercomputers: CSCS Alps (Figure 5.4) and ALCF Aurora (Figure 5.3), in both weak and strong scaling regime.

Weak scaling. In the weak-scaling regime, the network time is generally negligible compared to the other components, except on the two GPUs, where for subdomains of approximately 10^7 cells the compute time exceeds the network time for the cubic decomposition. Across all cases, the slab decomposition consistently offers the best performance, while the cubic decomposition remains the worst.

This can be explained by noting that, for sufficiently large subdomains, a slab decomposition results in less data being exchanged over the network. Given the size of the

subdomains involved, the network payloads are large enough to saturate the available bandwidth in all three decomposition schemes. Moreover, the compute and memory time scales coincide across all decompositions.

These observations suggest that a slab decomposition is the preferred choice whenever subdomain sizes exceed roughly 10^7 cells, corresponding to a domain of about $215 \times 215 \times 215$.

Strong scaling. From Figure 5.4 and Figure 5.3, it is evident that the slab decomposition outperforms the others by up to a factor of 3 up to roughly 20 nodes on both systems. Beyond this point, the cubic decomposition becomes the most effective. As the number of nodes increases and per-node subdomains shrink, the cubic (and, to a lesser extent, pencil) decomposition benefits from reduced communication volume.

On the two CPUs, LVR remains memory-bound up to approximately 10^2 nodes on the Intel Sapphire Rapids and up to about (70) nodes on the NVIDIA Grace Superchip. Once the subdomains become small enough to fit into the L3 cache, the compute time scale becomes dominant. At around (300) nodes for the Intel CPU and around (200) nodes for the NVIDIA CPU (considering the slab decomposition), LVR becomes network-bound. For the pencil decomposition, this transition occurs at around (2000) nodes on the Intel CPU and around (1000) nodes on the NVIDIA CPU. For the cubic decomposition, the cross-over point on both CPUs is approximately (4000) nodes.

On the two GPUs, LVR is memory-bound (for the slab decomposition) up to 70 nodes on the NVIDIA H100 and up to 30 nodes on the Intel MAX 1550, reflecting the lower memory bandwidth of the NVIDIA GPU. For the pencil decomposition, the Intel GPU becomes network-bound at around 70 nodes (the same cross-over point observed for the cubic decomposition), whereas for the NVIDIA GPU this occurs at around 300 nodes. Finally, for the cubic decomposition on the NVIDIA GPU, LVR becomes network-bound at around 600 nodes.

From this analysis, it becomes clear that for LVR the slab decomposition is the optimal choice in weak-scaling regimes, where each compute node is assigned a substantial workload. In contrast, under strong-scaling conditions, the cubic decomposition provides the best performance.

Given LVR's focus on high-throughput scenarios, the slab decomposition is therefore adopted. In addition, this scheme is considerably simpler to implement than the other two, as it only requires partitioning global data structures along a single dimension. This

Table 5.1: Time Scales when GPUs are used as accelerators, for a domain with 10^{11} cells.

Device	**Intel GPU 1550 Max**	**NVIDIA H100**
Compute Time	5 s	7 s
Memory Time	10.5 s	30 s
PCIe Time	60 s	35 s

leads to a simpler communication pattern, since each node needs to exchange data with at most two neighboring nodes.

5.2.2. Offloading to accelerators

GPUs can be used in two distinct regimes: In-Memory and Accelerator.

In the In-Memory regime, the problem size is small enough for all required data structures to fit in the GPU's global memory, the entire dataset can remain resident on the device throughout execution. In this scenario, host–device transfers are only needed to maintain consistency across subdomains. The data volumes involved are typically small enough that their impact on the total time-to-solution (TTS) is negligible.

However, when the GPU does not have sufficient memory to hold the full problem footprint, the device can only accelerate selected computational kernels (Accelerator regime), each operating on a subset of the domain at a time. This requires repeatedly uploading and downloading large portions of data between the host and the device. Because PCIe bandwidth is generally much lower than GPU memory bandwidth, the PCIe transfer time becomes a significant factor and must be included in any comprehensive performance model.

In Table 5.1, we assess the impact of such a regime with respect to the PCIe time, on the NVIDIA H100 GPU and the Intel GPU Max 1550, with a domain size that would not fit both the devices' memory. On the NVIDIA H100, the PCIe bandwidth is roughly three times higher than the Intel GPU Max 1550, which explains the difference in PCIe times between the two platforms.

In both cases, the PCIe time scale is higher than the compute and memory one, meaning that the impact of CPU-GPU data transfers is not negligible in this regime, and is likely to represent a bottleneck.

However, it must be considered that, while on past devices the difference in CPU and GPU memory was significant, on modern systems this not the case anymore. For example, on the CSCS Alps supercomputer and the ALCF Aurora supercomputer the ratio

Table 5.2: Analysis of FLUX kernel individual stages.

Stage	**Prj.**	**WENO**	**Inv. Prj.**	**HLLC**	**FLUX**
Per-point workload	101 F	680 F	32 F	290 F	1103 F
Per-point memory traffic	520 B	400 B	240 B	160 B	120 B
Operational intensity	0.19 F/B	1.7 F/B	0.13 F/B	1.81 F/B	9.19 F/B
Max FP64 peak fraction	3.7 %	47 %	3.6 %	50%	100 %
Per-point total memory traffic	1320 B				120 B

Table 5.3: Analysis of DUC kernel individual stages.

Stage	**DVG**	**UPD**	**CNV**	**DUC**
Per-point workload	7 F	2 F	33 F	35 F
Per-point memory traffic	960 B	360 B	360 B	600 B
Operational intensity	0.11 F/B	0.08 F/B	0.275 F/B	0.88 F/B
Max FP64 peak fraction	2.1 %	1.5 %	5.4 %	13.9%
Per-point total memory traffic	1680 B			600 B

between per-node CPU and GPU memory is 1.5 and 1.26, respectively. For this reason, it is likely that in the future, the Accelerator regime will become less relevant.

5.2.3. Kernel fusion

At each substep, two main kernels are executed: `FLUX` and `DUC`. As explained in chapter 3, a flux computation can be logically divided into four stages: variable projection, WENO reconstruction, inverse projection, and the HLLC Riemann solver. In contrast, the `DUC` kernel performs three operations: computing the divergence, applying the Runge-Kutta substep, and converting conserved variables back to their primitive form.

The decision to fuse these stages into two kernels was guided by the per-stage FLOP counts obtained from our symbolic framework.

In Table 5.2, the per-point workload and memory traffic of flux stages are reported. Using these data, we compute the operational intensity and the maximum achievable peak fractions according to Equation 5.3 for each stage. As evident from the table,

all stages taken individually exhibit low operational intensity and are therefore memory-bound, with theoretical compute performance well below the peak. In this context, kernel fusion is beneficial: it reduces per-point memory traffic because data are loaded and stored only once, while intermediate values are either retained in registers or spilled to memory. In the latter case, spilled values are likely cached due to the higher temporal locality of the fused kernel, as they are expected to be reused shortly thereafter. Conversely, in the unfused scenario, the output of each stage must be explicitly written to main memory. As shown in Table 5.2, the per-point effective memory traffic in the unfused case is more than an order of magnitude higher than in the fused case. Consequently, the `FLUX` kernel achieves a high operational intensity (9.19 F/B) and becomes compute-bound.

A similar analysis applies to the `DUC` stages, as summarized in Table 5.3. Individually, each stage performs minimal per-point arithmetic work while generating relatively high memory traffic, resulting in very low operational intensities and achievable performance. In this case, kernel fusion does not yield a compute-bound kernel. Nonetheless, `DUC` benefits from reduced per-point memory traffic and increased operational intensity compared to the individual stages, achieving a maximum fraction of peak performance of 13.9%.

6 | Program Synthesis

In the development of LVR, program synthesis was relied upon to a significant extent. Instead of manually writing the main computational kernels, they were generated starting from their mathematical formulation. The latter was expressed using the SymPy symbolic reasoning Python package. In the rest of this chapter, the program synthesis techniques leveraged in the present work are discussed in details.

6.1. Introduction to SymPy

SymPy is a Python package focused on symbolic computation. Through the usual Python syntax, it allows easily expressing complex mathematical expressions. SymPy provides a rich set of functionalities. Herein, only the one most pertaining to the present work are briefly described. Everything starts by defining *Symbols*, that is, immutable Python objects completely characterized by their name[1].

```
import sympy as sy
x = sy.Symbol("x")
y = sy.Symbol("y")
```

Symbols represent the simplest form of SymPy expressions. More complex expressions can be built by combining symbols or other expressions through the symbolic functions offered by Sympy, including the usual mathematical operators, such as +,-,*,/. SymPy expressions are internally represented as trees whose nodes are either operations or symbols.

```
import sympy as sy
z  = sy.Sin(2*x + y**2)
```

In addition to scalar symbols and operations, also matrix (which includes vector) ones are supported. In the following code snippet, it is shown how a purely symbolic vector v (whose entries are SymPy symbols), is multiplied twice by the 90° rotation matrix. It can

[1]This is of significant importance, as it allows determining two symbols' equality based upon their name only

be noticed how vectors are considered to be a special kind of matrix, sharing the same API.

```
import sympy as sy
x, y = sy.symbols("x y")
M = sy.Matrix([0, -1, 1, 0])
v = sy.Matrix([x, y])
w = M*M*v
```

Moreover, SymPy supports *Indexed* symbols, which enables expressing tensor operations, such as finite difference stencils. The following code snippet demonstrates how the 3-points finite difference scheme is applied.

```
1  import sympy as sy
2  def get(array, index):
3      subs = {}
4      for e in array.atoms(sy.Indexed):
5          base_name = str(e.base.label)
6          bounded_indices = []
7          for val in index:
8              bounded_indices.append(sy.Idx(val, (-1, 1)))
9          subs[e] = sy.IndexedBase(base_name, shape=(3,))[tuple(bounded_indices)]
10     return array.xreplace(subs)
11 R = 1 # gas constant
12 P = sy.IndexedBase("P")[0]
13 T = sy.IndexedBase("T")[0]
14 rho = P/(R*T) # density according to ideal gases' state equation
15 rho_diff = -1*get(rho, -1) + 2*get(rho, 0) -1*get(rho, 1)
```

In more details, the code shows how the first derivative of the density field is computed, when pressure and temperature field are available, assuming the ideal gases model .[2] On line 12-13, two *Indexed* symbols are created. On line 14, the derived density field is obtained. Unfortunately, the latter is not an actual SymPy symbol, but rather a *Mul* expression. Therefore, contrary to P and T, it cannot be subscripted. To address this issue, the *get* function (lines 2-10) can be employed to provide a similar functionality. It accepts as input an expression containing Indexed objects, and the index to access. It then replaces all instances of Indexed symbols with their correctly indexed counterpart. Now, on line 15, the density numerical derivative can be compactly computed.

When building complex expressions, it is not uncommon for certain sub-expressions to

[2]For ideal gases, the following equation of state holds: $p = \rho RT$, where p is the gas pressure, T is the gas temperature and R is the gas constant.

appear several times, which unnecessarily increases the expression complexity. To this end, SymPy includes the `sy.cse` routine, which finds recurring subexpressions within the input expression, and assigns them to temporary symbols, which are then replaced in the final expression. In practice, the CSE function returns a list of replacements (in which recurring subexpressions are assigned to temporary symbols), and a final expression, which depends on those temporary symbols. The following code snippet shows an example of how this function can be applied to a simple toy case.

```
import sympy as sy
x = sy.Symbol("x")
y = (x**2+1)*(x**2+2)
z = sy.cse(y)
```

As it can be seen, in the `y` expression, the `x**2` factor is repeated twice. This fact is correctly detected by the SymPy CSE function, which returns the following output:

```
([(x0, x**2)], [(x0 + 1)*(x0 + 2)])
```

Another powerful SymPy's feature is its capability to generate C code implementing a given expression. To this end the `sy.ccode` function can be used:

```
import sympy as sy
x = sy.Symbol("x")
y = (x**2+1)*(x**2+2)
print(sy.ccode(y))
```

which outputs

```
1 (pow(x, 2) + 1)*(pow(x, 2) + 2)
```

The use of the `cse` function can be neatly coupled with code generation. On one hand, eliminating recurring subexpressions is crucial to obtain a simpler and lighter code, without redundant operations. On the other hand, the simple output format of the CSE function allows generating the corresponding C code using a simple algorithm, implemented by the `expr2code` function shown in the following listing:

```
import sympy as sy
def expr2code(expr, output_identifiers):

    sub_expr, final_expr = sy.cse(list(expr))

    retval = ""
    for var, expr in sub_expr:
        retval += "     const double {} = {};\n".format(var, sy.ccode(expr))
```

```
    for k, v in zip(output_identifiers, final_expr):
        retval += "      %s = %s;\n" % (k, sy.ccode(v))

    return retval
```

The function takes as input a SymPy expressions and the identifiers of the output values (since the input expression may be a sy.Matrix, more than just one output value is possible). First, the common subexpression elimination on the input expression is performed, returning the found common subexpressions and the final expressions. Then, their corresponding C code is stored in the `retval` variable, which is finally returned.

6.2. SymPy's pitfalls

Despite SymPy's powerful symbolic computation and code generation features, if naively used it may expose some suboptimal behaviours, which required, for the purposes of the present work, to be properly addressed.

6.2.1. Use of the pow function

As shown in the previous section, when generating C code, SymPy makes use of the `pow` function, even when the exponent is a positive integer. It must be noticed that such a function is significantly less efficient than the corresponding repeated multiplications, particularly on GPUs. The latter optimization can be automatically applied by leveraging the SymPy built-in `create_expand_pow_optimization` function.

6.2.2. Inefficient application of the associative property

From our observations, we noticed that SymPy would consistently fail to properly optimize certain expressions, due to its inefficient application of the scalar multiplication associative property. For example, an expression as $y = 3(x+1)(x+1)$ may be associated either to the left, leading to $y = (3(x+1))\,(x+1)$ or to the right, leading to $y = 3((x+1)(x+1))$. While formally equivalent, in practice the second one exposes an important optimization opportunity: the $((x+1)(x+1))$ factor can be recast as $(x+1)^2$, thus reducing the operation count by 1 addition. As a result, the effective operation count of the first y expression is 2 additions and 2 multiplication, while for the second one it is 1 addition and 2 multiplications. Expressions like y appear in the symbolic code of the FLUX kernel.

6.2.3. High operation count and register pressure

When the SymPy `CSE` function is applied on long and complex expressions, followed by code generation, we noticed that the resulting code involves an inflated number of operations and a high register pressure. To overcome this issue, we adopt a *scope limiting* approach. We split the full expression into several consecutive subexpressions, that are individually optimized and turned into code. More details about how each kernel expression has been split are provided in the subsequent sections of this chapter. A potential drawback of this approach is that, as subexpressions are independently optimized, some inter-subexpressions optimization may be missed. However, from our observations, we found that better results are obtained with our *scope limiting* approach. In order to support this feature, the changes in the way expressions are built and in the code generation process have been required. However, we ensured to maintain the original simplicity and straightforwardness. In particular, we wanted the symbolic code to remain as clean as possible, abstracting away the technical details involved in the splitting of the full expression. The following listings show the comparison between the two approaches.

```
import sympy as sy
import synthesis sn
x, y = sy.symbols("x y")
M = sy.Matrix([0, -1, 1, 0])
v = sy.Matrix([x, y])
w = M*M*v
print(sn.expr2code(w))
```

```
import sympy as sy
import synthesis as sn
step, cse_list = sn.step_wrap(iterator)
x, y = sy.symbols("x y")
M = sy.Matrix([0, -1, 1, 0])
v = sy.Matrix([x, y])
w1 = step(M*v)
w2 = step(M*w1)
print(sn.step_by_step_expr2code(cse_list))
```

The `step` function takes an expression as input, and returns a fresh symbol (or a matrix of fresh symbols, depending on the input type), and updates the `cse_list` list with the new stage. The `cse_list` is then passed as parameter to the `step_by_step_expr2code` function, which optimizes and prints each stage's expression. The implementation of the `step_wrap` function is provided in the following listing.

```
import sympy as sy
def step_wrap(symb_iter):
    cse_list = []

    def add_step(exp, terminal_name=None):
        name = next(symb_iter) if terminal_name is None else terminal_name
        cse_list.append((sy.cse(exp), [name], terminal_name==None))
        return sy.Symbol(str(name))
```

```
def step_impl(expr, terminal_name=None, opt=None, fma_opt_=False):
    if isinstance(expr, sy.Matrix):
        fma_opt = build_fma_opt({})
        if fma_opt_ == True:
            cse_res = sy.cse([fma_opt(e) for e in expr], optimizations=opt)
        else:
            cse_res = sy.cse([e for e in expr], optimizations=opt)
        names = [terminal_name(i) if not terminal_name is None
            else str(next(symb_iter))
        for i in range(len(expr))]
        cse_list.append((cse_res, names, terminal_name==None))
        return sy.Matrix([sy.Symbol(name) for name in names])

    else:
        fma_opt = build_fma_opt({})
        if fma_opt_ == True:
            return add_step(fma_opt(expr))
        else:
            return add_step(expr)

return step_impl, cse_list
```

6.3. Symbolic-enabled optimizations

In LVR development, SymPy capabilities have been exploited to apply platform-specific optimizations.

6.3.1. Fused-Multiply Adds (FMAs) capturing

Inspection of PTX code of CUDA kernels revealed that, opportunities to emit FMA instructions were missed by the *nvcc* compiler. FMAs are twice as efficient than their split counterpart, it is therefore mandatory to guide the compiler in generating the desired instructions. To achieve this goal, our symbolic workflow captures FMAs present in the expressions generated by SymPy CSE stage and generate custom wrappers to `__fma_rn` CUDA intrinsics. As a result, a 3.3% performance increase was observed on the NVIDIA H100 GPU.

6.3.2. Reciprocal (RCP) capturing

On GPUs, the cost of a double precision reciprocal is comparable with the cost of an FMA [5]. Therefore, we decided to capture and replace each division a/b in the flux kernel, with the corresponding $a \cdot 1/b$. This makes sure the compiler, instead of a division instruction, emits a reciprocal followed by a multiplication. A 28% performance gain was obtained on the NVIDIA H100 GPU.

6.4. FLUX kernel

The FLUX kernel is the main kernel for which our program synthesis process has been widely employed. Fluxes computation can be logically divided into four steps: 1) variables projections into characteristic space, 2) projected variables reconstruction through the WENO scheme, 3) back projections into the original space and 4) numerical fluxes computation using the HLLC solver. This subdivision is reflected in the FLUX symbolic code.

```
import sympy as sy
def flux_expr(step, name_func, ndim = 3, symbs = None):
    t_m2, t_m1, t_0, t_p1, t_p2, t_p3 =
    sy.symbols("t_m2[i] t_m1[i] t_0[i] t_p1[i] t_p2[i]t_p3[i]")
    u_m2, u_m1, u_0, u_p1, u_p2, u_p3 =
    sy.symbols("u_m2[i] u_m1[i] u_0[i] u_p1[i] u_p2[i] u_p3[i]")
    v_m2, v_m1, v_0, v_p1, v_p2, v_p3 =
    sy.symbols("v_m2[i] v_m1[i] v_0[i] v_p1[i] v_p2[i] v_p3[i]")
    w_m2, w_m1, w_0, w_p1, w_p2, w_p3 =
    sy.symbols("w_m2[i] w_m1[i] w_0[i] w_p1[i] w_p2[i] w_p3[i]")
    p_m2, p_m1, p_0, p_p1, p_p2, p_p3 =
    sy.symbols("p_m2[i] p_m1[i] p_0[i] p_p1[i] p_p2[i] p_p3[i]")

   if ndim < 3:
      w_m2, w_m1, w_0, w_p1, w_p2, w_p3 = 0, 0, 0, 0, 0, 0

   if ndim < 2:
      v_m2, v_m1, v_0, v_p1, v_p2, v_p3 = 0, 0, 0, 0, 0, 0

   # (S^-1 W)
   ############################################################
   p_avg = 0.5*(p_0 + p_p1)
```

```
t_avg = 0.5*(t_0 + t_p1)
rho_avg = p_avg/(R*t_avg)
c_avg = sy.sqrt(gamma*R*t_avg)

alpha=1/t_avg # (1/T)
beta = 1/p_avg
imp = c_avg*rho_avg
sinv4 = -(imp*c_avg*beta-1)/(imp*c_avg*alpha)
imp, sinv4 = step(sy.Matrix([imp, sinv4]))
S_inverse = sy.Matrix([[1,       0,   0,   0, sinv4],
                       [0,       0,   1,   0, 0],
                       [0,       0,   0,   1, 0],
                       [0, 0.5*imp,   0,   0, 0.5],
                       [0,-0.5*imp,   0,   0, 0.5]])

W_m2 = sy.Matrix([[t_m2], [u_m2], [v_m2], [w_m2], [p_m2]])
W_m1 = sy.Matrix([[t_m1], [u_m1], [v_m1], [w_m1], [p_m1]])
W_0  = sy.Matrix([[t_0 ], [u_0 ], [v_0 ], [w_0 ], [p_0 ]])
W_p1 = sy.Matrix([[t_p1], [u_p1], [v_p1], [w_p1], [p_p1]])
W_p2 = sy.Matrix([[t_p2], [u_p2], [v_p2], [w_p2], [p_p2]])
W_p3 = sy.Matrix([[t_p3], [u_p3], [v_p3], [w_p3], [p_p3]])

c_m2 = step(S_inverse*W_m2)
c_m1 = step(S_inverse*W_m1)
c_0  = step(S_inverse*W_0)
c_p1 = step(S_inverse*W_p1)
c_p2 = step(S_inverse*W_p2)
c_p3 = step(S_inverse*W_p3)

S_ = sy.Matrix([[1,0,0,0,0],[0,0,0,1/imp,-1/imp],
     [0,1,0,0,0],[0,0,1,0,0],[0,0,0,1,1]])
S_[3] = -S_inverse[4]
S_[4] = -S_inverse[4]

#print(S_ * S_inverse)

# WENO (S^-1 W)
###############################################################
w_tilda_left  = []
w_tilda_right = []
```

```
    for i in range(5):
        w_l, w_r = weno.weno(c_m2[i], c_m1[i], c_0[i],
              c_p1[i], c_p2[i], c_p3[i], step)
        w_vec = step(sy.Matrix([w_l, w_r]))
        w_tilda_left.append(w_vec[0])
        w_tilda_right.append(w_vec[1])


    # S (WENO (S^-1 W))
    ###########################################################
    w_left  = step(S_*sy.Matrix(w_tilda_left))
    w_right = step(S_*sy.Matrix(w_tilda_right))

    # HLLC(S (WENO (S^-1 W)))
    ###########################################################

    def extract(r):
        return [r[0]] + list(r[1:1+ndim]) + [r[4]]

    return extract(step(hllc.hllc(w_left [0], w_left [1],
              w_left [2], w_left [3], w_left [4],
              w_right[0], w_right[1], w_right[2], w_right[3],
              w_right[4], step), terminal_name=name_func))
```

The WENO and HLLC stages are implemented as shown in the following listings.

```
import sympy as sy
def weno(a,b,c,d,e,f, step):
    C13 =  13.0/12.0
    C23 =  3.0/12.0
    aux1 = (b - 2*c + d); aux2 = (        b - d);
    is1_m = ((C13*(aux1*aux1) + C23*(aux2*aux2)))
    aux1 = (c - 2*d + e); aux2 = (3*c - 4*d + e);
    is2_m = ((C13*(aux1*aux1) + C23*(aux2*aux2)))
    aux1 = (a - 2*b + c); aux2 = (3*c - 4*b + a);
    is0_m = ((C13*(aux1*aux1) + C23*(aux2*aux2)))
```

```
aux1 = (e - 2*d + c); aux2 = (        e - c);
is1_p = ((C13*(aux1*aux1) + C23*(aux2*aux2)))
aux1 = (d - 2*c + b); aux2 = (3*d - 4*c + b);
is2_p = ((C13*(aux1*aux1) + C23*(aux2*aux2)))
aux1 = (f - 2*e + d); aux2 = (3*d - 4*e + f);
is0_p = ((C13*(aux1*aux1) + C23*(aux2*aux2)))
tmp = step(sy.Matrix([is1_m, is2_m, is0_m, is1_p, is2_p, is0_p]), fma_opt_=True)
is1_m = tmp[0]
is2_m = tmp[1]
is0_m = tmp[2]
is1_p = tmp[3]
is2_p = tmp[4]
is0_p = tmp[5]


is0plus_m = step(is0_m + WENOEPS)
is1plus_m = step(is1_m + WENOEPS)
is2plus_m = step(is2_m + WENOEPS)

q0_m = step(is1plus_m*is1plus_m*is2plus_m*is2plus_m)
q1_m = step(6*is0plus_m*is0plus_m*is2plus_m*is2plus_m)
q2_m = step(3*is0plus_m*is0plus_m*is1plus_m*is1plus_m)

is0plus_p = step(is0_p + WENOEPS)
is1plus_p = step(is1_p + WENOEPS)
is2plus_p = step(is2_p + WENOEPS)

q0_p = step(is1plus_p*is1plus_p*is2plus_p*is2plus_p)
q1_p = step(6*is0plus_p*is0plus_p*is2plus_p*is2plus_p)
q2_p = step(3*is0plus_p*is0plus_p*is1plus_p*is1plus_p)

return [(q0_m * (+1./3 * a - 7./6 * b + 11./6 * c) + \
      q1_m * (-1./6 * b + 5./6 * c + 1./3 * d) +  \
      q2_m * (+1./3 * c + 5./6 * d - 1./6 *
      e))/(q0_m+q1_m+q2_m),
      (q0_p * (+1./3 * f - 7./6 * e + 11./6 * d) + \
      q1_p * (-1./6 * e + 5./6 * d + 1./3 * c) +  \
      q2_p * (+1./3 * d + 5./6 * c - 1./6 *
      b))/(q0_p+q1_p+q2_p)]
```

```
import sympy as sy
def hllc(tL, uL, vL, wL, pL, tR, uR, vR, wR, pR, step):
   rhoL = step(pL/(R*tL))
   rhoR = step(pR/(R*tR))

   aL = step(sy.sqrt(gamma * R * tL))
   aR = step(sy.sqrt(gamma * R * tR))

   ubar = step((uL + uR) / 2)
   abar = step((aL + aR) / 2)

   sL = step(sy.Min(ubar - abar, uL - aL))
   sR = step(sy.Max(ubar + abar, uR + aR))

   sm = step(sy.Min(0, sL))
   sp = step(sy.Max(0, sR))

   ss = step((pR - pL + uL * rhoL * (sL - uL) - uR * rhoR * (sR - uR)) /\
                        (rhoL * (sL - uL) -       rhoR * (sR - uR)))

   EL = step(pL/(rhoL*(gamma-1)) + (uL**2+vL**2+wL**2)/2)
   ER = step(pR/(rhoR*(gamma-1)) + (uR**2+vR**2+wR**2)/2)

   # conservative state
   def primitive_2_conserved(rho, u, v, w, E):
      return sy.Matrix([rho, rho*u, rho*v, rho*w, rho*E])
   # intermediate conservative state
   def ics(rho, u, v, w, p, s, E):
      return (s-u)/(s-ss) * sy.Matrix([rho, rho*ss, rho*v,
                            rho*w, rho*E + (ss-u) * (rho*ss+ p/(s-u))])
   # physical flux
   def F(rho, u, v, w, p, E):
      return sy.Matrix([rho*u, rho*u**2 + p, rho*u*v, rho*u*w, (rho*E+p)*u])

   UL = step(primitive_2_conserved(rhoL, uL, vL, wL, EL))
   UR = step(primitive_2_conserved(rhoR, uR, vR, wR, ER))

   fluxL = step(F(rhoL, uL, vL, wL, pL, EL))
   fluxR = step(F(rhoR, uR, vR, wR, pR, ER))
```

```
    UsL = step(ics(rhoL, uL, vL, wL, pL, sL, EL))
    UsR = step(ics(rhoR, uR, vR, wR, pR, sR, ER))

    HLLC = (1 + sy.sign(ss)) / 2 * (fluxL + sm * (UsL - UL)) +\
           (1 - sy.sign(ss)) / 2 * (fluxR + sp * (UsR - UR))

    return HLLC
```

It can be noticed how we applied the *scope limiting* approach described in the previous section. Compared to non-split approach, it resulted in a 50% reduction in overall operation count. Moreover, in the computation of `is0, is1, is2` in the WENO stage, it can be seen how the `aux` coefficients are grouped as explained in subsection 6.2.2.

In Figure 6.1, a sample of the C code of the flux kernel, as generated by our program synthesis pipeline, is shown. It can be seen it is dense in floating-point operations, potentially exposing a significant amount of ILP. In Figures from 6.2 to 6.4, the assembly code for, respectively, the Intel Sapphire Rapids, the NVIDIA GH200 GPU and the AMD MI 300A GPU is reported. In all of them, it can be noticed how rich they are in FMAs instructions, and, more in general, floating-point instructions.

```
const double x113 = 10*x111 + x36;
const double x114 = 6*1.0/(x107 + x112)*1.0/(10*x107 + x113);
const double x115 = x47 - x48;
const double x116 = (x11 - x40 + x41 - x42 + x44)*(x115 + x45 + x46 + x50 - x51);
const double x117 = -x57;
const double x118 = (x115 + x117 + x5 + x55 + x56)*(-x59 + x60 + x61 - x62 + x65);
const double x119 = 1.0/(WENOEPS + x116 + x118)*1.0/(10*x116 + 10*x118 + x36);
const double x120 = -x70;
const double x121 = x120 + x72;
const double x122 = (0.125*p_m1[i] + 0.375*p_p1[i] - x108 - x42 - x75)*(1.5*p_p1[i] -
const double x123 = -x5;
const double x124 = (0.5416666666666663*p_m1[i] + 0.5416666666666663*p_p1[i] - x62
const double x125 = WENOEPS + x124;
const double x126 = 10*x124 + x36;
const double x127 = 3*1.0/(x122 + x125)*1.0/(10*x122 + x126);
const double x128 = 1.0/(x114 + x119 + x127);
const double x129 = -x87 - x88 + x89 + x90;
const double x130 = -x96 - x97 + x98 + x99;
const double x131 = x114*x128*x129 + x119*x128*(-x92 - x93 + x94 + x95) + x127*x128*
const double x132 = x105*x131;
const double x133 = 1.0/GAMMA;
const double x134 = GAMMA - 1;
const double x135 = -x134;
const double x136 = x104*x133*x135;
const double x137 = p_0[i]*x136;
const double x138 = t_0[i] + x137;
```

Figure 6.1: Sample of the flux kernel C code generated by our program synthesis pipeline.

```
vmulpd    zmm12{k6}{z}, zmm12, zmm11
mov       r11, QWORD PTR [80+rbx]
vmovaps   zmm14, zmm24
vfmadd213pd zmm7{k6}{z}, zmm11, zmm12
vmovupd   zmm2{k6}{z}, ZMMWORD PTR [r11+rsi*8]
vfmadd213pd zmm10{k6}{z}, zmm11, zmm7
vfnmadd213pd zmm14{k6}{z}, zmm0, zmm2
vmulpd    zmm4{k6}{z}, zmm20, zmm2
vmovups   ZMMWORD PTR [-432+rbp], zmm10
vmovupd   zmm10{k6}{z}, ZMMWORD PTR [r13+rsi*8]
vsubpd    zmm7{k6}{z}, zmm8, zmm2
vaddpd    zmm12{k6}{z}, zmm14, zmm8
vaddpd    zmm15{k6}{z}, zmm26, zmm4
vmulpd    zmm27{k6}{z}, zmm20, zmm10
vmovaps   zmm5, zmm28
vfmadd213pd zmm5{k6}{z}, zmm2, zmm13
vfmadd231pd zmm27{k6}{z}, zmm8, zmm17
vmulpd    zmm11{k6}{z}, zmm7, zmm5
vmulpd    zmm5{k6}{z}, zmm12, zmm15
vaddpd    zmm7{k6}{z}, zmm30, zmm5
vfmadd213pd zmm5{k6}{z}, zmm22, zmm31
vaddpd    zmm29{k6}{z}, zmm11, zmm7
vfmadd132pd zmm11{k6}{z}, zmm5, zmm22
vdivpd    zmm13{k6}{z}, zmm23, zmm29
vdivpd    zmm12{k6}{z}, zmm23, zmm11
vmulpd    zmm11{k6}{z}, zmm25, zmm0
vmulpd    zmm14{k6}{z}, zmm16, zmm13
vmulpd    zmm26{k6}{z}, zmm14, zmm12
vmulpd    zmm14{k6}{z}, zmm20, zmm0
```

Figure 6.2: Flux kernel assembly listing for the Intel Sapphire Rapids CPU.

```
sub.f64     %fd642, %fd640, %fd641;
add.f64     %fd643, %fd622, %fd642;
add.f64     %fd644, %fd639, %fd643;
mul.f64     %fd645, %fd621, 0d3FD0000000000000;
sub.f64     %fd646, %fd645, %fd380;
sub.f64     %fd647, %fd646, %fd379;
fma.rn.f64  %fd648, %fd620, 0d3FD0000000000000, %fd647;
add.f64     %fd649, %fd637, %fd648;
mul.f64     %fd650, %fd644, %fd649;
add.f64     %fd651, %fd635, 0d3EB0C6F7A0B5ED8D;
add.f64     %fd652, %fd651, %fd650;
rcp.rn.f64  %fd653, %fd652;
mul.f64     %fd654, %fd650, 0d4024000000000000;
fma.rn.f64  %fd655, %fd635, 0d4024000000000000, %fd654;
add.f64     %fd656, %fd655, 0d3EE4F8B588E368F0;
div.rn.f64  %fd657, %fd653, %fd656;
mul.f64     %fd658, %fd5, 0dC010000000000000;
mul.f64     %fd659, %fd314, 0d4010000000000000;
sub.f64     %fd660, %fd658, %fd659;
add.f64     %fd661, %fd316, %fd660;
add.f64     %fd662, %fd639, %fd661;
sub.f64     %fd663, %fd309, %fd5;
sub.f64     %fd664, %fd663, %fd314;
fma.rn.f64  %fd665, %fd315, 0d3FD0000000000000, %fd664;
add.f64     %fd666, %fd637, %fd665;
mul.f64     %fd667, %fd662, %fd666;
add.f64     %fd668, %fd332, %fd667;
div.rn.f64  %fd669, %fd144, %fd668;
fma.rn.f64  %fd670, %fd667, 0d4024000000000000, %fd333;
```

Figure 6.3: Flux kernel assembly listing for the NVIDIA GH200 GPU.

```
v_mul_f64 v[50:51], v[50:51], v[50:51]
v_fma_f64 v[2:3], s[18:19], v[2:3], v[32:33]
v_fma_f64 v[50:51], s[18:19], v[50:51], v[52:53]
v_mul_f64 v[58:59], v[38:39], s[28:29]
v_add_f64 v[2:3], v[2:3], s[10:11]
v_add_f64 v[50:51], v[50:51], s[10:11]
v_fmac_f64_e32 v[58:59], s[34:35], v[40:41]
v_mul_f64 v[2:3], v[2:3], v[2:3]
v_mul_f64 v[50:51], v[50:51], v[50:51]
v_mul_f64 v[54:55], v[50:51], v[2:3]
v_fmac_f64_e32 v[58:59], s[26:27], v[44:45]
v_mul_f64 v[44:45], v[48:49], v[48:49]
v_mul_f64 v[54:55], v[54:55], v[58:59]
v_mul_f64 v[58:59], v[46:47], s[16:17]
v_mul_f64 v[48:49], v[44:45], s[40:41]
v_mul_f64 v[44:45], v[44:45], s[8:9]
v_fmac_f64_e32 v[58:59], s[38:39], v[42:43]
v_mul_f64 v[44:45], v[44:45], v[2:3]
v_fmac_f64_e32 v[58:59], s[26:27], v[40:41]
v_fmac_f64_e32 v[54:55], v[58:59], v[44:45]
v_fmac_f64_e32 v[44:45], v[48:49], v[50:51]
v_fmac_f64_e32 v[44:45], v[50:51], v[2:3]
v_mul_f64 v[2:3], v[48:49], v[50:51]
v_rcp_f64_e32 v[48:49], v[44:45]
v_mul_f64 v[70:71], v[38:39], s[16:17]
v_fmac_f64_e32 v[70:71], s[38:39], v[40:41]
v_fmac_f64_e32 v[70:71], s[26:27], v[42:43]
v_fmac_f64_e32 v[54:55], v[70:71], v[2:3]
```

Figure 6.4: Flux kernel assembly listing for the AMD MI 300A GPU.

7 | Results

LVR has been thoroughly tested, both in terms of correctness and performance. In this chapter results of these tests are presented. Furthermore, we also report one application example where LVR have been applied, demonstrating its effectiveness in carrying out production simulation.

7.1. Validation

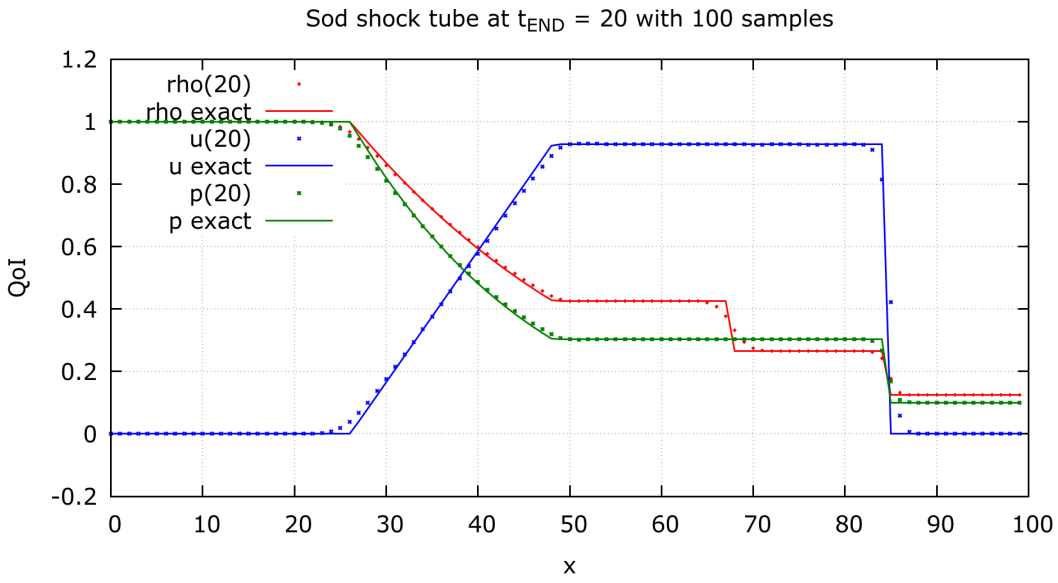
7.2. Sod and Lax shock tube

In this section, LVR simulation results for the Sod and Lax shock tube are presented, and validated by comparing them to reference values. The latter are computed by mean of an exact 1D Riemann solver. In both cases, the numerical solution is computed on a 3D 100^3 grid. The tests have been repeated for all the six possible shock directions, imposing appropriate initial conditions, as described below. Velocities orthogonal to the shock directions are set to 0.

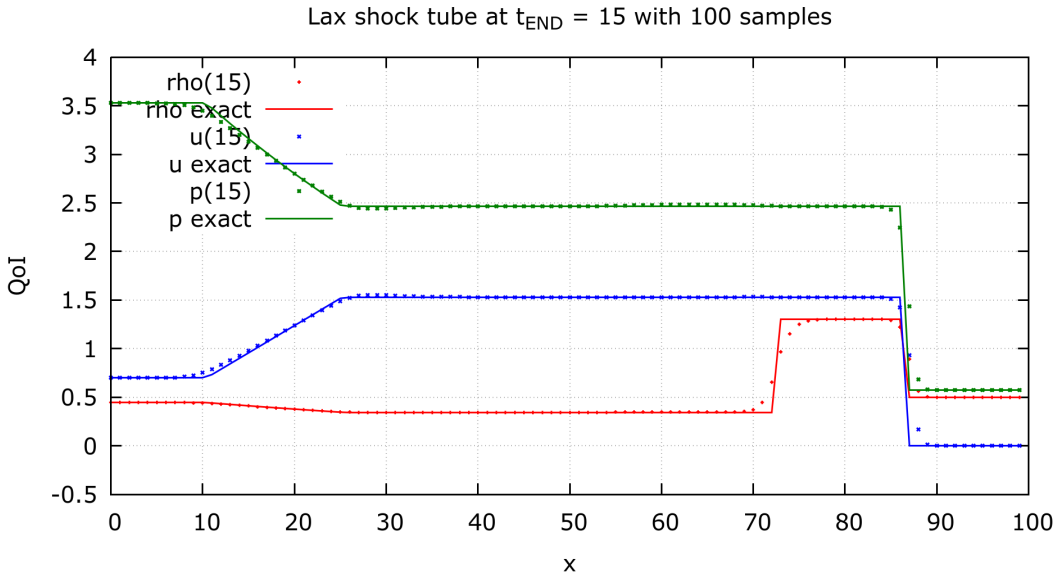
These tests allow assessing the shock capturing properties of the employed numerical schemes.

Sod's shock tube. This test case consists of a shock tube with length 1.0 in arbitrary units. The tube is filled with an ideal gas, with $\gamma = 1.4$ and $R = 1$. For $x < 0.5$. Initially pressure and temperature are set to 1, while for $x \geq 0.5$ they are set to 0.1 and 0.8 respectively. Velocity initial value is 0 everywhere. Figure 7.1a shows the obtained results, after $0.2s$ for the pressure, density and velocity. It can be observed how the reference and the numerical solution are in good agreement. Some discrepancies are observed at the contact lines, where the three quantities are slightly broadened. As reported in [13], this is expected and due to the employed flux splitting method.

Lax's shock tube. In this configuration, pressure, temperature and velocity are initialized as follows: 3.528, 7.928, 0.698 respectively, for $x < 0.5$, and 0.571, 1.142, 0 for



(a)



(b)

Figure 7.1: Reference (solid line) and numerical (dotted line) solutions for the Sod (top) and Lax's (bottom) shock tube tests.

$x \geq 0.5$. Obtained results are shown in Figure 7.1b.

Also in this case, the numerical solution closely matches the exact one. The main differences to notice are the ones identified for the Sod's shock tube, in addition to a slight overshoot of the velocity and both overshoot and undershoot of the pressure. Similar discrepancies have been previously reported for solvers employing LVR's numerical schemes.

7.3. Convergence order

LVR convergence order has been assessed on the Sod's shock tube described above, and on the Isentropic Vortex case.

Sod's shock tube. As mentioned in chapter 3, in the presence of shocks, L_1 error convergence is expected to be first-order, L_2 error convergence is expected to be half-order, and L_∞ error convergence is expected to be zero-order. The numerical solution has been obtained for spatial step lengths from 10^{-2} to 10^{-4}, and compared with a reference solution computed as in section 7.2. L_1, L_2 and L_∞ norms of the error have been computed. Results are shown in Figure 7.2. As it can be seen, obtained error convergence order are in good agreement with theoretical expectations.

Isentropic Vortex. The Isentropic Vortex corresponds to the following flow fields definition, which solve the inviscid Euler equations:

$$U(x,y,t) = U_\infty - \frac{\beta}{2\pi} e^{\frac{1-r^2(x,y,t)}{2}} r_y^2(y,t) \tag{7.1a}$$

$$V(x,y,t) = V_\infty + \frac{\beta}{2\pi} e^{\frac{1-r^2(x,y,t)}{2}} r_x^2(x,t) \tag{7.1b}$$

$$T(x,y,t) = T_\infty \left(1 - \frac{(\gamma-1)\beta^2}{8\gamma\pi^2} e^{1-r^2(x,y,t)}\right) \tag{7.1c}$$

$$P(x,y,t) = T(x,y,t)^{\frac{\gamma}{\gamma-1}} \tag{7.1d}$$

where, $r_x^2(x,t) = (x - U_\infty t - x_C)^2$, $r_y^2(y,t) = (y - V_\infty t - y_C)^2$, and x_C, y_C the initial center coordinates of the vortex. The above flow fields are used to compute the norm of the error of LVR output. The simulation is run on a 10x10 domain in arbitrary units, for 0.2 time units. The per-direction cell count is varied from 16 to 512. Results are shown in Figure 7.3. For L_1, L_2, L_∞ norms, second-order convergence can be observed.

7.4. Performance

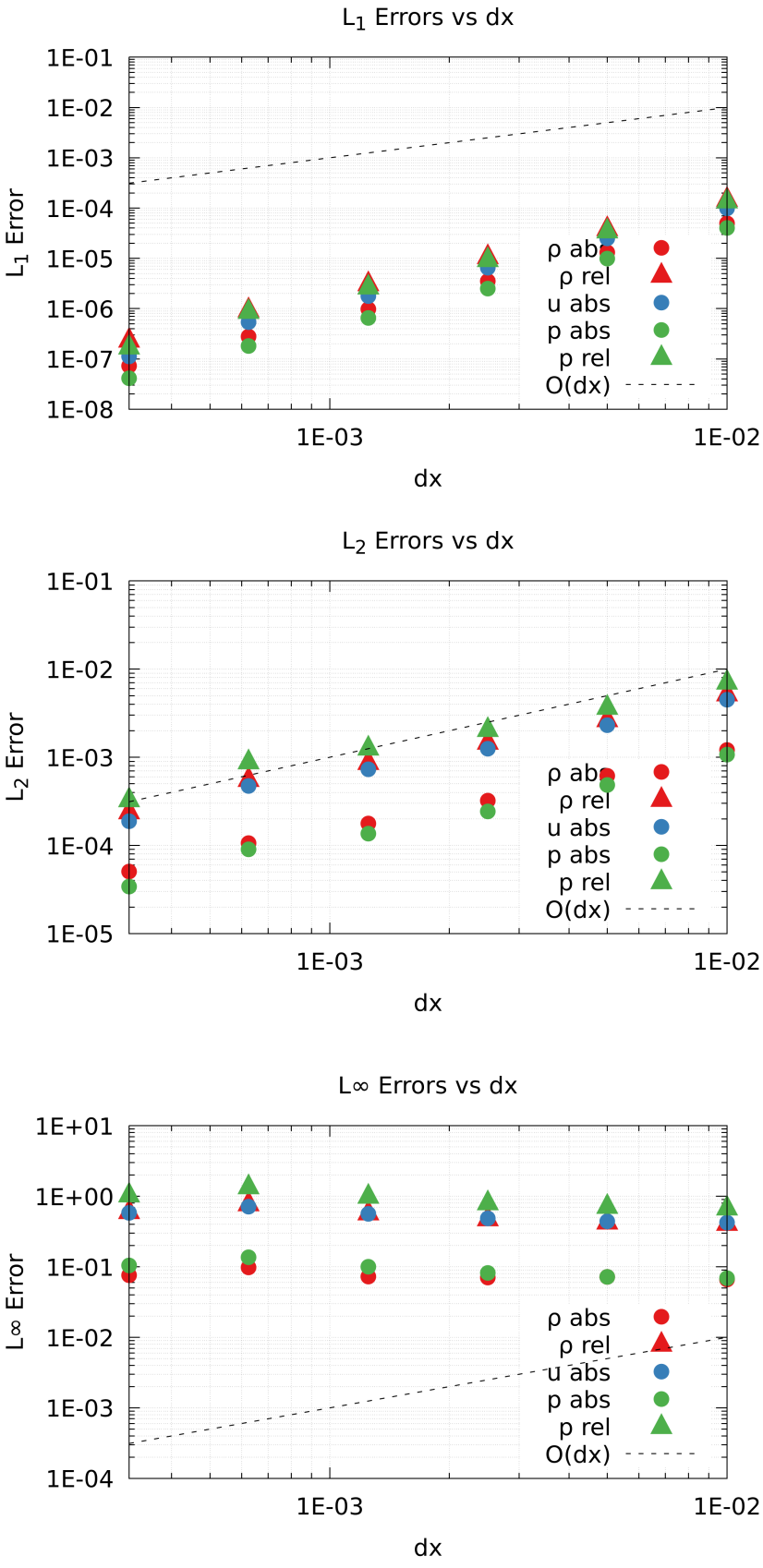


Figure 7.2: L_1 (top), L_2 (center), and L_∞ (bottom) error norms for the Sod's sock tube test case, as functions of the spatial step length.

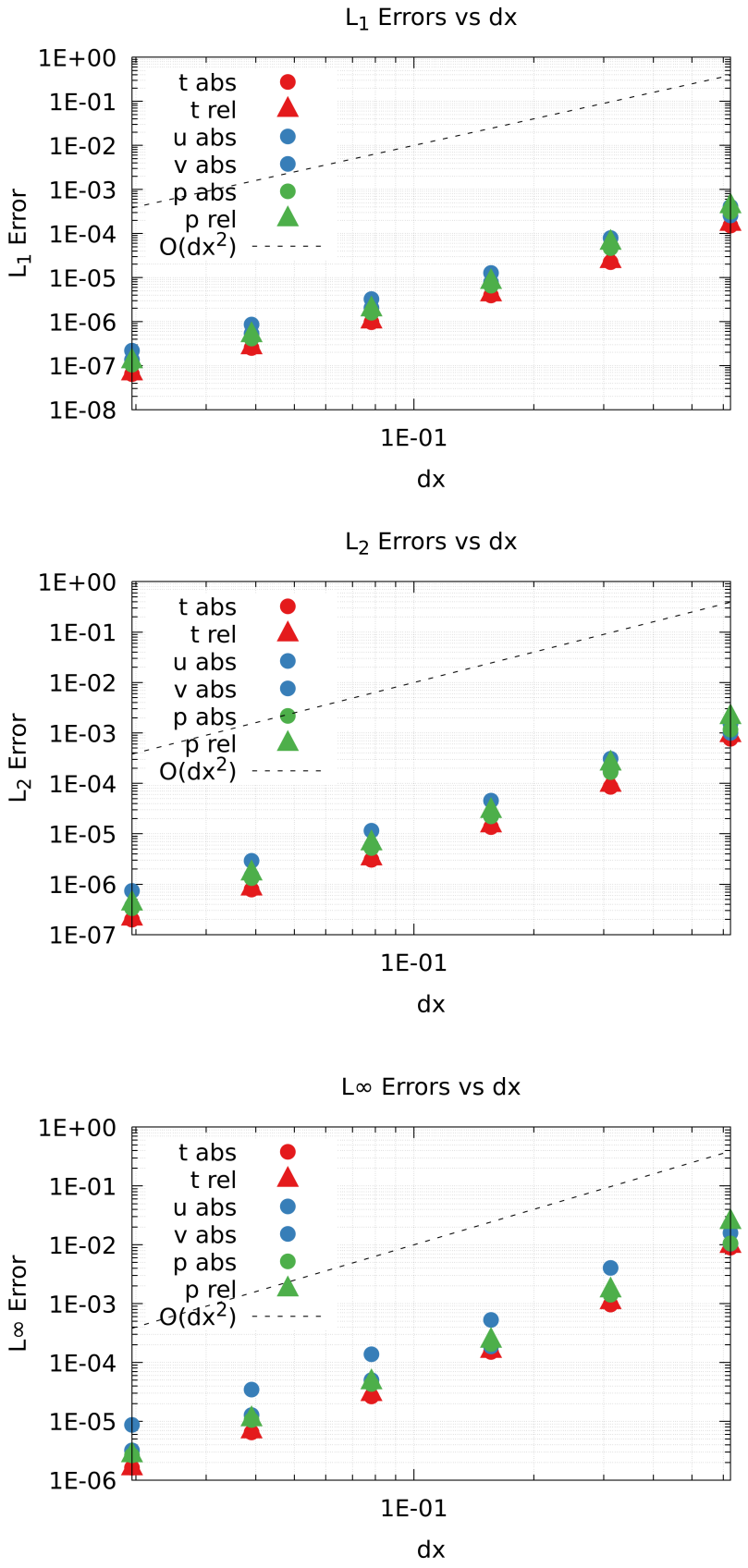


Figure 7.3: L_1 (top), L_2 (center), and L_∞ (bottom) error norms for the Isentropic Vortex test case, as functions of the spatial step length.

Table 7.1: Per-node performance characteristics.

		Alps	Aurora
	Facility	CSCS	ALCF
	Sockets	4	2
CPU	Vendor	NVIDIA	Intel
	Model	Grace Superchip	CPU Max
	Core count	288	104
	Threading	N/A	HT
	FP64 Peak	14.2 TF/s	6.7 TF/s
	DRAM Peak	1.6 TB/s DDR5	0.44 TB/s
			1.28 TB/s HBM
GPU	Vendor	NVIDIA	Intel
	Model	H100	GPU Max
	Count	4	6
	FP64 Peak	136 TF/s	178.2 TF/s
	BW Peak	12.8 TB/s	10.8 TB/s
	Notes		Exascale class

7.4.1. Experimental setup

The experimental setup consists of two platforms, summarized in Table 7.1. We focus on overall simulation throughput because it is a reliable metric, and TTS is straightforward to measure when it is not too small. For multi-node studies we considered a "slab" domain decomposition on the slowest changing index.

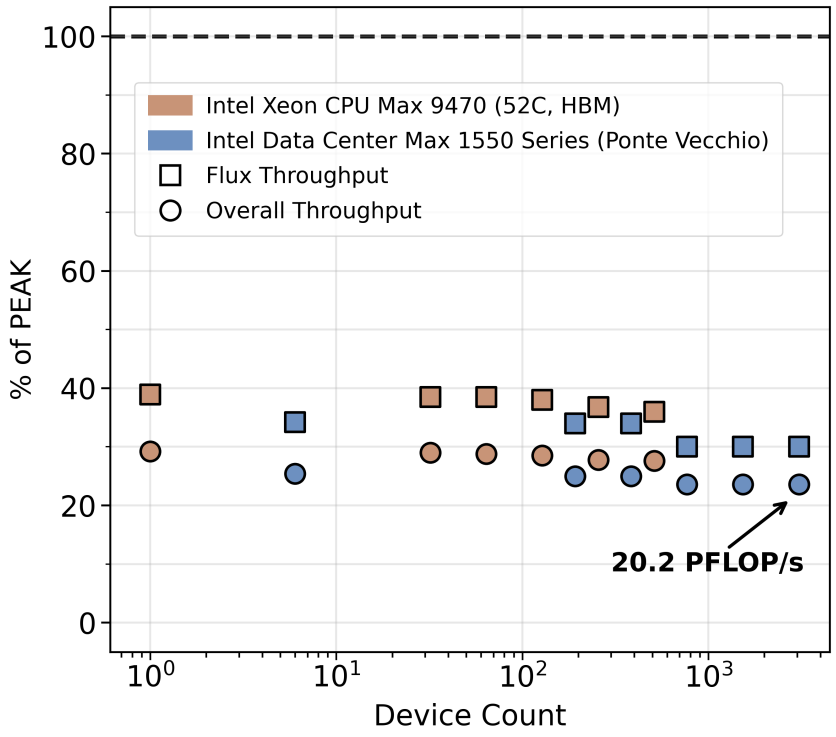
Unlike prior work that reports only relative numbers, our symbolic framework is able to precisely count floating point operations, and we rely on Equation 1 to compute fraction of peak.

The presented results were obtained from production runs and hence include real workload issues like resource contention. Performance reported in the rest of this section, refer to the average of several iterations.

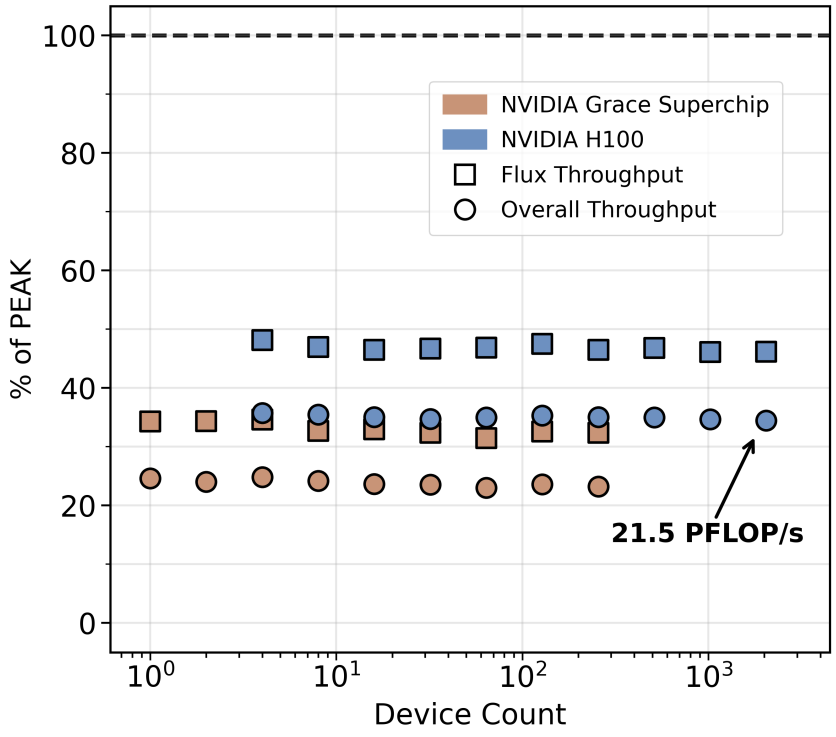
7.4.2. Performance results

CSCS Alps

Scaling benchmarks consider up to 256 nodes (1024 GPUs, 1024 cores), and a per-node gird consisting of more than 2 billions points and 1 billions points for CPUs and GPUs,



(a)



(b)

Figure 7.4: LVR weak scaling on ALCF Aurora (top) and CSCS Alps (bottom).

respectively. Results are shown in Figure 7.4b. On the NVIDIA Grace Superchip, LVR achieves a fraction of peak FP64 compute performance up to 35%, and an overall performance up to 25%. These results are consistently attained up to 512 nodes, with only a negligible reduction in the two metrics as the node number increases. On the NVIDIA H100, the flux kernel shows a floating-point performance up to 40% of the nominal value, and up to 35% for the overall performance. Also in this case, performance does not significantly vary with the used number of nodes.

ALCF Aurora

We assessed LVR on weak scaling up to hundreds of Aurora nodes. Per node, a grid size of more than 2 billion points, and 1.5 billion points were used, on CPUs and GPUs, respectively. On 512 nodes, a 20.2 PFLOP/s compute performance was achieved. Results in terms of overall fraction of peak is reported in Figure 7.4a. Weak scaling efficiency remains above 92% and 87% for CPUs and GPUs, respectively, even when several hundreds of devices are employed. Performance penalty emerges when the node count further increases, with a sharp drop for the GPU version when going from 64 to 128 nodes. In the dragonfly network topology of Aurora, nodes are divided in fully-connected groups of 64 nodes, and any two groups are connected by 2 links. As the node count increases beyond 64, a performance deterioration is therefore expected.

7.5. Simulation

In this section, we report one simulation that have been carried out using LVR.

In particular, we leveraged LVR to carry out shock tube simulations. We experimented with initial conditions leading to the generation of vorticity. Since LVR does not support viscous fluxes, the only way to obtain vortex generation consists in imposing pressure and density so that their gradients are not aligned. In order to show this, let's start from the Euler momentum equation:

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot (\boldsymbol{u}\boldsymbol{u}^T) = -\frac{1}{\rho}\nabla p$$

The vorticity $\boldsymbol{\omega}$ is the curl of the velocity, that is:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u}.$$

It can be shown that the vorticity evolution equation is:

$$\frac{D\boldsymbol{\omega}}{Dt} = (\boldsymbol{w} \cdot \nabla)\boldsymbol{u} - \boldsymbol{w}(\nabla \cdot \boldsymbol{u}) + \frac{1}{\rho^2}\nabla\rho \times \nabla p$$

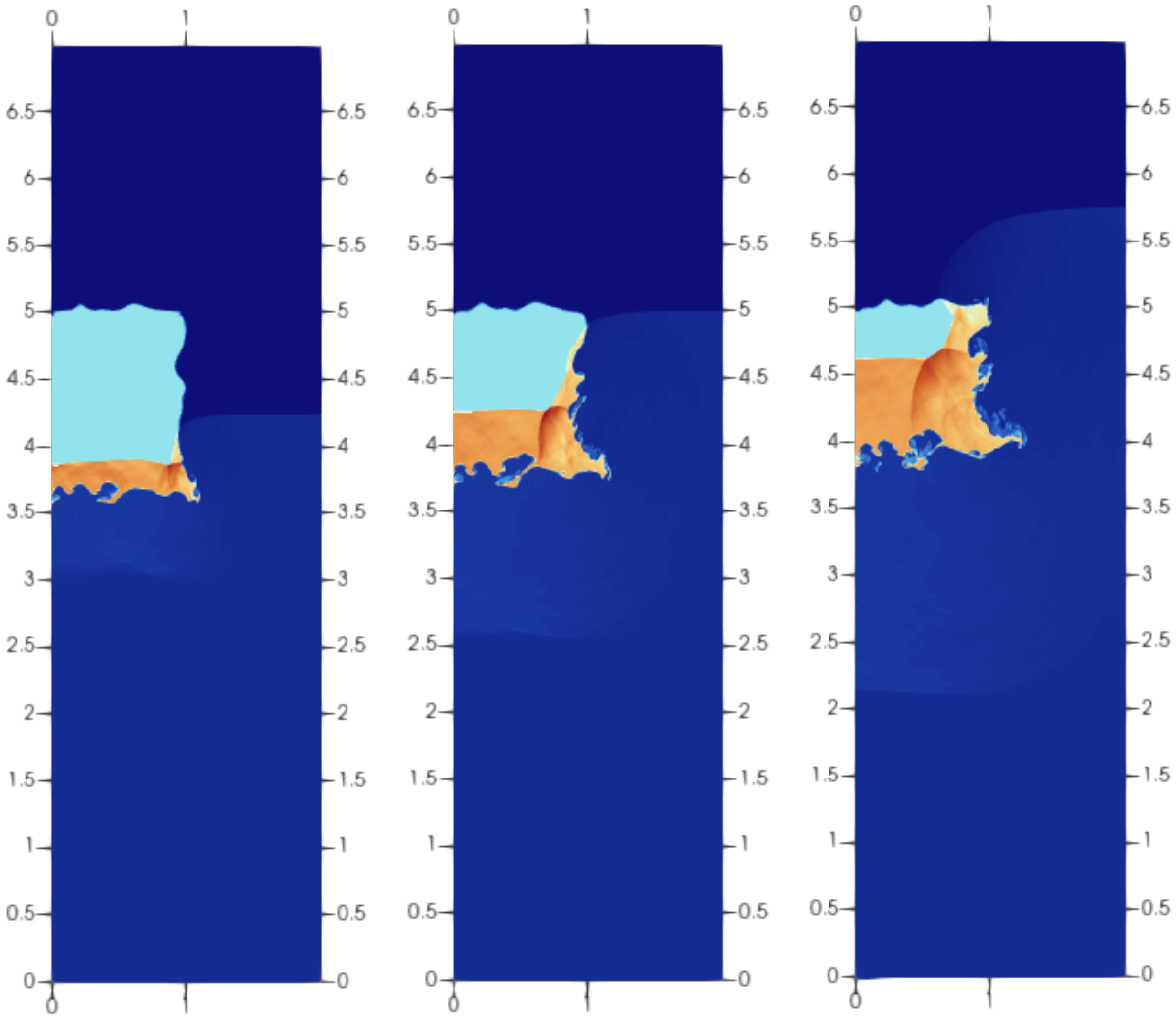
The $\frac{1}{\rho^2}\nabla\rho \times \nabla p$ term is referred to as the baroclinic term, which is the one producing the vorticity. Therefore, if the gradients of pressure and density are aligned, and thus $\nabla\rho \times \nabla p = 0$, no vorticity is produced.

As testbench, we choose a Triple-Point shock scenario, which occurs when three distinct shock waves or discontinuities intersect at a single point: an incident shock, a reflected shock, and a Mach stem. We set up the simulation with the following initial conditions:

- A 3D domain, with physical dimensions 1m x 2m x 7m, and 600 x 1200 x 3080 cells.
- A pressure of 1.68553 Pa for $z < 3.45$, and of 1.0 otherwise,
- Velocity along z equals to 0.4598 for $z < 3.45$, and 0 otherwise,
- Temperature equals to 1.1656 for $z < 3.45$. For $z > 3.45$, the temperature is 1.0, except in the region $y < 1, 3.45 < z < 5$, where it is set to 0.198. For the reasons explained above, the exact boundary of these regions is stochastically perturbed.

Figures from 7.5a to 7.5c show the density field over the middle xy plane at time $t = 2.3s, 4.6s, 6.8s$. It can be noticed the presence of several vortices.

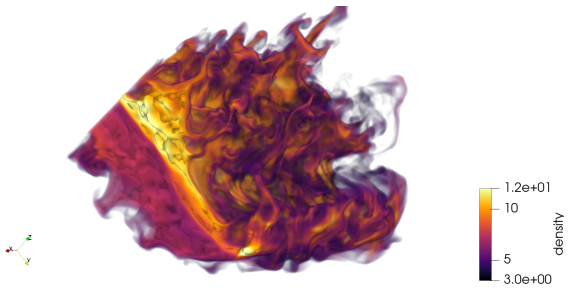
In figures from 7.6a to 7.6f temperature, density and pressure are shown, for $t = 4.62s$ and $t = 7.08s$. Their data range has been restricted so as to highlight the relevant features of the flow fields. Figures 7.6g and 7.6h show the pressure and temperature values on the density isosurface, where $\rho = 3kg/m^3$.



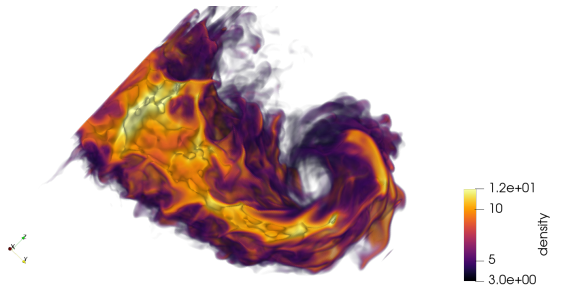
(a) Density field over the middle yz plane at time $t = 0.536s$

(b) Density field over the middle yz plane at time $t = 1.04s$

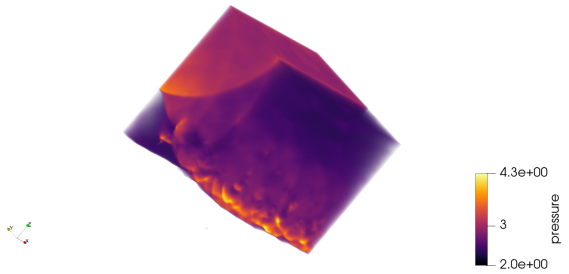
(c) Density field over the middle yz plane at time $t = 1.54s$



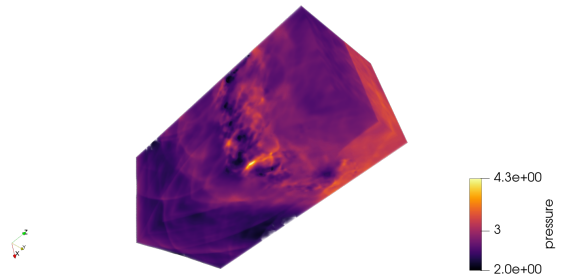
(a) Density at $t = 4.62\,\mathrm{s}$



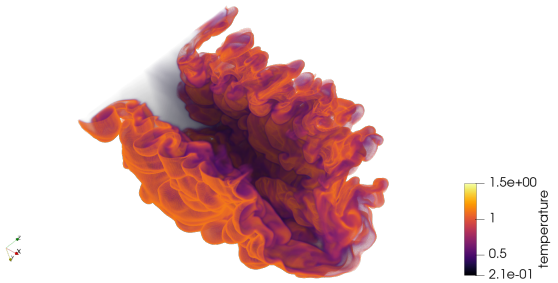
(b) Density at $t = 7.08\,\mathrm{s}$



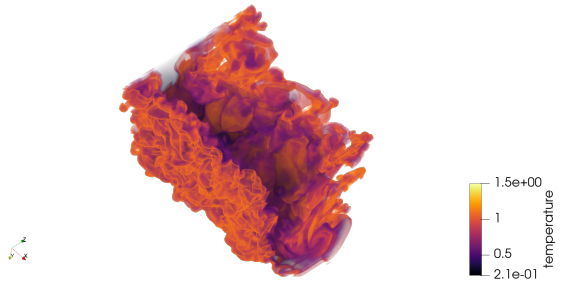
(c) Pressure at $t = 4.62\,\mathrm{s}$



(d) Pressure at $t = 7.08\,\mathrm{s}$

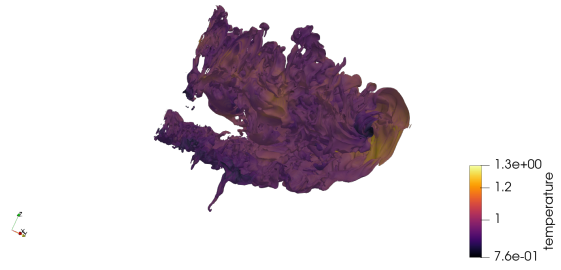


(e) Temperature at $t = 4.62\,\mathrm{s}$



(f) Temperature at $t = 7.08\,\mathrm{s}$

(g) Pressure on $\rho = 3\,\mathrm{kg/m^3}$ at $t = 7.08\,\mathrm{s}$



(h) Temperature on $\rho = 3\,\mathrm{kg/m^3}$ at $t = 7.08\,\mathrm{s}$

Figure 7.6: Density, pressure, and temperature fields at selected times and on a density isoline.

8 | Conclusions

We have presented an assessment of an approach to CFD software development that lies outside common practice. Our solver, built in three months by a team of four, reaches high fractions of peak compute performance across several architectures, including exascale systems. To our knowledge, no prior work has reported comparable results. Moreover, we surpass HTR, an established compressible flow solver, by up to one order of magnitude.

To achieve this goal we employed a new performance modelling technique and analysis techniques, showing how it guided the development process from its earliest stage, resulting in successful design decisions. It is based on microbenchmarks for system performance evaluation, and on a symbolic framework for application performance characterization, such as floating-point operations counting. In particular, we chose the domain decomposition strategy and how to fuse computational kernels following our performance modelling method. Its ability to provide reasonable performance upper bounds estimates, together with its consideration for different data rates (as functions of the associated payloads), makes it an insightful tool that can effectively support HPC application development beyond the specific case herein presented. In [37], it is reported how a subset of the same techniques leveraged for this work have led, for another CFD application, to outperforming factors, with respect to a baseline, up to 16X.

Finally, we have described how a plugin-based architecture can be leveraged to structure HPC applications, while, at the same times, avoiding the potential drawbacks of excessive reliance on Object-Oriented programming. In LVR, it is used to modularly target several HPC platforms, and to enable intra node data sharing through two variants of shared memory.

Moreover, we also showed how the use of symbolic-enabled program synthesis can both enhance performance portability and software productivity. The workflow we propose allows easily implementing intricate computational kernels, thus fostering correctness and making implementing (platform-specific) optimizations straightforward.

Outlook. Future work may include assessing whether refined code generation approaches could further improve exploitation of instruction level parallelism by reducing register pressure and extending our solver to support additional functionality. In particular, in order to represent a potential alternative to HTR, LVR would need to support viscous fluxes and chemical reaction modelling, in addition to multi-species simulations. Moreover, considering the performance modelling results described in chapter 5, assessing other domain decomposition schemes may lead to further performance improvement, particularly in strong-scaling regimes.

Bibliography

[1] S. R. Alam and J. S. Vetter. A framework to develop symbolic performance models of parallel applications. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

[2] P. Batten, N. Clarke, C. Lambert, and D. M. Causon. On the choice of wavespeeds for the hllc riemann solver. *SIAM Journal on Scientific Computing*, 18(6):1553–1570, 1997. doi: 10.1137/S1064827593260140. URL `https://doi.org/10.1137/S1064827593260140`.

[3] M. Bernardini, D. Modesti, F. Salvadore, and S. Pirozzoli. Streams: A high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows. *Computer Physics Communications*, 263:107906, 2021. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2021.107906. URL `https://www.sciencedirect.com/science/article/pii/S0010465521000473`.

[4] S. H. Bryngelson, K. Schmidmayer, V. Coralic, J. C. Meng, K. Maeda, and T. Colonius. Mfc: An open-source high-order multi-component, multi-phase, and multi-scale compressible flow solver. *Computer Physics Communications*, 266:107396, 2021. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2020.107396. URL `https://www.sciencedirect.com/science/article/pii/S0010465520301818`.

[5] K. Cameron, R. Ge, and X. Feng. High-performance, power-aware distributed computing for scientific applications. *Computer*, 38(11):40–47, 2005. doi: 10.1109/MC.2005.380.

[6] K. Cameron, R. Ge, and X. Feng. High-performance, power-aware distributed computing for scientific applications. *Computer*, 38(11):40–47, 2005. doi: 10.1109/MC.2005.380.

[7] Center for Turbulence Research. Proceedings of the summer program 2024: Data-driven methods group overview. Technical report, Stanford University, 2024. URL `https://web.stanford.edu/group/ctr/ctrsp24/ii00_overview.pdf`. Proceed-

ings of the 19th Biennial Summer Program of the Center for Turbulence Research (CTR).

[8] H. Collis, D. A. Bezgin, S. Mirjalili, and A. Mani. A thermodynamically consistent and robust four-equation model for multi-phase multi-component compressible flows using eno-type schemes including interface regularization. *arXiv preprint arXiv:2504.14063*, 2025. URL `https://arxiv.org/abs/2504.14063`.

[9] F. De Vanna and G. Baldan. Uranos-2.0: Improved performance, enhanced portability, and model extension towards exascale computing of high-speed engineering flows. *Computer Physics Communications*, 303:109285, 2024. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2024.109285. URL `https://www.sciencedirect.com/science/article/pii/S001046552400208X`.

[10] F. De Vanna, F. Avanzi, M. Cogo, S. Sandrin, M. Bettencourt, F. Picano, and E. Benini. Uranos: A gpu accelerated navier-stokes solver for compressible wall-bounded flows. *Computer Physics Communications*, 287:108717, 2023. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2023.108717. URL `https://www.sciencedirect.com/science/article/pii/S0010465523000620`.

[11] V. K. Decyk, S. R. Karmesin, A. de Boer, and P. C. Liewer. Optimization of particle-in-cell codes on reduced instruction set computer processors. *Computer in Physics*, 10(3):290–298, 05 1996. ISSN 0894-1866. doi: 10.1063/1.168571. URL `https://doi.org/10.1063/1.168571`.

[12] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2011.

[13] M. Di Renzo, L. Fu, and J. Urzay. Htr solver: An open-source exascale-oriented task-based multi-gpu high-order code for hypersonic aerothermodynamics. *Computer Physics Communications*, 255:107262, 2020. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2020.107262. URL `https://www.sciencedirect.com/science/article/pii/S0010465520300837`.

[14] B. Einfeldt. On godunov-type methods for gas dynamics. *SIAM Journal on Numerical Analysis*, 25(2):294–318, 1988. ISSN 00361429. URL `http://www.jstor.org/stable/2157317`.

[15] Y. Fu, W. Shen, J. Cui, Y. Zheng, G. Yang, Z. Liu, J. Zhang, T. Ji, F. Xie, X. Lv, H. Liu, X. Liu, X. Liu, X. Song, G. Tao, Y. Yan, P. Tucker, S. Miller, S. Luo,

S. Koric, and W. Zheng. Toward exascale computation for turbomachinery flows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092. doi: 10.1145/3581784.3627040. URL https://doi.org/10.1145/3581784.3627040.

[16] N. N. Gibbons, K. A. Damm, P. A. Jacobs, and R. J. Gollan. Eilmer: An open-source multi-physics hypersonic flow solver. *Computer Physics Communications*, 282: 108551, 2023. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2022.108551. URL https://www.sciencedirect.com/science/article/pii/S0010465522002703.

[17] W. Gropp. Performance driven programming models. In *Proceedings. Third Working Conference on Massively Parallel Programming Models (Cat. No.97TB100228)*, pages 61–67, Nov 1997. doi: 10.1109/MPPM.1997.715962.

[18] D. Grove and P. Coddington. Communication benchmarking and performance modelling of mpi programs on cluster computers. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 201–217, April 2004. doi: 10.1109/IPDPS.2004.1303309.

[19] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, Cambridge, MA, 6 edition, 2017. ISBN 9780128119051.

[20] C. T. Jacobs, S. P. Jammy, and N. D. Sandham. Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *Journal of Computational Science*, 18:12–23, 2017. ISSN 1877-7503. doi: https://doi.org/10.1016/j.jocs.2016.11.001. URL https://www.sciencedirect.com/science/article/pii/S187775031630299X.

[21] D. Kaushik, B. Smith, D. Keyes, D. Barry, and F. Smith. Prospects for cfd on petaflops systems. *Parallel Solution of Partial Differential Equations, The IMA Volumes in Mathematics and its Applications*, 120, 03 1998. doi: 10.1142/9789812812957_0060.

[22] S. Kronawitter, S. Kuckuk, and C. Lengauer. Redundancy elimination in the exastencils code generator. In J. Carretero, J. Garcia-Blas, V. Gergel, V. Voevodin, I. Meyerov, J. A. Rico-Gallego, J. C. Díaz-Martín, P. Alonso, J. Durillo, J. D. Garcia Sánchez, A. L. Lastovetsky, F. Marozzo, Q. Liu, Z. A. Bhuiyan, K. Fürlinger, J. Weidendorfer, and J. Gracia, editors, *Algorithms and Architectures for Parallel Processing*, pages 159–173, Cham, 2016. Springer International Publishing. ISBN 978-3-319-49956-7.

[23] Legion Project. Legion web page. https://legion.stanford.edu.

[24] Y. Liang, Z. Cui, K. Rupnow, and D. Chen. Register and thread structure optimization for gpus. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 461–466. IEEE, 2013.

[25] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Comput. Archit. Tech. Comm. Newsl*, December 1995.

[26] J. D. McCalpin. STREAM: sustainable memory bandwidth in high performance computers, 2025.

[27] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, Jan. 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL https://doi.org/10.7717/peerj-cs.103.

[28] G. Mudalige, I. Reguly, S. Jammy, C. Jacobs, M. Giles, and N. Sandham. Large-scale performance of a dsl-based multi-block structured-mesh application for direct numerical simulation. *Journal of Parallel and Distributed Computing*, 131:130–146, 2019. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2019.04.019. URL https://www.sciencedirect.com/science/article/pii/S0743731518305690.

[29] U. of Warwick and U. of Oxford. Cloverleaf: Hydrodynamics mini-app. https://uk-mac.github.io/CloverLeaf/, 2025. Accessed: 2025-08-11.

[30] D. Passiatore, J. M. Wang, D. Rossinelli, et al. Computational study of laser-induced modes of ignition in a coflow combustor. *Flow, Turbulence and Combustion*, 113: 1055–1079, 2024. doi: 10.1007/s10494-024-00575-x. URL https://doi.org/10.1007/s10494-024-00575-x.

[31] D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware software interface: RISCV edition*. 2020. ISBN 9780128203316.

[32] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *The International Journal of High Performance Computing Applications*, 18(1):21–45, 2004. doi: 10.1177/1094342004041291. URL https://doi.org/10.1177/1094342004041291.

[33] A. Radhakrishnan, H. Le Berre, B. Wilfong, J.-S. Spratt, M. Rodriguez, T. Colonius, and S. H. Bryngelson. Method for scalable and performant gpu-accelerated simulation of multiphase compressible flow. *Computer Physics Communications*, 302:109238, 2024. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2024.109238. URL `https://www.sciencedirect.com/science/article/pii/S0010465524001619`.

[34] Regent Project. Regent web page. `:http://regent-lang.org`.

[35] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith. The ops domain specific abstraction for multi-block structured grid computations. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 58–67, 2014. doi: 10.1109/WOLFHPC.2014.7.

[36] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Loop tiling in large-scale stencil codes at run-time with ops. *IEEE Transactions on Parallel and Distributed Systems*, 29 (4):873–886, April 2018. ISSN 1558-2183. doi: 10.1109/TPDS.2017.2778161.

[37] D. Rossinelli, T. Trabacchin, A. Voci, L. Brown, H. Collis, M. Khanwale, T. Zahtila, D. Brouzet, and G. Iaccarino. Crafting software in hpc: a quantitative approach. 2025.

[38] D. H. Rudy and J. C. Strikwerda. A nonreflecting outflow boundary condition for subsonic navier-stokes calculations. *Journal of Computational Physics*, 36(1):55–70, 1980. ISSN 0021-9991. doi: https://doi.org/10.1016/0021-9991(80)90174-6. URL `https://www.sciencedirect.com/science/article/pii/0021999180901746`.

[39] S. Sathyanarayana, M. Bernardini, D. Modesti, S. Pirozzoli, and F. Salvadore. High-speed turbulent flows towards the exascale: Streams-2 porting and performance. *Journal of Parallel and Distributed Computing*, 196:104993, 2025. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2024.104993. URL `https://www.sciencedirect.com/science/article/pii/S0743731524001576`.

[40] C. Schmitt, S. Kronawitter, F. Hannig, J. Teich, and C. Lengauer. Automating the development of high-performance multigrid solvers. *Proceedings of the IEEE*, 106 (11):1969–1984, 2018. doi: 10.1109/JPROC.2018.2854229.

[41] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11:25–33, 1967. doi: 10.1147/rd.111.0025.

[42] E. F. Toro, M. Spruce, and W. Speares. Restoration of the contact surface in the

hll-riemann solver. *Shock Waves*, 4(1):25–34, Jul 1994. ISSN 1432-2153. doi: 10.1007/BF01414629. URL https://doi.org/10.1007/BF01414629.

[43] V. Volkov. Better performance at lower occupancy, 2010. URL https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf. [conference presentation]. GPU Technology Conference (GTC), San Jose, CA, September 20–23, 2010.

[44] V. Volkov. Understanding latency hiding on GPUs, 2016. [conference presentation]. GPU Technology Conference (GTC), San Jose, CA, April 4–7, 2016.

[45] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

[46] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, 2009. doi: 10.1145/1498765.1498785.

List of Figures

List of Tables

List of Symbols

Variable	Description	SI unit
$\boldsymbol{u}$	solid displacement	m
$\boldsymbol{u}_f$	fluid displacement	m

Acknowledgements

I would like to express my deepest gratitude to my advisors, Dr. Diego Rossinelli and Dr. Patrick Zulian, for their invaluable contributions to this thesis. Their constant support, guidance, patience, and expertise made the completion of this work possible and served as a great source of inspiration to me.

Desidero anzitutto ringraziare i miei genitori, mio fratello e i miei nonni per avermi accompagnato in questo percorso, con pazienza, fiducia ed entusiasmo.
Un grazie speciale anche a Melanie per essermi stata accanto in questi anni, sempre credendo in me e dandomi la forza di continuare.
Grazie anche a Viviana, Luca B. e Luca M, per l'amicizia e la spensieratezza.
Desidero infine ringraziare Emanuele, Niccolò e tutti i membri della 26-Gang, per avermi accompagnato in questi anni.

Funding. This investigation was funded by the Advanced Simulation and Computing program of the U.S. Department of Energy's National Nuclear Security Administration via the PSAAP-III Center at Stanford, Grant No. DE-NA0003968. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05- 00OR22725, through allocation MED125. This research also used CSCS Alps supercomputer with the support of the PASC Project "XSES-FSI: towards eXtreme scale Semi-Structured discretizations for Fluid-Structure Interaction," and it was secondarily supported by the Swiss National Science Foundation and the project "Immersed Methods for Fluid-Structure-Contact-Interaction Simulations and Complex Geometries" (nr.: 200021 215627).