D2.4: Multi-fidelity simulations auto-tuning framework

Date: January 13, 2025

Funded by the European Union

This Project has received funding from the European Union's HORIZON Research and Innovation Programme under Grant Agreement number 101096698

Document Identification			
Status	Final	**Due Date**	31/01/2026
Version	1.0	**Submission Date**	31/01/2026

Related WP	WP2	**Document Reference**	D2.4
Related Deliverable(s)		**Dissemination Level (*)**	PU
Lead Participant	ICCS	**Lead Author**	Panagiotis-Eleftherios Eleftherakis, ICCS
Contributors	ICCS KTH TUDelft	**Reviewers**	Sotirios Xydis (ICCS) Konstantinos Iliakis (ICCS) Akshay Patil (TUDelft)

Keywords:
Computational Fluid Dynamics, GPU acceleration, Autotuning, Performance Portability, Porting,

Document Information

List of Contributors	
Name	**Partner**
Panagiotis-Eleftherios Eleftherakis	ICCS
Georgios Anagnostopoulos	ICCS
Konstantinos Iliakis	ICCS
Sotirios Xydis	ICCS

Document History			
Version	**Date**	**Change editors**	**Changes**
0.1	30/7/2024	Panagiotis-Eleftherios Eleftherakis (ICCS)	Final Table of Contents completed and reviewed internally
0.2	6/8/2024	Georgios Anagnostopoulos (ICCS)	Added Chapter 4, initial versions of Chapters 1,2,3,7
0.3	16/12/2025	Panagiotis-Eleftherios Eleftherakis (ICCS)	Added Chapters 5,6 and extended Chapters 1,2,3,7
0.4	20/01/2026	Panagiotis-Eleftherios Eleftherakis (ICCS)	Modified Chapter 4
0.5	22/01/2026	Panagiotis-Eleftherios Eleftherakis (ICCS) Akshay Patil (TUDelft)	Minor changes to improve readability and interpretability of the results.
1.0	31/01/2026	Gerardo Zampino (KTH)	FINAL VERSION TO BE SUBMITTED

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Sotirios Xydis (ICCS)	21/01/2026
Quality manager	Gerardo Zampino (KTH)	26/01/2026
Project Coordinator	Artem Kulachenko (KTH)	28/01/2026

Table of Contents

List of Figures

List of Tables

List of Acronyms

Abbreviation / acronym	Description
CF	ChannelFlow
CFD	Computational Fluid Dynamics
Dx.y	Deliverable number y belonging to WP x
FEM	Finite Element Method
FPGA	Field-Programmable Gate Array
GCD	Graphic Compute Die
GNOPS	Giga Node Operations per Second
GPU	Graphics Processing Unit
HPC	High Performance Computing
LES	Large Eddy Simulation
MPI	Message Passing Interface
OpenACC	Open Accelerators
OpenFOAM	Open Field Operation And Manipulation
PETSc	Portable, Extensible Toolkit for Scientific Computation
RK	Runge-Kutta
SLR	Super Logic Region
SOD2D	Spectral high-Order coDe 2 solve partial Differential equations
SoTA	State of the Art
TGV	Taylor-Green Vortex
UAM	Urban Air Mobility
WP	Work Package

Executive Summary

One of RefMap's [1] objectives is to develop powerful deep-learning models capable of predicting urban flow based on extensive computational fluid dynamics (CFD) simulations. Due to the substantial time requirements of these simulations, the computational power of exascale heterogeneous supercomputer clusters is essential. This task aims to reduce the time required to execute these simulations by accelerating CFD simulators like the Spectral High Order Code SOD2D [2] and OpenFOAM [3] that solve partial differential equations. These simulators constitute the high-fidelity and low-fidelity frameworks utilized in T2.1 and T2.2, respectively. In this deliverable (T2.4), we devise a multi-GPU auto-tuning and performance portability exploration framework to accelerate the runtime performance of both low and high-fidelity simulators, thus enabling fast and accurate multi-fidelity simulations as specified in T2.3. OpenFOAM is the state-of-the art regarding low fidelity CFD simulation and the selection of SOD2D in favor of other state-of-the-art simulators relies on several key points outlined in section 3.5.

To achieve the aforementioned acceleration, a framework capable of autotuning SOD2D on any GPU architecture and system configuration has been developed. A performance portability exploration framework has been further constructed in order to orthogonally examine the impact of application configuration, runtime, software- and hardware- infrastructure parameters. Moreover, we ensure GPU-enabled execution and acceleration of OpenFOAM by integrating the necessary libraries and performing GPU-accelerated runs on a representative case of the TUDelft campus. The strategy employed to complete the task regarding high-fidelity acceleration comprises three key components:

1. **Simulator Profiling:** Profile the target application, in this case, SOD2D, to identify the most time-consuming parts of the code. Given that this type of application is designed to run on High-Performance Computing (HPC) infrastructure, which includes multiple GPUs, profiling is conducted using various example sizes and different system configurations, such as the number of GPUs available.

2. **Simulator Autotuning:** A framework has been developed to modify GPU hardware parameters, such as the number of threads, blocks, and warps. This framework has been integrated with an autotuning library, namely Opentuner [4], to optimise the hardware parameters set by the user, with the goal of minimising execution time.

3. **Cross-Layer Performance Exploration:** A comprehensive performance portability study of SOD2D is performed, motivated by the increasing heterogeneity of modern supercomputing systems. This takes place across a parameter space spanning application configuration, input scale, software and hardware infrastructure as well as

manual memory access optimization parameters.

The profiling and autotuning efforts for the SOD2D simulator have led to significant improvements in speedup across various GPU configurations and functions. On a single GPU system, notable speedups were achieved, with enhancements up to 50% in certain functions. In multi-GPU configurations, the performance gains were even more substantial, reaching up to 625% with four GPUs. These results demonstrate the effectiveness of the autotuning framework in significantly reducing the execution time of extensive CFD simulations. This acceleration is crucial for advancing the predictive capabilities of deep-learning models for urban flow on exascale heterogeneous supercomputer clusters, aligning with RefMap's objectives.

The performance portability study exposed substantial, quantifiable variability across architectures, precision modes, and scale. Across the Ampere and Volta architectures (A100 and V100), ~1.47× variations were observed in FP32 and ~3.05× in FP64 for comparable configurations. On AMD MI250X, the baseline FP32 run was 10.2×, 14.9x slower than V100 and A100 respectively, but splitting the convection kernel and pairing prefetching with preloading improved MI250X performance by 1.84×, cutting down the performance gap to 5.5x, 8.1x with respect to NVIDIA GPUs. On LUMI under weak scaling, FP32 throughput averaged ~22% higher than FP64, while kernel/memory variants still produced best–worst spreads of ~23.8% (FP32) and ~14.6% (FP64) for Channel Flow; We concluded that modeling at scale is needed to avoid throughput penalties upwards of 20% by performance projections across scales.

To also accelerate the low-fidelity branch of the RefMap multi-fidelity pipeline, we enabled GPU execution for OpenFOAM by using tools for the integration of PETSc-based backends and GPU-accelerated algebraic solvers (e.g., AmgX). This is done through a lightweight coupling approach that preserves OpenFOAM’s native discretisation and case setup. By converting OpenFOAM’s sparse LDU matrices to GPU-friendly CSR representations and offloading the dominant pressure-solver workload, the low-fidelity simulator achieves substantial runtime reductions on representative input meshes (TU Delft campus), delivering up to ~9× end-to-end speedup at moderate MPI scaling on a single GPU. This complements the high-fidelity SOD2D autotuning and portability exploration results by enabling faster generation of multi-fidelity datasets and more rapid iteration of urban-flow prediction models on heterogeneous HPC systems.

1.Introduction

1.1 Motivation and Context

RefMap relies on large numbers of CFD runs to produce training data for urban-flow prediction models, with the high-fidelity branch (LES/DNS-like resolution) being particularly expensive. In this setting, the limiting factor is not a single simulation but the total throughput of an entire campaign: several configuration knobs quickly multiply into thousands of runs. Even modest reductions in per-run wall time translate into materially larger datasets, faster iteration on learning pipelines, and the ability to explore a broader portion of the design space within fixed HPC allocations.

This deliverable focuses on performance optimization in the specific regime that matters in the general scope of the project. More specifically, multi-fidelity simulations are targeted, combining high-fidelity LES with fast-paced RANS simulations to achieve the best of both worlds: accurate and runtime-efficient flow predictions. Such workloads are executed on modern heterogeneous HPC systems with a focus on achieving optimal performance through autotuning and performance portability studies. In practice, the same solver must run efficiently across differing GPU vendors, compiler stacks, and node configurations, while also scaling to multi-GPU execution where communication and data-motion overheads become prominent. For SOD2D, the best-performing execution configuration depends on several interacting design choices, including kernel launch structure, OpenACC mapping parameters, memory-access optimization strategy, floating point precision, and the target architecture's register and cache behaviour.

Within this context, auto-tuning is used as a practical mechanism to map a single, maintainable OpenACC-enabled codebase to efficient configurations on each target system, rather than relying on fixed "one-size-fits-all" kernel grid configuration parameters. The broader portability study complements this by exploring optimization transfer between platforms, especially under scaling where lower-scale intuition can be misleading. Finally, because RefMap's pipeline also requires a low-fidelity branch that can be executed rapidly at scale, the deliverable includes GPU-enablement work for representative OpenFOAM workloads to reduce end-to-end dataset generation time in mixed-fidelity workflows.

1.2 Purpose of the document

The shift towards Urban Air Mobility (UAM) introduces new challenges, including the management of complex air traffic, real-time forecasting, and ensuring safe drone operations within urban environments [5]. Additionally, the expansion of airspace leads to increased environmental impacts such as CO2 and non-CO2 emissions and disruptions to wildlife [6]. The societal impact of UAM is significant, as drone flights can degrade local air quality and cause noise-related disturbances for residents. These challenges underscore the need for an aviation business model that promotes a resilient and sustainable multi-scale aviation ecosystem.

To address these challenges, the European Horizon Project RefMap has been established. Its mission is to develop a digital cloud-based service that quantifies the environmental footprints of both airliners and UAV/UAM, optimising air traffic in real-time to minimise the environmental impact on communities. RefMap aims to achieve these goals through advanced AI techniques powering innovative prediction models. Specifically, for UAM, the objective is to establish a data-driven framework enabling real-time airflow predictions. This framework will ensure the safe and socially acceptable operation of UAVs, addressing concerns such as noise, visual pollution, and air traffic density in urban areas. These real-time predictions will be supported by deep-learning models, trained using data generated by Computational Fluid Dynamics (CFD) simulations. Given the scarcity of high-resolution urban airflow data, deep- learning models must rely on CFD simulations that account for turbulence effects to accurately model urban airflow. RefMap proposes a novel approach where predictive models are trained using high-fidelity simulation data, such as Large Eddy Simulations (LES) [7] or Direct Numerical Simulations (DNS) [8]. DNS comprehensively analyses turbulent structures by considering all dependent variables, but faces notable challenges, especially in simulations involving complex geometries and realistic scenarios. The extensive data demands for model training necessitate numerous CFD simulations, which are time-consuming.

To improve the efficiency of generating training datasets, RefMap's strategy involves optimising high-fidelity simulations through Autotuning Techniques and Performance Portability exploration for efficient acceleration. In alignment with RefMap's vision, the strategy includes transitioning from the Nek5000[9] simulation to SOD2D, a promising GPU-enabled solver. SOD2D is capable of effectively handling complex meshes and utilises a CFD algorithm designed for scalable simulation of turbulent compressible and incompressible flows. It employs a spectral formulation of the Continuous Galerkin Finite Elements model for spatial terms in the Navier-Stokes system, coupled with the Entropy Viscosity stabilisation model. This approach aligns with current state-of-the-art practices and recent initiatives aimed at advancing CFD solvers towards exascale computing

capabilities through the integration of GPU accelerators. In addition to the use of the high-fidelity simulator, OpenFOAM is utilized for the low-fidelity branch, a simulator typically parallelizable only through MPI on multi-CPU systems. We employ a set of tools to perform the necessary offloading of the input mesh onto the GPU as well as the conversion of the sparse representation utilized by the OpenFOAM interface to a format usable by NVIDIA's external solver AmgX API, or alternatively, the PETSc framework, providing access to GPU-accelerated solvers and preconditioners. Regarding the high-fidelity simulator, which is the most computationally intensive branch of the multi-fidelity workflow, two separate but orthogonal acceleration techniques are leveraged. Firstly, for a given hardware infrastructure, there is high sensitivity regarding the configuration of the kernel grid and blocks. To this end, an autotuner is integrated with the SOD2D framework, tuning the aforementioned knobs to optimize performance on the platform at hand. Having paired this tool with a cross-layer performance portability exploration framework, in this document, we present results of both analyses individually, both at scale, per hotspot function and at different mesh partitioning configurations.

1.3 Structure of the document

This subsection aims to provide an overview of the document's structure, giving readers and reviewers an understanding of its content before delving into the details. The document is organised into five key chapters:

Chapter 2 reviews literature on accelerating high-fidelity CFD/LES for large training datasets via HPC/MPI scaling and GPU offloading, and juxtaposes this work's solver-aware auto-tuning framework for performance-portable deployment across heterogeneous NVIDIA/AMD single- and multi-GPU systems.

Chapter 3 introduces the various tools and libraries that were employed to build the profiling and optimization framework, highlighting their roles and functionalities while also providing information about the techniques and the CFD simulators employed.

Chapter 4 provides a comprehensive guide on how the profiling and optimization frameworks were implemented, including a case study on two CFD simulation scenarios: TGV (Taylor-Green Vortex) and CF (ChanneFlow). Important achievements are highlighted.

Chapter 5 investigates performance portability in the SOD2D high-fidelity CFD simulator across heterogeneous GPUs and GPU clusters. It motivates performance portability for CFD, then uses a cross-layer methodology to explore efficient configurations.

Chapter 6 presents a non-intrusive GPU acceleration approach for the low-fidelity CFD simulator OpenFOAM applied to the TU Delft campus airflow case. We leverage third-party tools to obtain substantial speedup.

Chapter 7 summarizes the entire project, encapsulating key achievements and findings.

2. Related Work

As detailed in Chapter 1, Section 1.1, a substantial number of LES simulations are necessary to meet the data requirements for model training. To address this challenge and improve the efficiency of generating training datasets, RefMap planning proposes optimization strategies for high-fidelity simulations utilising High-Performance Computing (HPC) techniques and GPU acceleration. This approach aligns with cutting-edge research that employs HPC parallelization methods, such as MPI, for the parallel execution of CFD solvers.

The authors [10] were motivated to address the computational challenges in urban wind simulations caused by turbulence and large domain sizes, which are critical for understanding urban aerodynamics and pollutant dispersion. They employed a scalable domain decomposition method using a 3D incompressible Navier-Stokes solver, the Smagorinsky model for turbulence, and an implicit backward Euler finite difference method, solving the large-scale nonlinear algebraic system in parallel with a Newton-Krylov-Schwarz algorithm. Their results showed accurate reproduction of flow fields compared to wind tunnel data and detailed flow structures in Shenzhen, China, achieving a speedup of 17.5 times on TianHe-2A with up to 8,192 processors and 28 times on Sunway TaihuLight with up to 16,384 processors. This highlights the potential for high-fidelity, large-scale simulations leveraging HPC resources.

The work in [11] aimed to enhance the efficiency of real-time urban air flow simulations, crucial for emergency response and pollutant dispersion analysis. It utilised the lattice Boltzmann method (LBM) on a General-Purpose Graphics Processing Unit (GPGPU), implementing advanced algorithms such as grid refinement and the Esoteric Twist (EsoTwist) pattern. The results demonstrated faster-than-real-time simulation speeds with high accuracy, achieving a speedup of 146 times for coarse-grid resolution and significant speedups of 15 and 40 times for fine and uniform resolutions, respectively. This indicates the method's potential for practical urban air flow forecasting applications. The limitations of conventional CFD tools in modelling urban microclimates due to high computational costs and stability issues were tackled in [12]. CityFFD was developed using a

semi-Lagrangian approach with high-order interpolation schemes and a time-adaptive technique, implemented on GPUs to enhance computing speed and accuracy. The results showed that CityFFD could simulate airflow around buildings and thermal stratifications efficiently and accurately, validated against several benchmark cases, achieving speedups ranging from 2 to 4 times faster than real-time simulations, making it suitable for large-scale urban simulations on personal computers

The growing need for high-resolution and rapid urban microclimate analysis due to increasing urbanisation and its impact on human comfort and air quality was addressed in [13]. A large-eddy simulation (LES) solver was utilized, implemented on a multi-GPU platform using CUDA [14] Fortran and CUDA-aware MPI, optimised with a monolithic projection-based method and staggered time discretization to ensure efficient computation. This technique also incorporated wall-modelled LES and an immersed boundary method to accurately capture complex urban geometries. The results demonstrated significant computational speedup, achieving real-time performance with a 4-metre grid resolution over a 10.49 km² area in Seoul, Korea, and achieving up to 30.93 times speedup using eight NVIDIA A100 GPUs. This highlights the solver's capability to perform high-fidelity, real-time simulations suitable for urban wind pattern analysis and pedestrian comfort assessment.

In [15], the scalability of Nek5000 was improved to fully utilise petascale computing systems for complex fluid dynamics simulations. A strong scaling study was conducted using direct numerical simulations of turbulent flow in a straight pipe across three supercomputers: Mira, Beskow, and Titan. The technique involved assessing the performance of two coarse grid solvers, XXT and AMG, to understand their impact on computation and communication times. The results demonstrated super-linear scaling due to reduced cache misses, achieving a speedup of 2.84 times with XXT on Mira, 2.64 times on Beskow, and 2.53 times on Titan for the largest test case (Re_τ=1000). These findings underscore the solver's capability to achieve high performance and efficiency on modern HPC systems.

In [16], the potential of GPUs to enhance the performance of Nek5000 was explored, a code widely used for simulating incompressible flow in applications such as thermal hydraulics in nuclear reactors and ocean current modelling. OpenACC directives were employed to port NekBone, a simplified version of Nek5000, to GPU systems. This involved optimising key subroutines like ax3D and gs_op by collapsing nested loops and reducing data transfer between CPU and GPU. Results showed a significant performance improvement: the optimised OpenACC version achieved a speedup from 16.27 Gflops on the CPU to 42.67 Gflops on a single GPU, more than twice the performance of the naive

OpenACC implementation. Additionally, they achieved a parallel efficiency of 79.9% on 1024 GPUs on the Titan supercomputer.

Work in [17] was motivated by the need to improve the performance and scalability of the NekCEM code for solving time-dependent Maxwell equations in high-fidelity electromagnetics simulations. It employed an MPI/OpenACC implementation that leverages GPUDirect communication to facilitate direct data transfer between GPUs, bypassing the CPU to enhance data exchange efficiency. The key techniques included optimising element-by-element operator evaluations and developing a GPUDirect gather-scatter kernel for nearest neighbor flux exchange. The results demonstrated significant performance improvements, achieving more than a 2.53 times speedup over CPU-only performance on the Cray XK7 supercomputer Titan using up to 16,384 GPUs for problem sizes up to 6.9 billion grid points [17]. This implementation showcased the potential of GPU-based computing for large-scale, high-order electromagnetic simulations.

The need to optimise GPU kernel performance for tensor-product operations in high-order finite element methods (FEM), which are critical for achieving high computational throughput in simulations involving complex geometries and physics, is addressed through a focus on three tensor-product operations: mass matrix multiplication (BK1.0), stiffness matrix multiplication (BK5.0), and stiffness matrix with de-aliasing integration (BK3.0), implemented on an NVIDIA Tesla P100 GPU. The optimization strategies included reducing global memory accesses, using shared memory, and leveraging register variables, loop unrolling, and data padding. The results demonstrated substantial performance improvements, achieving a speedup of up to 31 times for BK1.0, 6 times for BK5.0, and significant gains for BK3.0, bringing the performance close to the empirical roofline model's theoretical limits, indicating the effectiveness of these optimizations in maximising GPU utilisation for FEM operations [18].

A study aimed to address the performance limitations of high-order spectral element methods in the Nekbone proxy application for Nek5000, particularly focusing on tensor-product operations, which dominate the computational cost. It employed a series of CUDA optimizations, including a 2D thread structure, loop unrolling, shared memory caching, and register usage, to enhance the efficiency of these tensor-product operations on both NVIDIA Pascal (P100) and Volta (V100) GPU architectures. The results demonstrated significant performance improvements, achieving a speedup of 31 times compared to the initial implementation, with the optimised kernels reaching up to 2.5 TFLOPS/s on the V100 GPU. These optimizations brought the performance close to the empirical roofline model, indicating efficient utilisation of the GPU resources for high-order

FEM operations [19].

The authors in [20] aimed to enhance the scalability and performance of the high-order spectral element fluid dynamics solver Nek5000[34] by leveraging GPU acceleration. They implemented OpenACC directives and optimised CUDA Fortran kernels to enable efficient use of GPU resources, addressing the challenges of loop unrolling and reducing data transfers between CPU and GPU. Their results demonstrated substantial performance improvements, achieving a speedup of 3-5 times compared to the CPU version across several HPC systems, including Piz Daint, Longhorn, JUWELS Booster, and Berzelius. For instance, the runtime for 20 timesteps in a turbulent pipe flow simulation at Re_τ = 550 reduced from 43.5 seconds with 64 GPUs to 13.2 seconds with 512 GPUs on the JUWELS Booster system, illustrating the potential for high throughput and scalability on modern GPU clusters.

Reguly and Mudalige [21] frame performance portability in CFD as the central challenge of heterogeneous HPC: highly tuned approaches (e.g., CUDA or architecture-specific OpenMP/offload) can achieve strong single-platform efficiency but often lock codes to one device or force divergent CPU/GPU implementations. They survey separation-of-concerns strategies—C++ portability layers such as Kokkos and RAJA, and CFD-oriented DSLs like OPS/OP2—that express computations via higher-level parallel patterns while delegating mapping and optimization to architecture backends. The review stresses that “runs everywhere” is not enough: portability must be evaluated as *efficiency across a platform set*, motivating metrics such as harmonic-mean performance efficiency and proxy/mini-app studies that reveal remaining gaps due to data layout, loop scheduling, locality, and host–device transfer costs. Overall, these abstractions can narrow the productivity–performance trade-off by improving cross-architecture efficiency without maintaining separate code paths.

While the aforementioned works demonstrate significant progress in accelerating CFD solvers through GPU offloading, MPI-based scalability, and kernel-level optimizations, they primarily focus on architecture-specific performance tuning or manual kernel refactoring for a fixed solver, hardware platform, or execution model. In contrast, the contribution of this work lies in the development of a systematic, solver-aware auto-tuning and cross-layer performance exploration framework that explicitly targets performance portability across heterogeneous GPU architectures and execution scales. Rather than introducing new numerical formulations or rewriting solver kernels in low-level accelerator-specific languages, we build on the existing OpenACC-enabled SOD2D codebase and expose key parallelization and memory-access parameters as a tunable design space. This enables automated exploration across application characteristics

(mesh size, precision, flow configuration), software infrastructure (compiler backends), and hardware platforms (NVIDIA and AMD GPUs, single- and multi-GPU systems). Furthermore, unlike prior studies that report performance results on a single architecture or a narrow set of benchmarks, our work provides a cross-layer analysis that explains why specific optimizations succeed or fail depending on precision mode, kernel structure, memory hierarchy, and scaling regime. Finally, by integrating this framework into the RefMap multi-fidelity workflow and complementing it with GPU acceleration of both high-fidelity (SOD2D) and low-fidelity (OpenFOAM) solvers, this work advances beyond isolated acceleration efforts and contributes a practical, extensible methodology for deploying portable, high-performance CFD simulations on modern heterogeneous supercomputing infrastructures

3. Background

3.1. Computational Fluid Dynamics (CFD) Simulations

Computational Fluid Dynamics Simulations (CFD) play a crucial role in modern engineering and scientific research, offering detailed insights into fluid behaviour and dynamics through advanced numerical methods. These simulations employ various solvers to model and analyse complex fluid flows in diverse applications. One such solver is the Nek5000, which utilises MPI to harness both inter and intra-node parallelism, showcasing impressive scalability [15][11] across thousands of nodes on petascale systems. However, the current version of Nek5000 is not optimised to leverage the capabilities of modern High-Performance Computing (HPC) architectures, which are transitioning from traditional homogeneous systems to modern GPU-accelerated Exascale computing platforms. Therefore, current research efforts are actively focused on transforming solvers like Nek5000 to leverage accelerated technologies, such as GPUs. Several GPU-accelerated solvers [16][17][18] have utilised OpenACC and OCCA [22] for GPU programming. More recent developments [19] employ CUDA for Nvidia GPUs or a hybrid approach combining OpenACC and CUDA [20]. NekRS [23] represents another recent and promising GPU-accelerated solution written in C++/OCCA, utilising just-in-time (JIT) compilation for platform portability. While it offers significant performance improvements in certain scenarios, existing GPU-enhanced tools do not universally support all simulation types and necessitate a tool-specific problem formulation. This can lead to substantial overhead for scientists, requiring months to set up a new mesh for the tool.

3.2. SOD2D

SOD2D [2], a promising new GPU-enabled solver capable of handling complex meshes. SOD2D is a CFD algorithm designed for scalable simulations of turbulent compressible flows, utilising a spectral formulation of the Continuous Galerkin Finite Elements model [24] applied to the spatial terms in the Navier-Stokes equations, along with an Entropy Viscosity stabilisation model. This Fortran-based solver supports MPI execution, enabling it to scale simulations across multiple CPU or GPU-enhanced nodes. GPU acceleration is achieved through GPU-oriented OpenACC directives. SOD2D employs MPI for multi-scale parallel execution on CPU or GPU nodes, with the solver's parallel execution based on partitioning the mesh into tasks and distributing these tasks across different

MPI ranks. Before runtime, the grid points of the mesh are divided into a connectivity matrix, indicating the distribution across ranks. Special synchronization and attention are required for communication between ranks, computing boundaries and periodic nodes. The algorithmic structure for the most computationally intensive functions, such as convective and diffusive kernels, is designed to allocate one mesh element per thread block, with threads within the block managing either nodal or Gaussian quadrature operations. OpenACC directives are strategically placed above relevant loops to facilitate parallelization across nodes and dimensions. [2]

3.3. GPU Acceleration

GPU acceleration has revolutionised computational science and engineering by providing significant improvements in processing power and efficiency. GPUs are designed to handle multiple tasks concurrently, making them exceptionally well-suited for parallel processing workloads. This capability stems from their architecture, which features thousands of smaller, efficient cores that can work simultaneously on different parts of a problem. By leveraging GPU acceleration, applications in fields such as computational fluid dynamics (CFD), machine learning, and data analysis can achieve substantial speedups, enabling more complex simulations, faster data processing, and ultimately, more insightful results. The use of GPU acceleration is especially transformative in high-performance computing (HPC), where it facilitates the execution of large-scale simulations and data-intensive tasks that were previously impractical or impossible with conventional CPU-only approaches.

3.4. Open Accelerators (OpenACC)

Complementing CUDA, OpenACC (Open Accelerators) provides a user-friendly programming standard designed to simplify parallel programming for heterogeneous computing environments, including multi-core CPUs and GPUs. OpenACC uses compiler directives to specify parallelism in the code, allowing the compiler to generate the appropriate parallel code for the target architecture. This approach is particularly beneficial for scientists and engineers who may not have extensive experience with parallel programming, as it enables them to accelerate applications without deep knowledge of the underlying hardware. By incrementally placing directives in existing code, developers can optimise their applications, harnessing GPU acceleration to achieve performance improvements in a straightforward and portable manner. OpenACC thus bridges the gap between high-level programming ease and the computational power of GPUs [25].

3.5. Design Space Exploration

Design space exploration involves systematically analysing various configurations of a system to identify optimal settings that maximise performance, efficiency, or other key metrics. In this context, OpenTuner, an autotuning framework, is utilised. It is an autotuning framework designed to automatically find the optimal configuration of parameters for a given program or system. It works by exploring a vast space of possible parameter settings through an iterative process. OpenTuner employs various search techniques, including machine learning algorithms, evolutionary strategies, and multi-armed bandit approaches, to efficiently navigate the configuration space. By running multiple configurations and evaluating their performance, OpenTuner identifies the best settings that maximise the desired metrics, such as execution time, accuracy, or resource utilisation. This approach allows for significant performance improvements without extensive manual tuning.

3.6. Performance Portability

Performance portability refers to the extent to which an application can achieve high and consistent performance across a defined set of hardware platforms using a single, maintainable implementation. It is distinct from portability in the functional sense, because it includes an explicit requirement that the code attains good efficiency on each target architecture, relative to that platform's capabilities for the same workload. This requirement is non-trivial because modern HPC systems are increasingly heterogeneous—pairing many-core CPUs (with wide vector units) and accelerators such as GPUs—so differences in parallel execution models, memory hierarchies, and data-movement costs strongly influence the optimal choice of parallel granularity, data layout, and memory-access patterns. Empirical studies show that the performance impact of compiler optimisations can vary sharply across architectures, applications, and even inputs, with the same transformation yielding substantial speedups in some environments and significant slowdowns in others [26]. Consequently, performance-portable design aims to separate algorithmic intent from hardware mapping decisions, so that platform-appropriate choices for scheduling, memory placement, and locality can be applied without proliferating divergent code paths. In addition, performance portability is commonly treated as a measurable property over a platform set: it combines (i) portability as the ability to execute correctly on all platforms in the set and (ii) performance efficiency on each platform (e.g., relative to peak capability or best-known performance), aggregated in a way that penalizes low-efficiency outliers and yields zero when any platform is unsupported. [27]

3.7. OpenFOAM

OpenFOAM complements the high-fidelity branch of REFMAP, and together, they constitute the multi-fidelity CFD pipeline. It is another widely adopted open-source CFD platform, valued for its extensive solver ecosystem and case workflow, but its core design remains largely CPU-centric: production deployments typically rely on MPI domain decomposition (and, depending on build/configuration, limited shared-memory parallelism), while end-to-end GPU acceleration has historically been fragmented and uneven. As also noted in prior CFD portability surveys, GPU support in mainstream OpenFOAM usage has been sporadic and often not treated as a first-class, officially supported execution path, which makes “single-codebase, high-efficiency” performance portability difficult in practice. Recent efforts instead tend to target selective acceleration points—most commonly the linear-algebra/pressure-solver stack via external libraries (e.g., PETSc-based integrations that can exploit GPU-enabled backends)—or experimental refactorings and forks that offload parts of the compute loops using CUDA or OpenMP target offload. Very recent research prototypes (e.g., SPUMA) explicitly pursue a more performance-portable approach by making minimally invasive changes that better support heterogeneous CPU–GPU execution without committing to a single vendor technology, but these remain emerging rather than the de facto standard toolchain for general OpenFOAM workloads.

3.8. Use of SOD2D vs alternative SoTA CFD simulators

Apart from SOD2D, other SoTA simulators exist in the literature, with Nek5000[9] being one of the most competitive ones. The transformation of existing Nek5000 input cases to SOD2D is feasible and technically straightforward. Both solvers are based on a spectral element discretization, which ensures that the underlying mesh design philosophy is equivalent. In practice, meshes can be generated using standard tools such as gmsh [28], followed by solver-specific preprocessing scripts to translate the mesh and boundary condition definitions into the required input format. This workflow is already well established in both communities and does not represent a methodological barrier. As a result, existing Nek5000 benchmark cases can be migrated to SOD2D with limited overhead, preserving mesh quality, polynomial order, and geometric fidelity.

While Nek5000 has historically been regarded as the reference implementation within the Nek framework, NekRS[23] has now reached a level of maturity that is close to parity with Nek5000 for a broad range of incompressible and weakly compressible flow applications. Importantly, SOD2D is a European-developed solver which facilitates access, licensing, and direct interaction with the developer team. This proximity enables faster iteration, tailored feature development, and more efficient debugging cycles, which is particularly valuable for research-driven projects. From a practical standpoint, the combination of comparable numerical capabilities and closer developer access makes SOD2D a robust and strategically advantageous alternative.

A systematic, case-by-case runtime comparison between Nek5000 and SOD2D on identical input configurations is not currently available. However, it is important to emphasize that the two solvers target fundamentally different hardware paradigms: Nek5000 is optimized for CPU architectures, whereas SOD2D is designed for GPU execution. As a consequence, direct wall-clock comparisons are not always meaningful. That said, GPU-accelerated solvers such as SOD2D typically achieve substantially higher throughput and energy efficiency for equivalent spectral element workloads, particularly for large-scale, high-order simulations. Based on existing experience with GPU-based spectral element solvers, SOD2D is expected to be more efficient in general, especially for production-scale simulations where hardware acceleration can be fully exploited.

4. SOD2D Auto-Tuning[1]

4.1. Methodology

This section outlines the RefMap methodology for effectively deploying LES simulations on heterogeneous supercomputing clusters. Our approach utilises the baseline architecture-agnostic GPU acceleration capabilities of the existing SOD2D solver profiles them, and enhances them further by integrating an autotuning algorithm. This ensures portability and highly efficient code across various systems. OpenACC pragmas can be found in various places in SOD2D, starting from the initialization phase to the time integration loop. Depending on their placement, they have a different impact on the parallelization and overall performance of SOD2D.

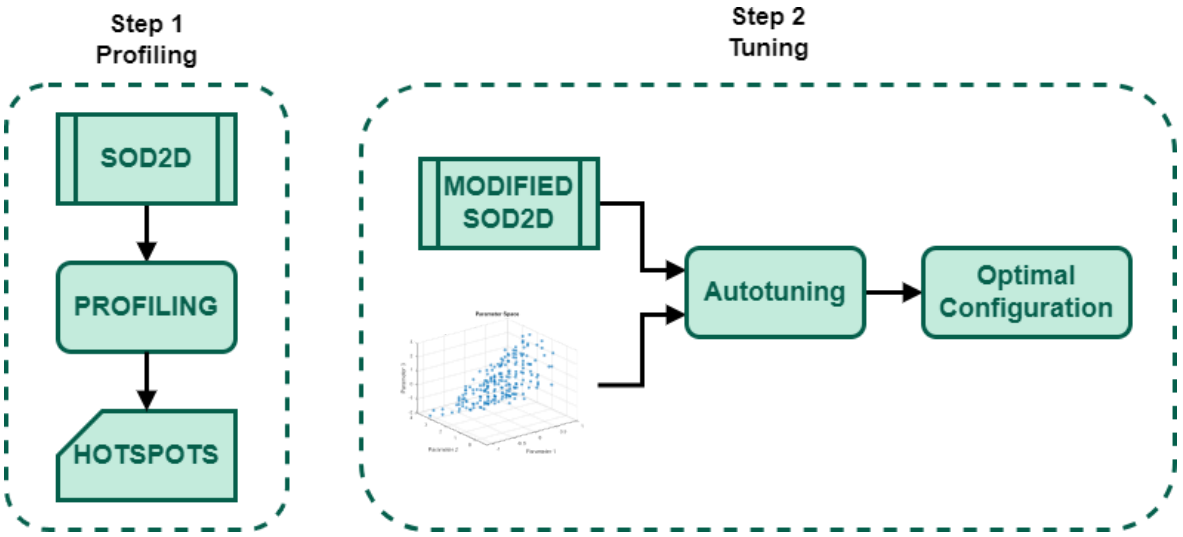

Figure 1: SOD2D Auto-Tuning Methodology

4.1.1. SOD2D Profiling

To gather profiling data across the entire SOD2D codebase, it was essential to develop a profiling framework using the NVIDIA Nsight Systems tool [29]. By profiling the entirety of SOD2D, we collected information from representative benchmark simulations where input data was available. This included the Taylor-Green Vortex simulation, which

[1] Auto-Tuning Framework: https://github.com/GeoAnagn/-RefMap-WP2_CFD_Performance_Autotuning

demonstrates the model's ability to handle both discontinuities and turbulent structures, and the Incompressible Channel Flow simulation, which models fluid flow in a channel with constant fluid density, offering insights into turbulence and flow dynamics. When profiling, data is gathered for every function in the SOD2D code, and the most compute and data-intensive code regions, named hotspots, can be distinguished and passed to the optimization stage. Here, information such as whether a function is CPU or GPU-bound or GPU-related statistics can be parsed.

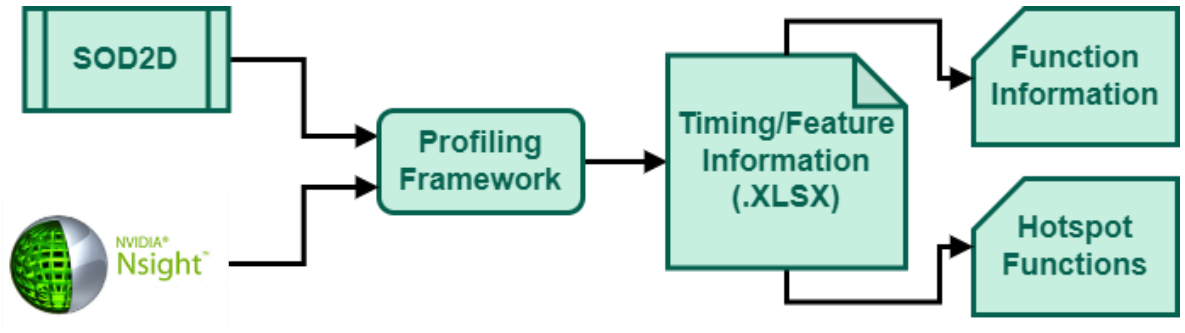

Figure 2: SOD2D Profiling

4.1.2. SOD2D Acceleration

SOD2D strategically places OpenACC pragmas within computationally intensive functions, ensuring adherence to data dependencies and support for parallel execution. These pragmas are primarily positioned above for-loop regions to indicate the level of parallelism by specifying the number of gangs and vector length. SOD2D uses global values for these parameters, applying them uniformly across all pragmas. However, this approach is arbitrary, overlooking the target GPU architecture and heterogeneous configurations for pragmas in different locations. Consequently, the resulting parallelization may not be optimal. The key idea of the proposed methodology is to automatically adjust these values and create a parameter space that better represents the problem and captures the acceleration capabilities. An autotuning framework will be utilised to efficiently explore this search space and achieve a superior, if not global, optimal configuration. This strategy combines algorithmic parallelization with architecture-specific acceleration capabilities, resulting in higher performance. Two main strategies have been employed. The first one explores different pragma values per

execution, but the same for all for-loop regions named coupled strategy. The second one adds an additional layer of complexity where each for-loop region has its own pragma values denoted as a decoupled strategy. The reasoning behind the first strategy is to automatically search for a better pragma value configuration while keeping the original SOD2D parallelization strategy intact. The second one was employed to see what performance improvements may be achieved when each for-loop region has its own parameter space. This means that while the coupled strategy has a smaller exploration space thus it may find an optimal solution quickly; the decoupled strategy fine-tunes each for-loop region, possibly reaching even higher performance improvements. The OpenACC pragmas included in the hotspot functions define the parameter space for the autotuning exploration. These dominant pragmas only include the gang and vector parallelization factors. Based on our decoupling approach, we define a single point in the parameter space as a configuration of gang number and vector length values for all located pragmas in the hotspots. Table I reports the parameter space and the range of values for each one, as well as their place in SOD2D source code. The values are selected based on the minimum number of threads per GPU warp. The searchable solution space, i.e. all possible combinations for the field value, amount to $\prod^{K}(gang\_values_i * vector\_lengths_i)$ configurations, where K is the number of functions to perform the exploration on and gang values, vector lengths are cardinality of the respective sets, i.e. the number of possible values for the number of gangs and vector length, respectively. In our case, this amounts to $250^4 * 20^4$.

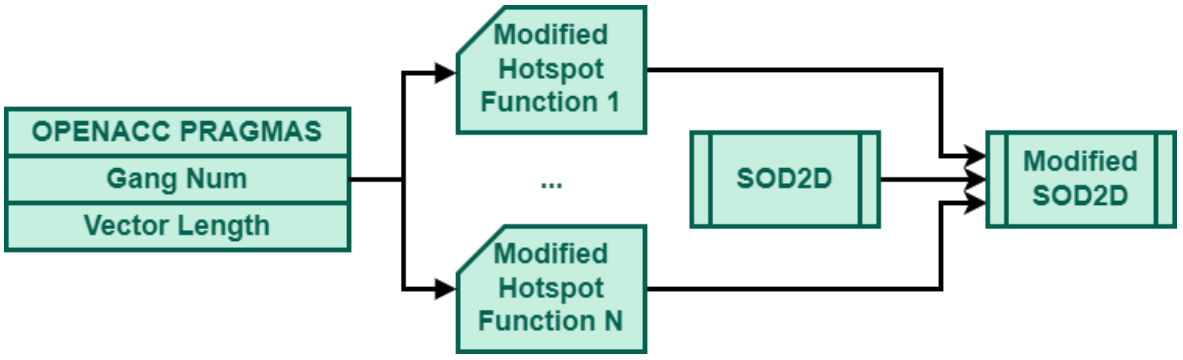

Figure 3: Modified code with OpenACC pragmas

4.1.3. Optimization objective and Exploration Strategy

OpenTuner implements the primary optimization loop by sampling the parameter space in each iteration, executing the chosen configuration on the available infrastructure, and collecting performance metrics, specifically execution time. Based on this metric, the optimizer component evaluates both the previous selections and the most recent one to determine the next configuration. For the optimizer, we use an exploration strategy based on the Multi-Arm Bandit approach, specifically AUC Bandit. AUC Bandit is a multi-armed bandit with a sliding window and an area under the curve credit assignment meta technique. This technique combines the AUC Bandit approach with greedy mutation, differential evolution, and two hill-climber instances to optimise performance.

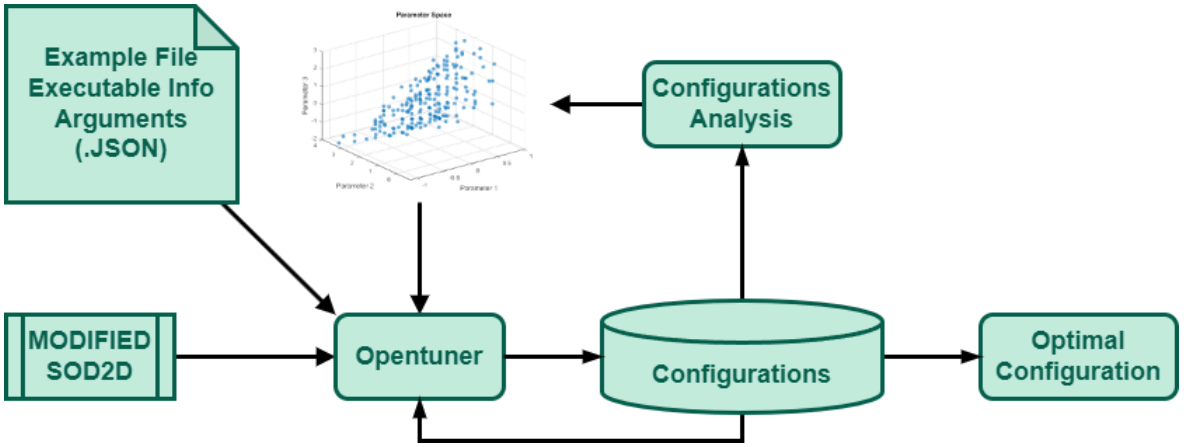

Figure 4: Autotuning Workflow

4.1.4. Simulation Correctness

Simulations for cases such as flow prediction demand very high accuracy and are highly sensitive to changes in computational patterns due to floating-point operations. The proposed methodology investigates multiple configurations that distribute initial computations across varying numbers of GPU blocks or threads per block. This results in altering the execution order of floating-point operations or producing different intermediate values, potentially reducing precision and failing to meet accuracy requirements. To address this challenge, we meticulously examine the accuracy of each configuration by comparing outputs to the reference output of the SOD2D simulation, defining an acceptable deviation of corresponding values greater than 10^{-6}. Additionally, we ensure

result accuracy through a 10-fold validation for the top configurations, averaging output values over 10 executions and comparing them to the reference values.

4.2. Application Characterization

Each time the code is refactored, profiling must be redone to ensure accurate performance metrics and optimization for the new version. Using the March 2023 SOD2D version, the Taylor-Green Vortex example was profiled and accelerated using the SOD2D solver, as a channel flow example was not available at that time. Later, for the SOD2D January 2024 release, both the Taylor-Green Vortex and Incompressible Channel Flow simulations were profiled and optimised on the updated version of SOD2D. However, subsequent information indicated that the Taylor-Green Vortex example was not relevant to simulations run for the REFMAP project. Consequently, in the latest version, only the Incompressible Channel Flow example was optimised to align further with the project's objectives.

4.2.1. TGV Profiling

In the earlier iterations of SOD2D, two input meshes named cube40 and cube80 were provided, differing in size with 64,000 nodes and 512,000 nodes, respectively. Profiling was conducted to analyse the distribution of execution time across various functions (as seen in Figure 5), with results expressed as percentages of the total execution time. For the cube40 mesh, the profiling results were as follows: notably, “full convec ijk” accounted for 18.92% and “full diffusion ijk” for 11.9%. Consistently, the cube80 mesh showed similar execution time distributions for these functions: “full convec ijk” took 22.04% and “full diffusion ijk” took 16.37%, Despite the fact that rk4main emerges as a hotspot for 512,000 nodes, evidently, the same key functions—convection and diffusion—are the primary consumers of execution time, as also observed in Section 5. These profiling results revealed scale-dependent differences in computational load distribution between the two mesh sizes, providing valuable insights for targeted optimization efforts to enhance performance in subsequent versions.

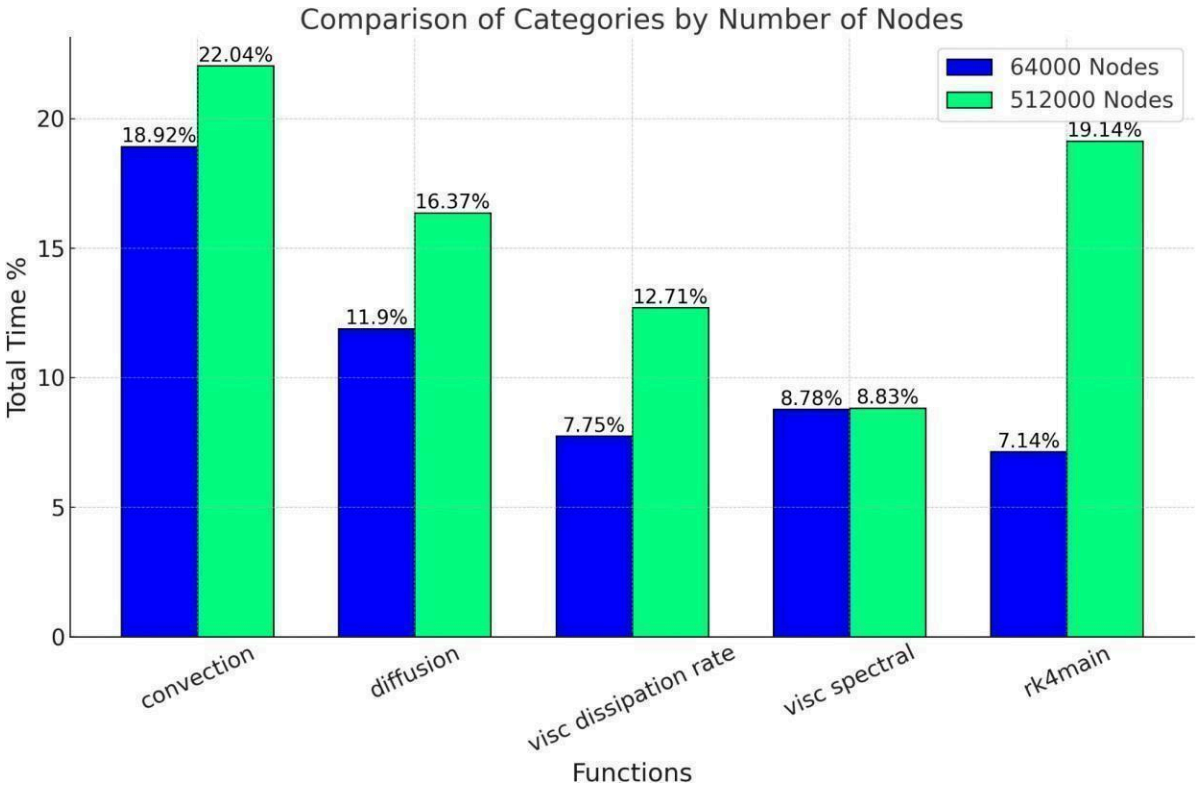

Figure 5: TGV Initial Profiling

In the most recent iterations of SOD2D, input mesh generation scripts were provided for both the Taylor-Green Vortex (TGV) and Incompressible Channel Flow examples, allowing for the creation of meshes with 1, 2, 4, and 8 million nodes. Profiling was executed on all these examples across 1 and 2 GPUs. This comprehensive profiling approach ensured that performance metrics were thoroughly analysed for a range of mesh sizes and GPU configurations, enabling a detailed assessment of computational efficiency and scalability.

Figure 6 presents the execution time percentages for various critical functions in the SOD2D simulator across different configurations categorised by mesh size (1 million, 2 million, 4 million, and 8 million nodes) and the number of GPUs (1 or 2). The functions analysed include char_length_spectral, visc_dissipationRate, full_convec_ijk, full_diffusion_ijk, smart_visc_spectral, write_data_2d, and copy_rcvBuffer.

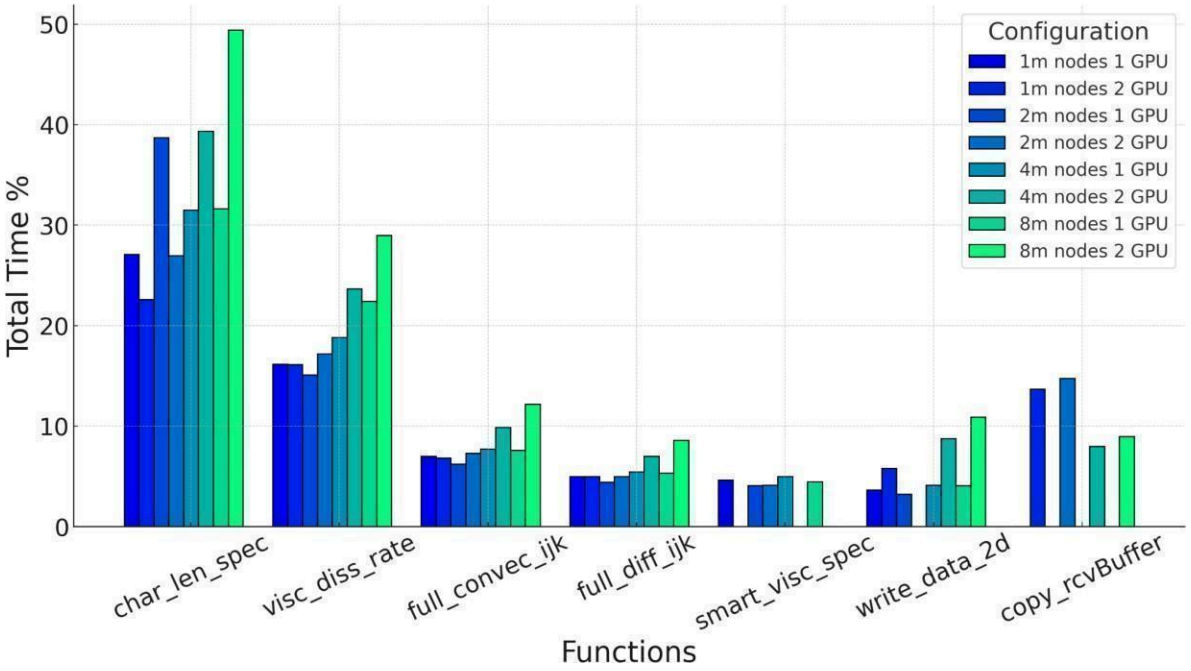

Figure 6: TGV Final Profiling

In the TGV profiling results (Figure 6), the runtime distribution remains concentrated in a small set of routines, but the dominant costs shift as the node count and GPU configuration increase. At 1 million nodes on 1 GPU, char_length_spectral already accounts for about 27% of the execution time, with visc_dissipationRate contributing roughly 16%. As the problem scales up, the spectral characteristic-length computation becomes increasingly limiting: it rises to nearly 39% at 2 million nodes (1 GPU) and reaches its most severe case at 8 million nodes on 2 GPUs, where it grows to approximately 49%, indicating a clear scaling bottleneck. In multi-GPU configurations, a new and non-negligible overhead appears through copy_rcvBuffer, which is around 14% at 1–2 million nodes and remains close to 9% at 8 million nodes, showing that inter-GPU communication is a persistent cost. At the largest scales, data output and handling also becomes more visible, with write_data_2d reaching roughly 11% at 8 million nodes on 2 GPUs.

Despite these newly emerging hotspots, the convection and diffusion kernels (full_convec_ijk and full_diffusion_ijk) remain the most relevant functions at scale because they are consistently present across all cases and form the core per-timestep

workload. Moreover, several of the aforementioned functions can be removed from the workload once correctness has been asserted. Therefore, while optimisation is clearly required for char_length_spectral, visc_dissipationRate, and multi-GPU communication (copy_rcvBuffer) to address scaling limitations for initial runs, sustained performance gains should continue to prioritise convection and diffusion, consistent with the earlier cube40/cube80 profiling where these kernels were already primary time consumers.

4.2.2. Channel Flow Profiling

Profiling of the Incompressible Channel Flow case in SOD2D (Figure 7) was also performed across mesh sizes from 1 to 8 million nodes and using either 1 or 2 GPUs. The goal was likewise, to identify the main hotspot functions and understand how the runtime distribution changes with scale. Across all configurations, a small set of routines repeatedly dominates the execution time: quicksort_matrix, eval_laplacian, conjGrad_pressure, char_length_spec, and—when two GPUs are used—communication routines such as copy_rcvBuffer and fill_sendBuffer. At the largest scale, output overhead (write_data_2d) also becomes visible. A consistent outcome is that quicksort_matrix is the main bottleneck, particularly in single-GPU runs. It consumes about 49% of runtime at 1 million nodes and rises to nearly 59% at 4 million nodes (global maximum), meaning that this function alone can account for roughly half (or more) of the total execution time. When moving to two GPUs, the relative cost of quicksort_matrix drops substantially—typically to around 23–32% (for example, 22.6% at 1 million nodes and 25.6% at 8 million nodes). Even after this reduction, it remains the single largest hotspot, so improving or reducing this sorting cost is likely to have the biggest overall impact.

Alongside quicksort_matrix, two GPU-heavy routines remain consistently important. eval_laplacian typically sits in the mid-to-high teens and reaches around 20% in larger 2-GPU configurations (e.g., approximately 20% at 4–8 million nodes). conjGrad_pressure is also persistent and becomes increasingly significant at scale on two GPUs, reaching about 16% at 8 million nodes. This indicates that the aforementioned functions remain core costs of the simulation across all sizes, and optimising them would benefit both small and large cases.

The main change introduced by using two GPUs is the appearance of communication overhead. copy_rcvBuffer becomes a substantial contributor in multi-GPU runs, surpassing 10% in some cases (around 13% at 2 million nodes and about 12.6% at 8 million nodes). fill_sendBuffer adds further overhead, sometimes reaching 6–7%. These functions do not exist in single-GPU profiling, and their presence highlights that scaling to multiple GPUs is limited not only by compute kernels, but also by data movement

between devices. Finally, char_length_spec is usually secondary at smaller sizes (only a few percent), but it becomes more relevant at the largest meshes, rising to about 13% at 8 million nodes on 1 GPU and remaining close to 9% on 2 GPUs. Output overhead is generally minor, but write_data_2d becomes noticeable in the largest 2-GPU case, reaching roughly 7%, indicating that I/O and data handling can also affect end-to-end performance at scale. As mentioned in subsection 4.2.1, many of the above functions should be optimized for the first runs of the simulation at each configuration, ensuring functional correctness and could be omitted for subsequent runs, rendering the convection and diffusion calculation functions the effective bottlenecks as underlined in Section 5.

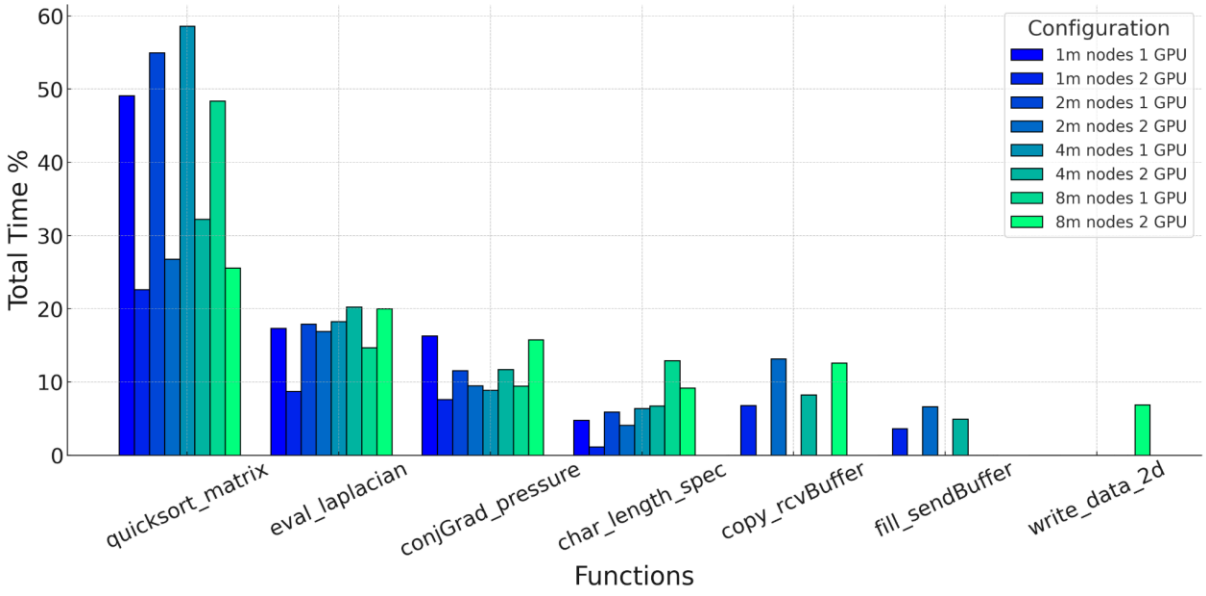

Figure 7: Channel Flow Profiling

4.3. Auto-Tuning Performance Gain

The proposed framework has been deployed on BerzeLiUS [30] supercomputing platform. Each node of the platform is equipped with AMD Epyc 7742 CPUs and 8 NVIDIA A100 Tensor Core GPUs. All modules of our framework are containerized to facilitate portability on any infrastructure and GPU architectures. The container used on BerzeLiUS is built on a base image with CUDA toolkit 11.8 and NVHPC version 23.3. In the case of TGV example, the optimised functions were convection and diffusion terms of Navier Stokes equations and secondarily from the viscosity dissipation rate and smart viscosity spectral functions. In the case of channel flow. More relevant information can be found in the relevant publication [31].

4.3.1. TGV Optimizations

Our experimental campaign aims for a fair and comprehensive evaluation of the proposed methodology. We compare three frameworks the original SOD2D code with default OpenACC pragma values and no auto tuning tool, the tuning decoupled, which implements the proposed methodology and uses OpenTuner after decoupling the OpenACC pragma values, and tuning coupled, an alternative approach that applies OpenTuner over a compact search space by coupling the values for the examined pragmas, i.e., enforcing the same value for all gang numbers and vector lengths for the four hotspots. For both tuning decoupled and tuning coupled, we constrain the examined configurations with a runtime limit of approximately 4 hours. We first evaluate each framework on a single GPU in terms of performance, accuracy, and the number of configurations examined. Accuracy is measured as an error rate, the percentage of output mesh values deviating from the original SOD2D output by more than 10^{-6}. The OpenACC pragma values and evaluations for each framework's configurations are as follows: the abbreviations "ng" and "vl" correspond to the pragmas "num gangs" and "vector length" respectively, with subscripts "fc," "fd", "vdr", and "svs" indicating the hotspot functions. The tuning coupled and tuning decoupled approaches deliver 6% and 15% performance gains, respectively, without any loss in output accuracy. We validate that the optimal solution includes different configuration values for all pragmas. Despite the limited overall speedups, the aggregated time savings are significant, e.g., saving days of computations for simulations running over two weeks that need to be executed multiple times.

Figure 8 compares the performance of two tuning methods, Tuning-Coupled and Tuning-Decoupled, across different functions: full_convec_ijk, full_diffusion_ijk,

smart_visc_spectral, and visc_dissipationRate in a 1-GPU system. The y-axis represents the speedup percentage achieved by each method. For the **full_convec_ijk** function, Tuning-Coupled achieved a speedup of 37%, while Tuning-Decoupled achieved a higher speedup of 48%. This indicates that for this function, the Tuning-Decoupled method significantly outperforms the Tuning-Coupled method, making it more effective for optimising performance. In the **full_diffusion_ijk** function, Tuning-Coupled speedup was 15%, whereas Tuning-Decoupled's speedup was higher at 25%. Both methods show positive speedup, but the decoupled method again proves to be more efficient in this context, demonstrating its general advantage in handling diffusion-related calculations. With the **smart_visc_spectral** function, Tuning-Coupled exhibited a negative speedup of -5%, indicating a performance degradation, while Tuning-Decoupled achieved a speedup of 20%. This stark contrast highlights a potential issue or inefficiency with the coupled method in this specific function, while the decoupled method still provides a significant performance boost. For the **visc_dissipationRate** function, Tuning-Coupled achieved a speedup of 35%, and Tuning-Decoupled reached 50%. Both methods show substantial speedup, but the decoupled method outperforms the coupled method once more, confirming its effectiveness in optimising performance for this function.

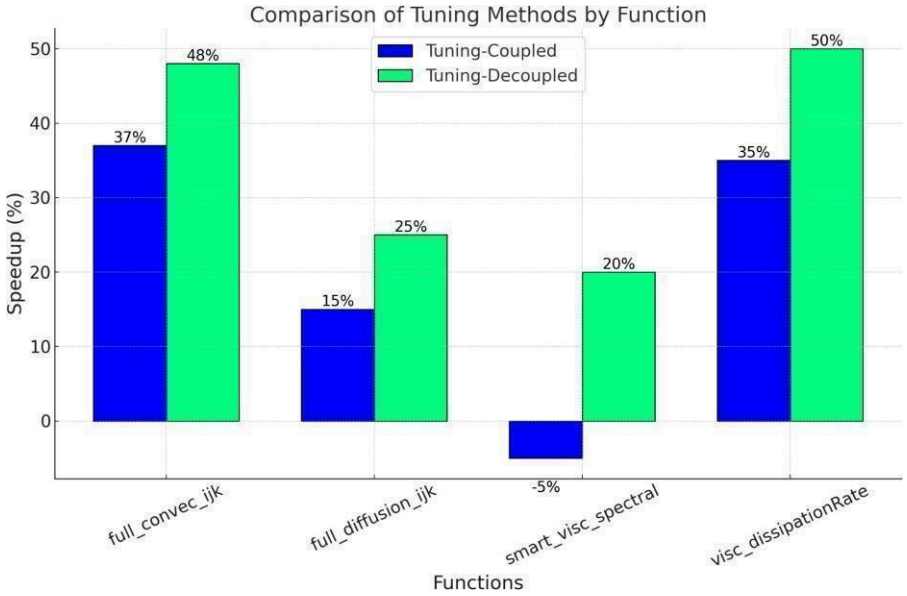

Figure 8: Speedup Evaluation per function - TGV

These findings indicate that the Tuning-Decoupled method consistently outperforms the Tuning-Coupled method across all tested functions, suggesting its broader applicability and effectiveness in optimising various computational tasks. The Tuning-Coupled method, while generally effective, shows limitations, particularly highlighted by its performance degradation in the smart_visc_spectral function. Overall, the decoupled

method demonstrates higher efficiency and better resource utilisation, making it a preferred choice for achieving significant speed-ups in a 1-GPU system. Figure 9 shows the comparison of the performance of two tuning methods, Tuning-Coupled and Tuning-Decoupled, across different GPU configurations (1, 2, 4, and 8 GPUs). The y-axis represents the speedup percentage achieved by each method. In the **1-GPU configuration**, Tuning-Coupled achieved a 7% speedup, while Tuning-Decoupled achieved a 18% speedup. This indicates that with a single GPU, the Tuning-Decoupled method significantly outperforms the Tuning-Coupled method, suggesting that decoupled tuning is more effective at maximising performance in this configuration. In the **2-GPU configuration**, Tuning-Coupled speedup increased to 14%, whereas Tuning-Decoupled's speedup slightly decreased to 16%. Both methods show improved performance with two GPUs compared to a single GPU. However, the gap between the two methods narrows, indicating that coupled tuning starts to become more competitive as more GPUs are utilised. With the **4-GPU configuration**, Tuning-Coupled's speedup increases dramatically to 625%, and Tuning-Decoupled's speedup also rises significantly to 569%. Both methods exhibit exponential improvements in speedup with four GPUs, suggesting a synergistic effect where adding more GPUs exponentially enhances the parallel processing capabilities. Although the coupled method shows a slightly higher speedup, both methods perform exceptionally well at this configuration. In the **8-GPU configuration**, Tuning-Coupled achieves a speedup of 181%, while Tuning-Decoupled reaches 151%. The speedup gains are lower than in the 4 GPU configuration, indicating potential diminishing returns when scaling beyond four GPUs. This could be due to overheads or inefficiencies in utilising a higher number of GPUs. Nonetheless, the Tuning-Coupled method maintains a higher performance.

Both tuning methods exhibit positive scalability with an increasing number of GPUs, but the efficiency gain diminishes beyond a certain point, particularly noticeable when moving from 4 to 8 GPUs. Initially, the Tuning-Decoupled method is more effective, especially for lower GPU counts (1 and 2 GPUs). As the number of GPUs increases, the Tuning-Coupled method gains a competitive edge. The most significant performance gains are observed with four GPUs, suggesting an optimal configuration for balancing speedup and resource utilisation. However, the performance benefits start to plateau or decrease slightly beyond four GPUs, indicating that simply adding more GPUs does not linearly translate to higher performance and may require more sophisticated optimization techniques to manage overheads. In conclusion, both tuning methods are effective, but their relative performance varies with the number of GPUs used. For configurations beyond two GPUs, the Tuning-Coupled method shows slightly better results, while the Tuning-Decoupled method is advantageous at lower GPU counts.

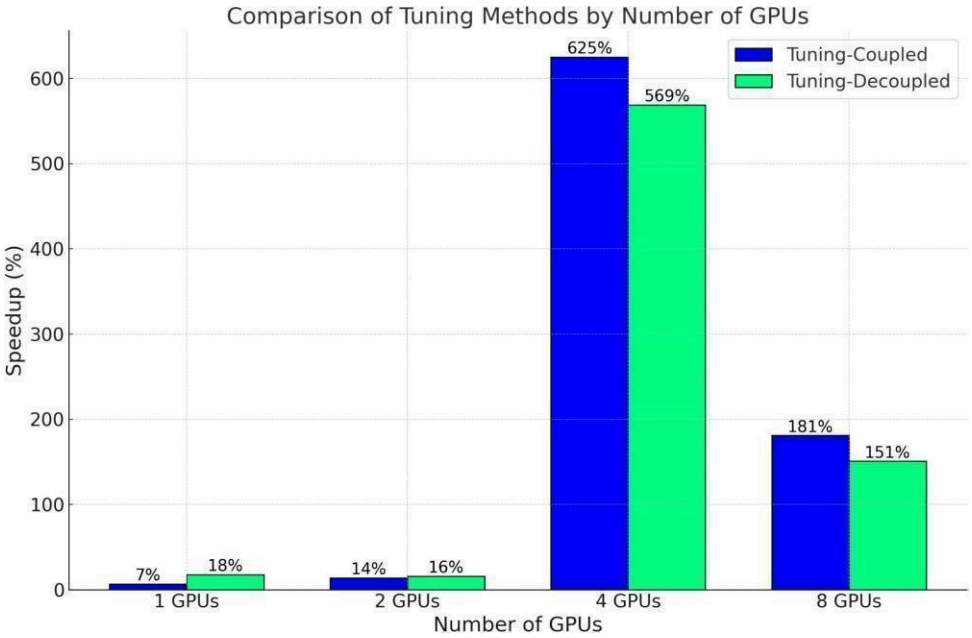

Figure 9: Speedup Evaluation Multi-GPU - TGV

4.3.2. Channel Flow Optimizations

In the new version of SOD2D, we did not rerun the optimization for the Taylor-Green Vortex (TGV) example, focusing solely on the incompressible channel flow. We applied the optimization tool to the examples with 4 million and 8 million nodes. For each example, 150 configurations were tested, and the total optimization process took approximately 10-12 hours. Evaluations were conducted on systems with 1 and 2 GPUs. Accuracy was measured as an error rate, defined as the percentage of output mesh values deviating from the original SOD2D output by more than 10^{-6}. We compare the original SOD2D execution time with the results generated by the decoupled optimization strategy.

In analysing the top 10 configurations for each example, we observed the following average configuration parameter values and execution times. Denoting as *"ng"* and *"vl"* the referring to pragmas *"num gangs"* and *"vector length",* and the abbreviations *“fdijki”, “elm”, “cgvi”* and *“cgpi”,* the GPU hotspot functions *“full diffusion incompressible ijk”, “eval laplacian mult”, “conjGrad veloc incomp”* and *“conjGrad press incomp”* respectively.

Optimal configurations for the respective OpenACC pragma parameters and the respective average execution times are reported in Table 1.

Configuration	OpenACC Pragma	Values	Time (s)
4m Nodes 1 GPU	*ngfdijki , vlfdijki* *ngelm , vlelm* *ngcgvi , vlcgvi* *ngcgpi, vlcgpi*	[440320, 128] [167936, 160] [178176, 160] [18432, 160]	148.7
4m Nodes 2 GPUs	*ngfdijki , vlfdijki* *ngelm , vlelm* *ngcgvi , vlcgvi* *ngcgpi, vlcgpi*	[217088, 160] [165888, 160] [8192, 96] [14336, 96]	120.8
8m Nodes 1 GPU	*ngfdijki , vlfdijki* *ngelm , vlelm* *ngcgvi , vlcgvi* *ngcgpi, vlcgpi*	[417792, 160] [131072, 160] [149504, 96] [20480, 160]	297.6
8m Nodes 2 GPUs	*ngfdijki , vlfdijki* *ngelm , vlelm* *ngcgvi , vlcgvi* *ngcgpi, vlcgpi*	[77824, 128] [432128, 160] [57344, 96] [36864, 128]	190.8

Table 1: Hyperparameter Configuration - Channel Flow

Following Figure 10, the results from the Channel Flow example indicate substantial performance improvements with the optimised configurations. Specifically, for the 4 million nodes configuration, the optimised settings resulted in a 7.3% gain in performance on a single GPU and an impressive 18.9% gain on two GPUs. Similarly, for the 8 million nodes configuration, the performance gains were 8.4% on a single GPU and 19.4% on two GPUs. These gains highlight the effectiveness of the autotuning approach in optimising the performance of the SOD2D solver across different hardware setups.

The higher gains observed with the two-GPU configurations compared to the single GPU configurations suggest that the autotuning methodology effectively leverages the additional computational resources available in multi-GPU setups. This demonstrates the scalability of the optimization approach and its potential to achieve significant performance enhancements in large-scale simulations.

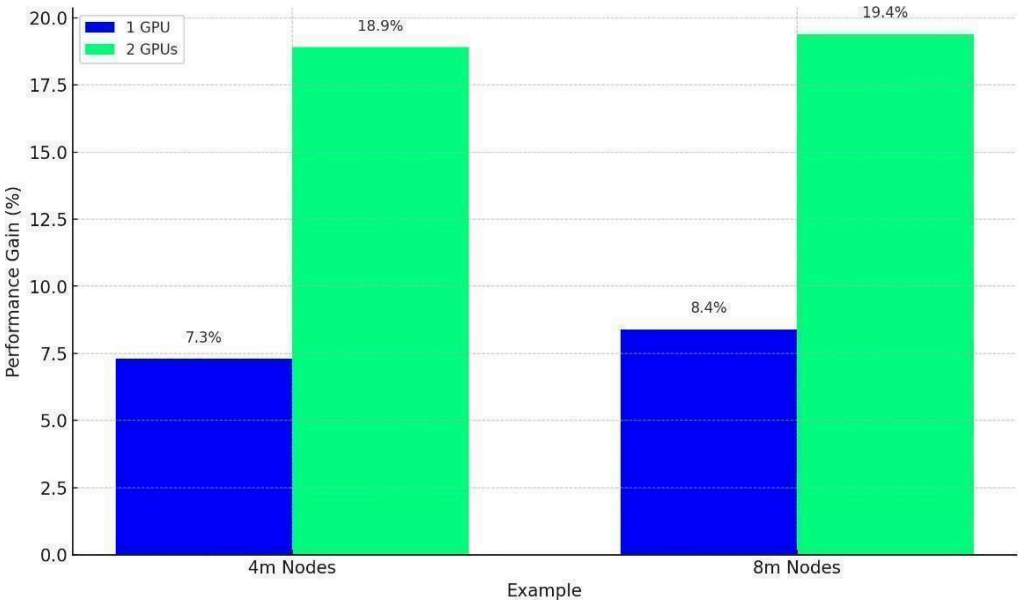

Figure 10: Performance Gain - Channel Flow

The profiling data and configuration analysis provide several key insights into the performance optimization of the SOD2D solver for different node and GPU configurations. Firstly, the most dominant values for the configuration parameters vary significantly across different setups, highlighting the need for tailored optimization strategies for each

specific configuration. The number of gangs and vector lengths play a crucial role in optimising the performance, and finding the right balance between these parameters is essential for achieving minimal execution times. Secondly, the data shows that configurations with multiple GPUs generally achieve lower execution times compared to single GPU setups. This underscores the importance of leveraging multi-GPU systems for large-scale simulations to enhance computational efficiency. Thirdly, the significant variations in execution times among the top 10 configurations within each example indicate that fine-tuning the configuration parameters can lead to substantial performance gains. The dominant values identified in this analysis serve as a valuable reference for setting up optimised configurations for similar simulations in the future.

Lastly, the profiling results emphasise the critical role of certain hotspot functions such as "full diffusion ijk" ("fdijki"), "eval laplacian mult" ("elm"), "conjGrad veloc incomp" ("cgvi"), and "conjGrad press incomp" ("cgpi"). By focusing optimization efforts on these key functions, further performance improvements can be achieved. Overall, this analysis demonstrates the effectiveness of using detailed profiling and parameter tuning to optimise the performance of complex CFD simulations, providing a clear path for achieving high efficiency in computationally intensive tasks.

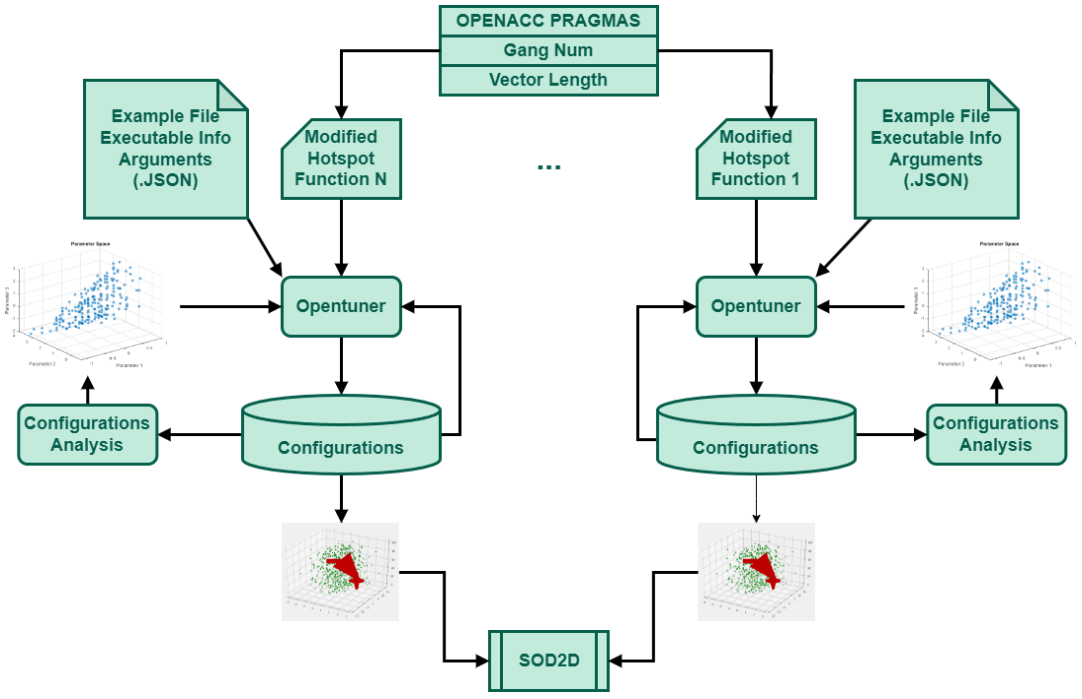

Figure 11: Whitebox Methodology

5. Performance Portability[2] Study of SOD2D

5.1. Motivation

Maintaining performance portability for CFD simulations on heterogeneous supercomputing architectures is essential because the underlying hardware and software environments are both diverse and rapidly evolving, whereas CFD frameworks such as SOD2D are long-lived and costly to develop and maintain. Modern HPC software increasingly relies on diverse GPU-accelerated infrastructure, often comprising different generations and vendors of accelerators (eg, NVIDIA and AMD GPUs) as well as different compiler stacks and runtime environments. At the same time, high-fidelity CFD simulations—especially scale-resolving ones based on high-order spectral finite element methods—are computationally intensive and form a critical building block in multi-fidelity pipelines, including the generation of training data for data-driven or physics-informed models. In this context, a CFD solver that is only well-optimized for a single architecture, precision mode, or problem configuration can become a bottleneck when deployed on a different system.

For SOD2D in particular, performance is shaped by a combination of characteristics that span multiple layers of the stack. At the application level, the solver targets complex, compressible flows using the Spectral Element Method on unstructured meshes, and is designed to handle fundamentally different flow configurations such as Taylor–Green Vortex (e.g. periodicity across all bounds) and Channel Flow. These cases exhibit varying computational behaviour, and their discretization parameters (mesh size, polynomial order, and choice of single or double precision) directly influence arithmetic intensity, memory pressure, and communication patterns. At the kernel level, SOD2D's main hotspots—the convective and diffusive operators implemented in routines such as *full_convec_ijk* and *full_diffusion_ijk*—combine non-trivial memory access patterns (e.g. through connectivity matrices and index linearisation) with element-wise operations over quadrature points. Their performance is therefore highly sensitive to memory hierarchy usage, register pressure and the mapping of OpenACC gangs, workers and vectors to the underlying GPU execution model.

[2] Performance Portability Framework:
https://github.com/GeoAnagn/-RefMap-WP2_CFD_Performance_Portability_Framework
Manual Source Core Memory Optimizations: https://github.com/pelef/sod2d_gitlab

On the software infrastructure side, SOD2D is implemented using OpenACC in order to support multiple back-ends while preserving a single, maintainable code base. However, relying on a directive-based portability layer does not automatically guarantee performance portability: different OpenACC compiler implementations (such as NVHPC for NVIDIA GPUs and Cray Clang on LUMI for AMD GPUs) can generate substantially different low-level code, especially for complex kernels with nested loop optimizations such as preloading of local buffers, or the use of niche OpenACC prefetching directives, thus creating great performance gaps. Choices such as enabling or disabling preloading of element-local data, inserting the aforementioned directives, or restructuring kernels (to explore the tradeoff between kernel launch overheads and register pressure) can drastically shape performance outcomes on varying platforms. Without a systematic view, it is unclear which combinations of these options will be robust across platforms and which will benefit from the individual characteristics of a single compiler or GPU.

At the hardware infrastructure layer, server-grade GPUs from different vendors expose distinct memory hierarchies, register-file sizes and cache structures, even when they offer broadly comparable peak floating-point throughput. In diverse execution environments, performance bottlenecks may shift, and optimisations that improve locality or coalescing on one architecture may interact differently with hardware-managed caches, shared memory usage, or power and frequency limits on another. Moreover, execution at scale (multi-GPU configurations) may introduce additional layers of complexity, where domain decomposition, GPU–GPU interconnects, and MPI communication patterns may affect kernel behavior necessitating the aforementioned analysis.

Taken together, these aspects motivate a cross-layer exploration of performance portability for SOD2D rather than a narrowly focused optimization at a single level. Thus, it becomes possible to reason about how choices at one layer constrain or enable options at another. This cross-layer perspective is necessary to identify optimisations that remain effective across heterogeneous architectures, to understand when architectural differences require different strategies, and to avoid drawing misleading conclusions from experiments that only probe a narrow slice of the space. The performance portability study of SOD2D therefore uses cross-layer exploration as its central methodological tool, with the goal of characterising how the solver behaves across this full stack and of informing future development and deployment of GPU-accelerated CFD simulations on heterogeneous supercomputing platforms.

5.2. Methodology

5.2.1. Code Organization

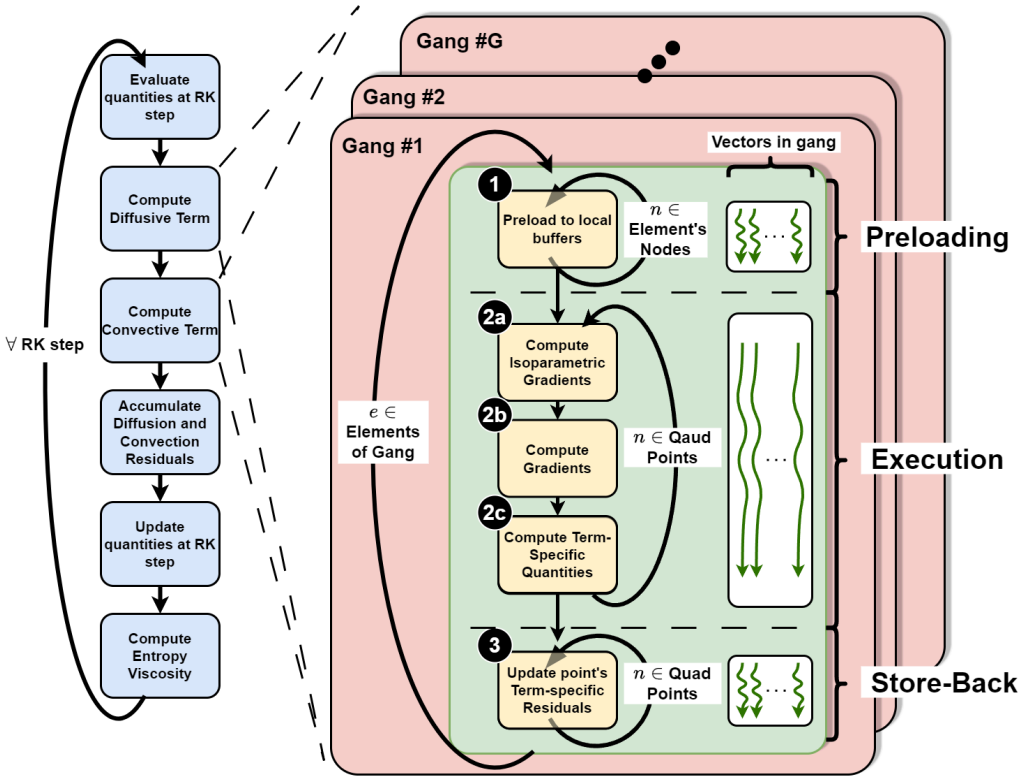

Figure 12: High-level algorithmic overview of SOD2D

Figure 12 shows the execution flow of the main computational kernels in SOD2D, represented by blue rectangles. The algorithm is organized around a loop consisting of four Runge–Kutta steps. During each step, the quantities required for evaluating the Runge–Kutta function are computed. This is followed by calculating the Diffusive and Convective term residuals at every point within each mesh element. Afterwards, the residuals of both terms are accumulated at each node, and the solution values are updated for the current Runge–Kutta stage. Finally, entropy viscosity is evaluated, which is a numerical stabilization approach commonly used in fluid flow simulations, particularly when resolving shock phenomena.

The main focus is on the computation of the Diffusive and Convective terms. This process is separated into three stages: Preloading, Execution, and Store-Back. These stages are handled independently by the available vector gangs, with synchronization points represented by dashed lines in Figure 12. The calculation of each term iterates over all

mesh elements. This element loop is executed in parallel across gangs and vectors, with sizes determined by the kernel configuration.

In step 1, the global quantities associated with the nodes of the active element are loaded into local arrays. This reduces irregular memory access by transforming it into a more structured pattern. Once the required data is available in the local buffers, the main computation begins. First, the isoparametric gradients are computed (step 2a), which are gradients in the natural coordinate system. These are then transformed into gradients in the global coordinate system (step 2b) by multiplying them by the inverse Jacobian. Next, additional term-specific quantities are calculated (step 2c). This procedure is repeated for all quadrature points, which in SOD2D correspond to the nodal points whose values must be evaluated.

Finally, each point's residual contribution for the current term is computed (step 3). Because this operation writes values into the global residual array, it requires multiple atomic updates to maintain correct memory synchronization. These atomic operations serialize access to shared memory and ensure that updates are performed safely.

5.2.2. Hotspots: Convective & Diffusive Operator

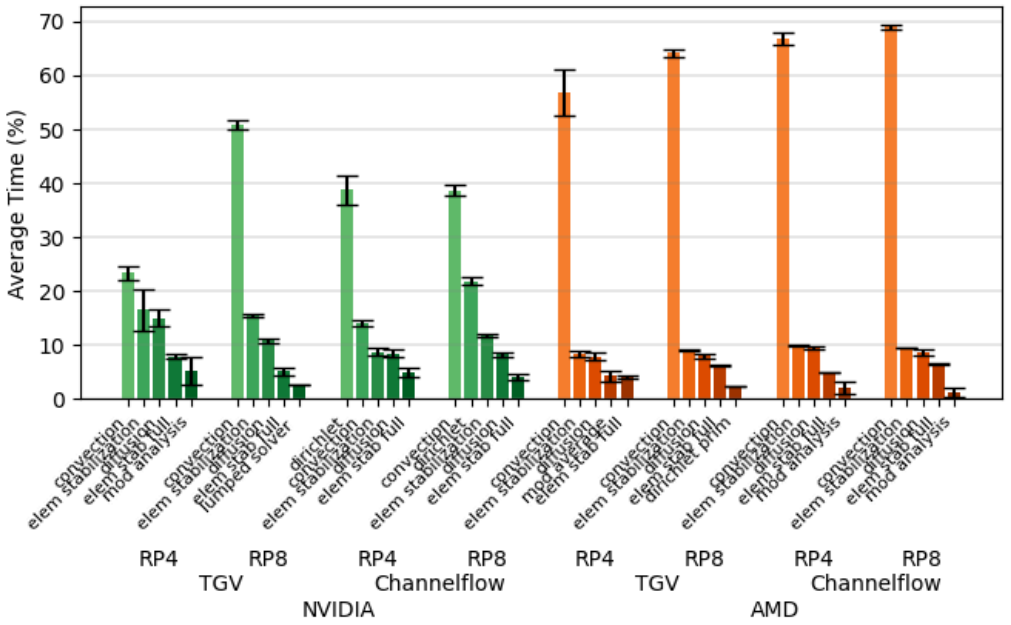

Figure 13: Hotspot Analysis of SOD2D

Figure 13 shows the distribution of the top five execution hotspots for the combination of input Case, floating-point precision, and GPU vendor. The error bars represent the results

across five runs, covering problem sizes from 1 million up to 16 million nodes. The Convection kernel (full_convec_ijk) is consistently the dominant hotspot in SOD2D. The Diffusion kernel (full_diffusion_ijk) is also usually among the top five, except for the Channel Flow case using single precision on an NVIDIA V100 GPU.

The Convection kernel performs worse on AMD GPUs, a behavior explained in more detail later. With respect to floating-point precision, convection consistently performs slower in double precision, but the sensitivity is more noticeable on the NVIDIA V100 architecture. Comparing relative performance between cases, Channel Flow tends to perform better on NVIDIA GPUs, while the Taylor–Green Vortex case performs better on AMD GPUs.

5.2.3. Memory Access Optimizations

```
!========================================
! KERNEL: full_convec_ijk (BASE)
!========================================
do elem = 1, nelem
 ipoin = connec(:,elem)
 reset Rmom_e, Rmass_e, Rener_e

 do igaus = 1, ngp
   reset gradient accumulators
   ! indirect global loads at quadrature
   do ii = 1, npoin
     inode  = ipoin( invAtoIJK(ii,igaus) )
     rho_v  = rho(inode)
     u_v    = u(:,inode)
     q_v    = q(:,inode)
     p_v    = p(inode)
     call assemble_iso_gradients(...)
```

```
  end do



  call compute_global_gradients(...)
  call convection_terms(Rmom_g,Rmass_g,Rener_g,...)
  accumulate R_e
end do

call atomic_add(Rmom,Rmom_e,ipoin)
call atomic_add(Rmass,Rmass_e,ipoin)
call atomic_add(Rener,Rener_e,ipoin)
end do
```

Figure 14: Pseudocode for Full_convec_ijk (Baseline)

The full_convec_ijk kernel is the dominant computational hotspot in SOD2D, and its original implementation evaluates the mass, momentum and energy convection terms within a single launch. All three equations follow the same execution stages already illustrated in Figure 14, which presents the pseudocode for the baseline kernel, and the partial results and final memory writes do not depend on each other.

```
!==================================
! KERNEL: full_convec_mass  (SPLIT)
!==================================



do elem = 1, nelem
 preload rhol,ul
 call quadrature_mass(Rmass_e,rhol,ul,...)
 call atomic_add(Rmass,Rmass_e,ipoin)
end do
```

```
!=====================================
! KERNEL: full_convec_momentum   (SPLIT)
!=====================================

do elem = 1, nelem
 preload rhol,ul,ql,pl,F_l,U_l
 call quadrature_momentum(Rmom_e,...)
 call atomic_add(Rmom,Rmom_e,ipoin)
end do

!==================================
! KERNEL: full_convec_energy   (SPLIT)
!==================================

do elem = 1, nelem
 preload rhol,ul,ql,El,energy_flux
 call quadrature_energy(Rener_e,...)
 call atomic_add(Rener,Rener_e,ipoin)
end do
```

Figure 15: Pseudocode for Full_convec_ijk (Split kernels)

On the AMD MI250X this unified design leads to high register usage per thread, since average vector register allocation is at the device limit and local memory is heavily utilized. To reduce this pressure and simplify kernel behavior, the original kernel was reorganized into three separate kernels, each responsible for one equation system. The pseudocode structure of the split approach is shown in Figure 15, where each variant operates independently and carries only the data structures it needs.

```
!=============================================
! KERNEL: full_convec_ijk_prl   (PRL)
!=============================================
```

```
do elem = 1, nelem
 ipoin = connec(:,elem)
 reset Rmom_e, Rmass_e, Rener_e

 ! NEW: preload nodal values into local arrays
 do inode = 1, npoin
   rhol(inode) = rho(ipoin(inode))
   ul(:,inode) = u(:,ipoin(inode))
   ql(:,inode) = q(:,ipoin(inode))
   pl(inode)   = p(ipoin(inode))
 end do
 call build_local_flux_arrays(F_l,U_l,ul,ql)
 do igaus = 1, ngp
   reset gradients
   ! NEW: quadrature reads only local buffers
   do ii = 1, npoin
     inode = invAtoIJK(ii,igaus)
     call assemble_iso_gradients_using(rhol,ul,ql,pl,inode,...)
   end do
   call compute_global_gradients(...)
   call convection_terms_local_flux(Rmom_g,Rmass_g,Rener_g,...,F_l,U_l)
   accumulate R_e
 end do
 call atomic_add(...)
end do
```

Figure 16: Pseudocode for Full_convec_ijk (Preloading kernel)

Preloading is a commonly used GPU optimization mechanism intended to improve spatial locality, exploit fast shared memory resources and increase effective bandwidth by grouping element data reads. In the baseline version of SOD2D, illustrated in Figure 13, data structures that must be indexed through the mesh connectivity are stored in global device memory and accessed repeatedly inside the quadrature stage. To remove indirect addressing and improve access regularity, these structures are first copied into element-local arrays before the compute phase begins. This approach places frequently used data close to computation and avoids repeated indirect global reads. However, removing preloading and accessing global matrices directly can sometimes show or moderate benefits depending on input and architecture, likely due to the large size of the local arrays. The pseudocode associated with the preloading technique is shown in Figure 16.

```
!============================================================

! KERNEL: full_convec_ijk_prl_prf   (PRL + PRF)

!============================================================




do elem = 1, nelem

 ipoin = connec(:,elem)

 reset Rmom_e, Rmass_e, Rener_e


 ! preload primitives

 preload rhol,ul,ql,pl


 ! NEW: single fused loop builds all local flux arrays

 call build_local_flux_arrays_fused(F_l,U_l,ul,ql)


 do igaus = 1, ngp

   reset gradients

   do ii = 1, npoin
```

```
    inode = invAtoIJK(ii,igaus)
    call assemble_iso_gradients_using(rhol,ul,ql,pl,inode,...)
  end do
  call compute_global_gradients(...)
  ! NEW: convection terms reuse preloaded flux arrays, no recomputation
  call convection_terms_reuse_flux(Rmom_g,Rmass_g,Rener_g,...,F_l,U_l)
  accumulate R_e
end do

call atomic_add(...)
end do
```

Figure 17: Pseudocode for Full_convec_ijk (Preloading & Prefetching kernel)

Prefetching is implemented through the "!$acc cache" directive in the *full_convec_ijk* routine. This directive is tied to a loop structure and informs the compiler that the specified subarrays should be brought into the highest cache level for that loop. When applied directly to large global arrays with indirect access, prefetching provides little benefit. For this reason, the directive is instead applied to the element-local arrays created by preloading. Prefetching every quantity in every nested loop is also ineffective, so the directive is only used for the quantities involved in the deepest nested computations within the quadrature evaluation. The pseudocode representation of this combination of preloading and selective prefetching is displayed in Figure 17.

```
!=============================================================
! KERNEL: full_convec_ijk_reg   (REG)
!=============================================================

! FIRST TIME ONLY: cache indirect indices
if (convec_step == 1) then
 do elem = 1, nelem
   ipoin = connec(:,elem)
```

```
  do igaus = 1, ngp
    do ii = 1, npoin
      connecr(elem,igaus,ii) = ipoin( invAtoIJK(ii,igaus) )
    end do
  end do
 end do
endif
convec_step = convec_step + 1
! REGULAR EXECUTION
do elem = 1, nelem
 preload rhol,ul,ql,pl,F_l,U_l
 do igaus = 1, ngp
   reset gradients
   ! NEW: quadrature uses cached connecr only
   do ii = 1, npoin
     inode = connecr(elem,igaus,ii)
     call assemble_iso_gradients_using(rhol,ul,ql,pl,inode,...)
   end do
   call compute_global_gradients(...)
   call convection_terms_reuse_flux(Rmom_g,Rmass_g,Rener_g,...,F_l,U_l)
   accumulate R_e
 end do
 call atomic_add(...)
end do
```

Figure 18: Pseudocode for Full_convec_ijk (Regularized Access kernel)

Access regularity is another important mechanism for improving effective device bandwidth. In the no-preloading variant of SOD2D, memory coalescing is hindered by irregular access patterns, especially in quadrature statements such as rho(connec(elem, invAtoIJK(ii,j,k))), where the connectivity matrix combines with index linearization. Prior

studies have shown that linearization macros can introduce a significant overhead in HIP-based CFD implementations. Since both connec and invAtoIJK depend only on the static mesh topology, their mapping does not change across kernel launches. To avoid recomputing the linearization and indirect addressing, the combined result of connec(elem, invAtoIJK(...)) is stored during the first launch into a compact lookup matrix indexed by element node, quadrature index and polynomial order. All subsequent launches then access this lookup directly with regular and coalesced addressing. The lookup itself does not make the underlying access to rho regular, since the connectivity is inherently irregular, but it removes the index-linearization overhead entirely. For this approach to remain efficient, the lookup matrices must remain sufficiently small to fit within the lower levels of the memory hierarchy. The pseudocode reflecting this regularized access strategy is shown in Figure 18.

5.3. Cross-Layer Performance Exploration

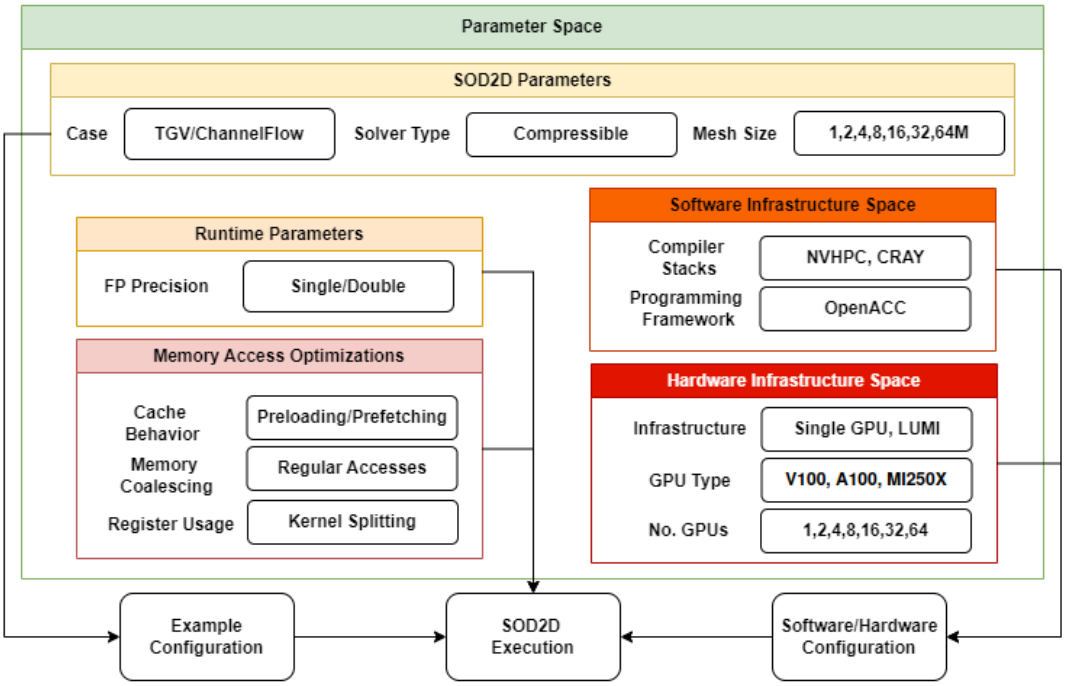

Figure 19: Organization of the examined cross-layer design space.

This work treats performance portability in SOD2D as a cross-layer problem involving the application configuration, the software stack, and the GPU hardware, as seen in Figure 19. A set of parameters is systematically exposed at each layer, and their combined effects on execution behavior are evaluated.

At the application layer, two compressible-flow configurations supported by SOD2D are used: the Taylor–Green vortex (TGV) and plane channel flow (CF). Both solve the compressible Navier–Stokes equations in conservative form for mass, momentum, and energy, with an ideal-gas relation. The spatial discretization employs the spectral element method on unstructured meshes, with high-order shape functions and Gauss–Lobatto–Legendre points collocated with element nodes. This yields diagonal mass matrices and avoids global sparse linear solves, which benefits GPU execution. Time integration uses a fourth-order Runge–Kutta scheme with four stages per time step, where spatial operators are evaluated at each stage. For both benchmark configurations,

unstructured meshes of approximately 1, 2, 4, 8 and 16 million nodes are generated using existing SOD2D tools. Mesh partitioning for multi-GPU experiments is applied after generation, keeping the numerical formulation unchanged while allowing controlled variation in problem size.

At runtime, floating-point precision is treated as a first-class parameter. All configurations are executed in both single precision (RP4) and double precision (RP8). Solver settings, including compressible mode, time-stepping parameters, and Runge–Kutta formulation, are kept constant across all tests so that observed differences result from precision, compiler behavior, memory access, and GPU architecture rather than from changes in the physical model or discretization. This separation allows direct examination of how data footprint and register use interact with other layers of the stack.

Between the numerical formulation and the software infrastructure, SOD2D exposes a set of implementation-level parameters that govern how data are accessed and organized on the GPU. These parameters define alternative versions of the convection and diffusion kernels without altering the underlying equations, discretization strategy, or Runge–Kutta integration. The numerical roles and execution stages of these kernels have been presented in Section 6.4.1, and the different memory access optimizations have been outlined in Section 6.4.2.

The software infrastructure layer remains consistent across platforms. All GPU parallelism and data movement are encoded using OpenACC directives in Fortran. Element and quadrature loops are mapped to the gang/worker/vector hierarchy, which the compiler translates into the appropriate GPU execution model. The same source code and directives are used across architectures, with the only difference being the compiler. On NVIDIA systems, we use NVHPC 24.5, which lowers OpenACC directives to CUDA kernels. On the LUMI system, we use Cray Clang 17.0.1 with OpenACC support targeting AMD's ROCm backend. Compiler choice therefore, becomes an explicit dimension of the design space, allowing evaluation of how different backends handle identical directives and memory patterns.

At the hardware layer we examine two single-GPU platforms and a production multi-GPU setting. Single-GPU tests are executed on an NVIDIA Tesla V100-PCIE-32GB and on one Graphics Compute Die (GCD) of an AMD MI250X-128GB. Each MI250X contains two GCDs, which are treated as independent GPUs for solver execution and MPI placement. Multi-GPU experiments use LUMI's MI250X nodes in a weak-scaling configuration. The global mesh is partitioned into as many subdomains as there are GCDs, assigning approximately one million nodes to each. As the GPU count increases, the global problem

size increases proportionally so that the per-GPU workload remains nearly constant. One MPI rank is bound to each GCD, using the native node topology and interconnect of the LUMI system.

All experiments follow a consistent procedure. For each combination of physical case (TGV or CF), mesh size, precision (RP4 or RP8), memory-access strategy, kernel structure, GPU architecture, and compiler backend, the corresponding SOD2D executable is built and executed. Single-GPU tests are performed on both the V100 and individual MI250X GCDs; multi-GPU weak-scaling tests are performed on LUMI. Each configuration is run for a fixed number of time steps with an initial warm-up period, and repeated runs provide stable means and variability. Timing is recorded for both the overall time-stepping phase and individual kernels. Representative runs are analyzed using vendor profiling tools (NVHPC profilers on NVIDIA, Rocprofv3 and Omnitrace on AMD) to extract metrics including register use, local memory allocation, memory operations, and cache activity. For scalability analysis, we compute a throughput metric, GNOPS (Giga Node Operations Per Second), defined as the total number of mesh nodes across all GPUs multiplied by the number of Runge–Kutta stages and time steps, divided by runtime. GNOPS reflects the uniform node-update structure of SOD2D and provides a meaningful way to compare different configurations.

This methodology defines a structured cross-layer exploration. The physical cases and meshes fix the equations and discretization, precision and solver settings govern runtime behavior, memory-access strategies and kernel structure determine how that behavior is exposed to the compiler, the compiler bridges to the GPU architecture, and single- and multi-GPU deployments provide the environments in which performance and scalability characteristics are measured.

5.4. Single-GPU evaluation

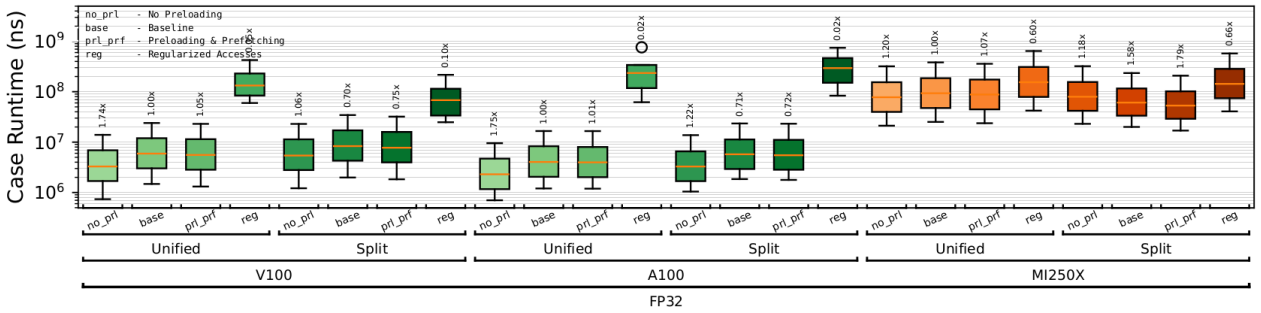

Figure 20a: Single-GPU run times across all kernel optimizations for the Taylor-Green Vortex Case (FP32)

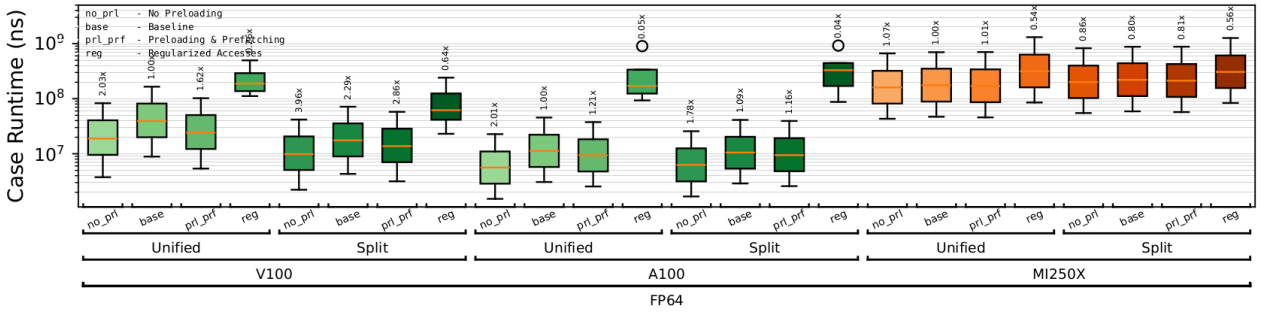

Figure 20b: Single-GPU run times across all kernel optimizations for the Taylor-Green Vortex Case (FP64)

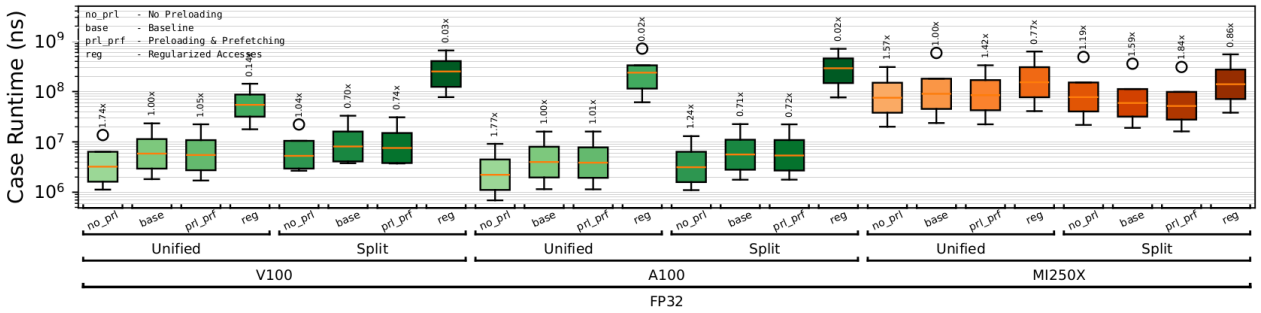

Figure 21a: Single-GPU run times across all kernel optimizations for the ChannelFlow Case (FP32).

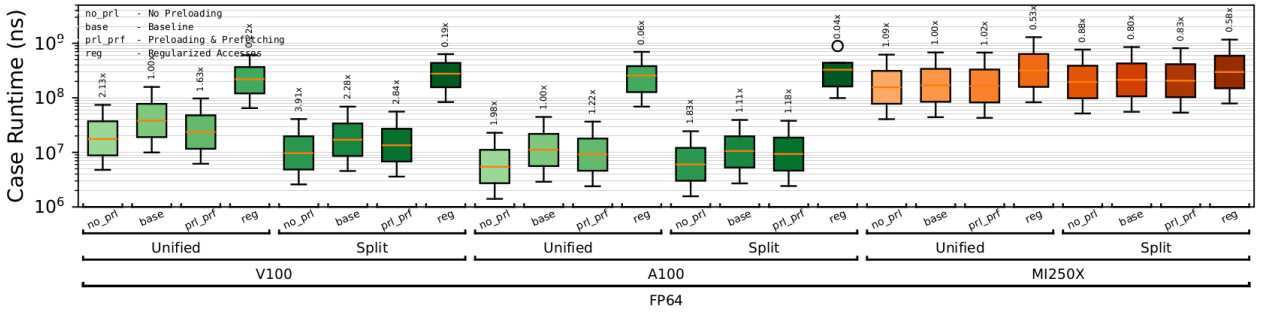

Figure 21b: Single-GPU run times across all kernel optimizations for the ChannelFlow Case (FP64)

The evaluation targets heterogeneous systems using OpenACC and MPI, and measurements are collected on three server-grade accelerators: an NVIDIA Tesla V100, an NVIDIA A100, and a single Graphics Compute Die (GCD) of an AMD MI250X. The goal is to understand how different kernel structures and memory-access strategies behave across GPU architectures and floating-point precisions, and what this implies for performance portability.

The workload used for this evaluation is a turbulence-resolving channel-flow configuration, which is representative of wall-bounded flows relevant for practical applications, as well as the Taylor-Green Vortex case. Mesh sizes span approximately 1, 2, 4, 8 and 16 million nodes. For each mesh, the solver runs with both single precision (FP32) and double precision (FP64) arithmetic, and a fixed number of time steps and Runge–Kutta stages. The software is compiled with vendor-specific OpenACC-capable toolchains: NVHPC on the NVIDIA systems and Cray Clang with ROCm on the AMD system. In this analysis, the primary metric is total wall-clock time for a fixed problem size and iteration count. Relative performance is discussed in terms of speedups between code variants on the same GPU and slowdowns or improvements across GPUs for the same code variant.

Across all GPUs and configurations, profiling shows that the convection operator is the dominating hotspot, accounting for around half the runtime on NVIDIA GPUs and up to roughly 70% on the AMD MI250X. The diffusion operator is consistently the second hottest kernel, but at a much smaller fraction of the total runtime. This stability in hotspot distribution across mesh sizes, precisions and GPU vendors justifies the focus of tuning efforts on the convection kernel, as improvements there directly translate into overall runtime reductions.

We perform a thorough exploration across the whole cross-layer space (Figures 20a-21b). On the NVIDIA Tesla V100, the “no_prl” variant consistently outperforms the “base”

implementation. Removing the explicit preloading phase eliminates the expensive additional pass over the element data and the need to maintain and fill large element-local buffers. The V100's cache hierarchy is sufficiently effective that direct global-memory accesses from the inner loops, with compiler and hardware-managed caching, perform better on average than the preloading-based approach. This behaviour is particularly strong in FP32, where the smaller data size reduces pressure on caches and registers. In FP64, the advantage of "no_prl" over "base" is smaller but still present on average.

Enhancing the preloading strategy with manual prefetching ("prl_prf") improves performance over "base" in several cases. On the V100, for FP64 and the split kernel, prefetching yields around a 14% speedup relative to preloading alone. This makes "prl_prf" a competitive option in contexts where preloading is retained for structural or algorithmic reasons. However, prefetching only helps when it is carefully restricted to small, high-reuse local arrays in the innermost loops; attempts to prefetch large global arrays with irregular access patterns are ineffective or even detrimental.

The "reg" variant, which introduces a larger mapping structure to eliminate index-linearization macros, underperforms on the V100. Although the mapping array itself enjoys more regular, coalesced access patterns, its significantly larger size increases the working-set and strains the cache hierarchy. The cost of moving these larger structures outweighs gains from more regular indexing, and overall runtime increases compared to "no_prl" and even "base". This highlights that regularising access patterns by growing auxiliary data structures can function counterproductively if the additional memory footprint pushes data out of faster memory levels.

The impact of kernel splitting on the V100 depends strongly on floating-point precision. For FP32, splitting the convection kernel into three equation-specific kernels typically degrades performance relative to the unified kernel. The V100 has enough register and scheduler capacity to handle the unified kernel without severe occupancy loss, so the overhead of extra kernel launches and additional global-memory transactions dominates. For FP64, the situation is more nuanced and favourable to splitting: because each thread carries larger FP64 values, intermediate results consume more registers, and reducing the per-kernel working set with splitting can recover occupancy and yield better performance in several configurations.

Moving from the V100 to the A100, the overall performance improves as expected. The A100 provides approximately a 1.47× speedup over the V100 in FP32 and about a 3.05× speedup in FP64 for comparable code variants and problem configurations. Importantly, the relative ranking of the variants remains broadly similar to the V100. Direct global

access without preloading (“no_prl”) continues to be the most robust choice, prefetching on top of preloading (“prl_prf”) provides modest incremental benefits when a preloading design is kept, and the “reg” variant remains non-competitive due to its unfavourable memory footprint. Kernel splitting again tends to be unnecessary or even harmful in FP32, but can be beneficial in FP64 when register pressure is the limiting factor. From the point of view of performance portability within the NVIDIA ecosystem, a reasonable default for this solver is to use “no_prl” with a unified kernel in FP32 and to profile both unified and split configurations in FP64, optionally adding prefetching if preloading cannot be removed.

On the AMD MI250X, the starting point is a much larger performance gap compared to NVIDIA. For average runtimes corresponding to mesh sizes in the range of [1M, 2M, 4M, 8M, 16M] nodes in FP32 with the baseline preloading-based implementation, the MI250X is 14.9x and 10.2x slower than the A100 and V100 respectively. With our refined kernel structure and memory optimization techniques (presented in Section 5.2.3) exploration, these slowdowns were reduced down to 8.1x and 5.5x with respect to A100 and V100 GPUs. Several factors contribute to this performance gap: (i) the GPUs on the evaluated system operate under a reduced power cap (400 W instead of their nominal 500 W), (ii) the OpenACC and compiler support for this specific code path is less mature on ROCm than on the NVIDIA toolchain, and (iii) the significant architectural differences in cache hierarchy, register files and memory system that affect how well the solver maps to this hardware.

Despite the lower absolute performance, the MI250X clearly benefits from kernel splitting, particularly in FP32. Profiling indicates that, with the unified kernel, the vector-register usage and local memory allocation per workgroup are close to their maximum values, heavily constraining occupancy. Splitting the convection kernel into three equation-specific kernels reduces the per-kernel register and local-memory requirements, allowing more wavefronts to reside concurrently and increasing effective occupancy. This yields an approximately 1.5× speedup in the convection hotspot, with a significant impact on overall runtime.

Regarding the memory-access strategies on AMD, the trends partly mirror NVIDIA, but with larger magnitudes. As on NVIDIA, “no_prl” tends to outperform “base” across the board, and the benefit is often more pronounced. The expensive preloading phase, combined with the large local buffers, interacts poorly with the MI250X memory hierarchy, making direct global accesses with hardware caching more efficient in many scenarios. When preloading is retained, adding prefetching (“prl_prf”) can substantially improve performance. In FP32, on the split kernel, speedups of the order of 40–60% over “base”

are observed, making “prl_prf” the best-performing preloading variant. In FP64, the gaps are smaller, but the ordering remains the same: “no_prl” and “prl_prf” with splitting are preferred, while “base” and “reg” perform worse. As in the NVIDIA experiments, the “reg” variant remains consistently unattractive on the MI250X because the inflated mapping arrays exceed the capacity of the fastest memory levels and incur substantial additional memory traffic. The single-GPU study therefore, leads to several general observations on performance portability. There is no single configuration that is optimal across all GPUs and precisions. On NVIDIA FP32, a unified kernel with direct global access is typically best, while on AMD FP32, a split kernel combined with either “no_prl” or “prl_prf” often gives the best trade-off. In FP64, both vendors benefit in some cases from splitting the kernel, but the degree of benefit and the interaction with memory-access strategies differ. Moreover, the same optimization lever can have opposite effects depending on the platform and precision: kernel splitting is beneficial on AMD FP32 but detrimental on NVIDIA FP32, whereas it is helpful on NVIDIA FP64 and more modestly advantageous on AMD FP64. Attempts to regularize memory via large auxiliary structures are consistently harmful, demonstrating that any such “regularization” must be tightly constrained in terms of memory footprint and cache fit.

In summary, the single-GPU evaluation shows that performance portability for this class of high-order CFD solvers cannot be achieved by hard-coding a single, “best” kernel configuration. Instead, kernel structure (unified versus split) and memory-access patterns (preloading, prefetching, direct global access) should be treated as tunable parameters. Systematic profiling is required per GPU family and precision mode to select robust configurations, and overly aggressive code transformations that significantly increase memory footprint should be avoided. Within this framework, direct global access without preloading emerges as a reliable default, with optional prefetching and kernel splitting applied selectively where profiling justifies their use.

5.5. Multi-GPU evaluation

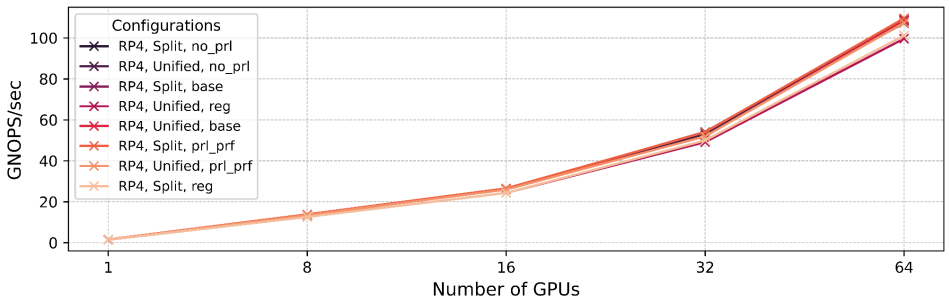

Figure 22: Taylor-Green Vortex throughput at scale (Single Precision).

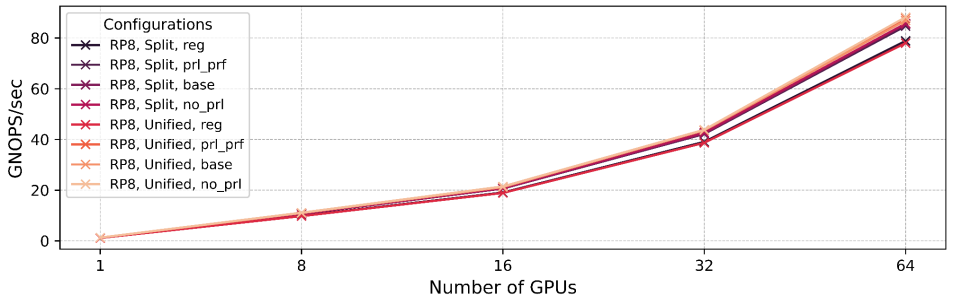

Figure 23: Taylor-Green Vortex throughput at scale (Double Precision).

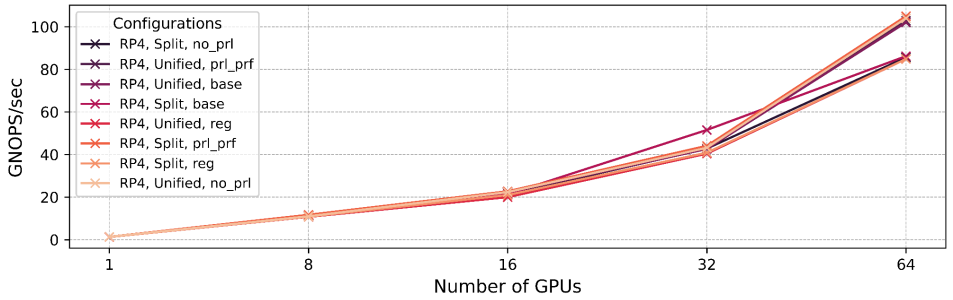

Figure 24: ChannelFlow throughput at scale (Single Precision)

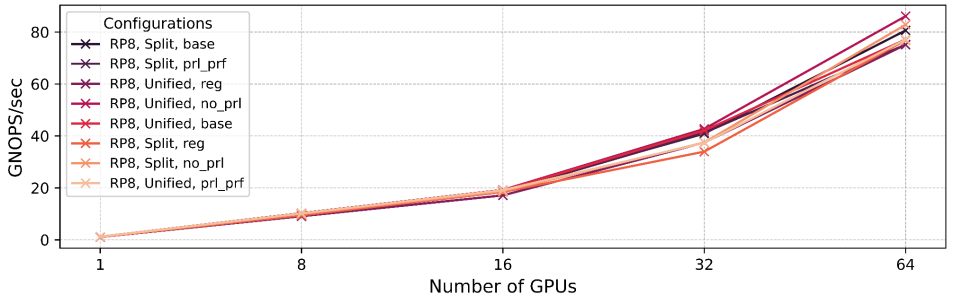

Figure 25: ChannelFlow throughput at scale (Double Precision)

The multi-GPU experiments (Figures 22-25) extend the single-GPU study by examining how the same kernel structures and memory-access variants behave when the solver is distributed across many accelerators under weak scaling. The global problem size is

increased proportionally to the number of GPUs so that each device advances approximately one million mesh nodes, keeping the per-GPU computational workload nearly constant while increasing the MPI communication surface. The channel-flow configuration is used as the primary benchmark, with both single-precision (FP32) and double-precision (FP64) arithmetic and all eight SOD2D variants (unified versus split convection kernel, combined with no_prl, base, prl_prf, and reg). To quantify scalability in a comparable manner, we employ a throughput metric, GNOPS (Giga Node Operations Per Second), defined as the total number of mesh nodes multiplied by the number of Runge–Kutta stages and time steps, divided by the solver runtime. This reflects the homogeneous update pattern of each node in the convection–diffusion pipeline and provides a device- and scale-independent measure of effective progress per unit time.

Across the weak-scaling campaign, SOD2D exhibits generally good scalability for all kernel variants: GNOPS increases with the number of GPUs without any dramatic loss of efficiency, and FP32 systematically achieves higher throughput than FP64, with an average advantage of approximately 22%. However, the choice of kernel structure and memory-access pattern has a non-negligible impact on performance at scale. For the channel-flow configuration in FP64, the unified convection kernel without preloading (no_prl) consistently delivers the highest GNOPS across all tested GPU counts, with improvements of up to about 15% relative to the worst-performing variant. In this regime, the additional data motion and local-memory footprint introduced by preloading are not compensated by increased locality, and the simpler direct-access design proves more efficient. In FP32, the picture is more nuanced: while the no_prl variant often remains competitive, combinations of preloading, prefetching and kernel splitting can become favourable at specific scales, and the ordering of the variants is not invariant with respect to the number of GPUs.

The sensitivity of performance to these implementation choices is particularly pronounced for the channel-flow case, where the spread between the slowest and fastest configuration reaches roughly 23.8% in FP32 and 14.6% in FP64. Moreover, the variant that is optimal at an intermediate scale (e.g., 32 GPUs) is not guaranteed to remain optimal at larger scales (e.g., 64 GPUs), so naive extrapolation of single-GPU or small-scale results can lead to substantial throughput loss at production scale. For instance, when naively extrapolating from 32 to 64 GPUs for ChannelFlow on FP32, one would face a 23.8% throughput penalty. Together with the single-GPU analysis, these findings indicate that there is no universally optimal combination of kernel structure and memory-access strategy. Instead, kernel splitting, preloading and prefetching should be treated as tunable parameters whose optimal configuration depends on precision, flow case and GPU count. From a performance-portability perspective, this argues for integrating lightweight profiling

or auto-tuning at the target scale, rather than relying on fixed, architecture-agnostic optimization choices.

5.6. Cross-Layer Exploration Performance Gains

Across platforms, the main cross-layer takeaway is that convection dominates (≈50% of runtime on NVIDIA, up to ~70% on MI250X), so optimizations there drive overall gains. On NVIDIA, the most robust improvement is removing explicit preloading: no_prl beats base across meshes/precisions (especially FP32), while reg is consistently worse due to its larger working set. If preloading is kept, manual prefetching can still help (e.g., ~14% speedup for V100, FP64, split). Kernel splitting is precision/vendor dependent: typically hurts FP32 on NVIDIA (launch/traffic overhead) but can help FP64 by reducing register pressure. Generation-to-generation, A100 improves over V100 by ~1.47× (FP32) and ~3.05× (FP64) with broadly similar variant ranking. On AMD MI250X, baseline performance can be ~15–16× slower than V100 for an 8M-node FP32 baseline, but the code is highly occupancy-sensitive: splitting yields ~1.5× convection speedup, and prl_prf can deliver ~40–60% over base (FP32, split). At scale, GNOPS increases with GPU count and FP32 averages ~22% higher than FP64, yet kernel choices still matter—best vs. worst differs by ~23.8% (FP32) and ~14.6% (FP64) and can reorder with GPU count—so there is no universally optimal variant, only robust defaults (no_prl) plus conditional choices (split/prefetch). Relevant information can be found in the corresponding publication [32].

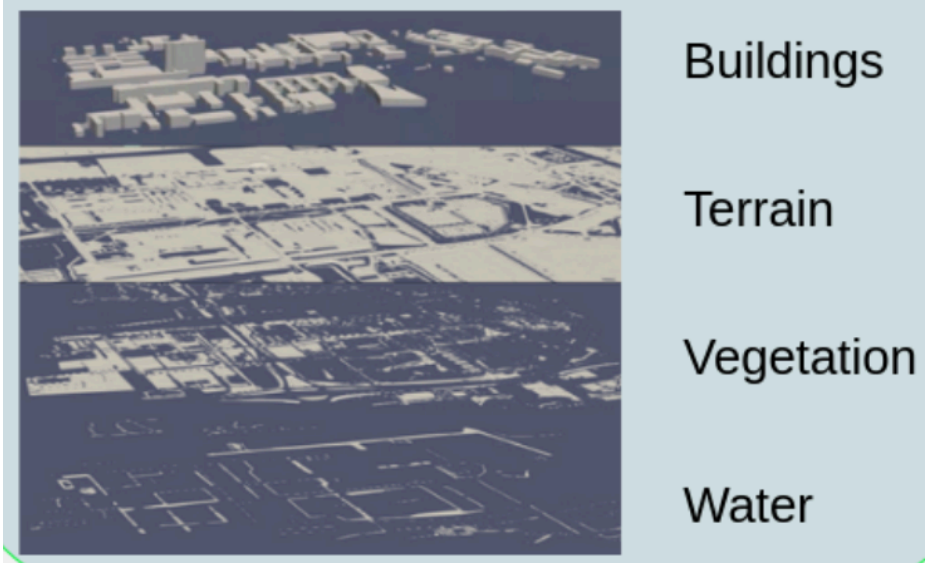

Figure 26: TUDelft input case

6. GPU Acceleration of OpenFOAM Solvers

6.1. Acceleration Strategy and Implementation

To accelerate the OpenFOAM solvers for resolving airflow around the TU Delft campus case visualized in Figure 26, we utilize a methodology that integrates GPU-enabled linear algebra libraries under the standard OpenFOAM numerical operators without altering the discretization, the physical models, or case configuration. An example integration of the tools is provided by [33] for OpenFOAM v2306 [34].The acceleration workflow builds upon the OpenFOAM solver stack and introduces petsc4Foam [35][36] as an intermediate layer. petsc4Foam provides a direct coupling between OpenFOAM's finite-volume operators and the PETSc framework, enabling access to scalable linear solvers, algebraic multigrid preconditioners, and implicit data structures optimized for GPU execution. As part of this integration, OpenFOAM's native LDU matrix format is transformed into compressed sparse row (CSR) form using foam2csr. This conversion enables the PETSc and AmgX backends to operate efficiently on device-resident matrices while preserving the existing OpenFOAM discretization and assembly logic. Within this setup, PETSc serves as the primary algebraic interface, exposing solvers and preconditioners that are executed either fully on the GPU or in mixed host/device configurations. For GPU-accelerated runs, AmgX [37] is introduced through the AmgXWrapper interface, which maps PETSc linear-algebra objects to CUDA-optimized AmgX solvers. AmgXWrapper also provides implicit data management, which ensures that data movement between CPU and GPU is minimized without rewriting OpenFOAM's matrix-assembly routines. From the OpenFOAM user perspective, solver and preconditioner selection is performed entirely through dictionary entries, and no major solver-side code modifications are required.

This methodology allows the TU Delft case to be executed using the standard OpenFOAM icoFoam or simpleFoam workflow while replacing the CPU-only pressure correction with a GPU-accelerated solver. For large meshes such as the one at hand, the computational bottleneck shifts from OpenFOAM's finite-volume assembly to the linear algebra kernel; thus, the runtime of the pressure equation is greatly reduced. In summary, acceleration of the TU Delft OpenFOAM case is achieved by:

- Keeping the core OpenFOAM solver unchanged (same discretization, same case files).
- Replacing the native sparse linear algebra backends with GPU-optimized PETSc/AmgX through petsc4Foam.

- Performing matrix format conversion (ldu → CSR) using foam2csr to expose device-efficient sparse structures.
- Executing the pressure equation on the GPU using PETSc or AMGX.
- Maintaining OpenFOAM configuration semantics, so solver choice and GPU use are controlled by the *"system/fvSolution"* configuration file rather than low-level code changes.

Utilizing this approach enables low-fidelity CFD simulations of the airflow around the TUDelft campus to benefit from GPU acceleration without core changes to the baseline OpenFOAM code, ensuring compatibility, reproducibility, and portability across heterogeneous HPC environments. The specification of the hardware infrastructure setup used for the measurements in the following section is outlined below:

Hardware Infrastructure Specification	
CPU	Intel Core i7-6700 @ 3.40GHz
CPU Cores	4 Cores (8 threads with hyper-threading)
CPU Speed	Base 3.40 GHz, up to 4.00 GHz
CPU Cache	L1: 256K, L2: 1M, L3: 8M
GPU	NVIDIA GeForce GTX 1060 3GB
RAM Total	80 GiB System Memory
RAM Configuration	2 x (32GiB, 8GiB) DIMMs at 2133 MHz
Motherboard	H170-HD3

Table 2: Specification for the platform utilized.

6.2. Performance Evaluation and Results

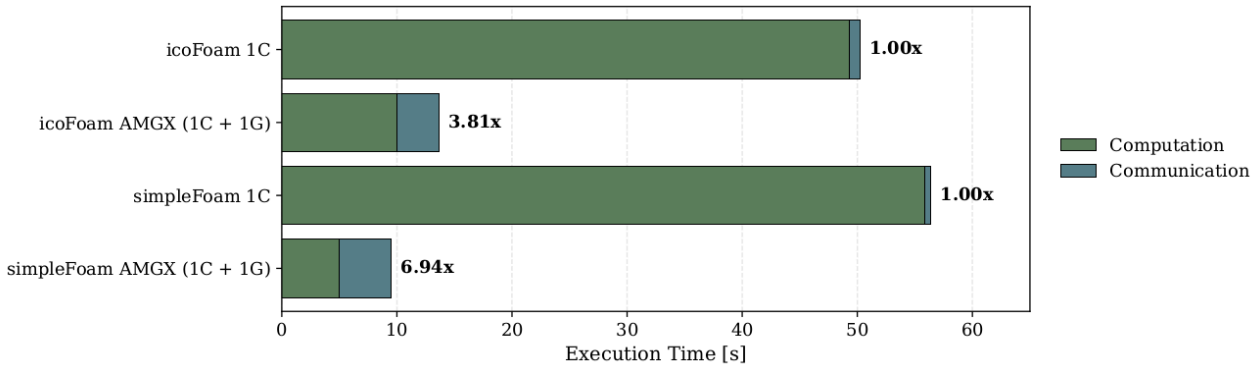

Figure 27: AMGX-accelerated OpenFOAM runtimes on the windAroundBuildings tutorial case.

Figure 27 illustrates the absolute runtimes broken down into computation and communication (includes all system calls captured with the Linux time command). For both the icoFoam and simpleFoam solvers. The CPU-only single-core (1C) runs define the baseline performance (1.00×). Replacing the native OpenFOAM GAMG linear solver with an AMGX backend using one MPI process and one GPU (1C+1G) leads to notable acceleration for both solvers. The simpleFoam solver obtains significantly higher acceleration, indicating that it is more strongly dominated by the pressure-correction linear solve (repeated within the SIMPLE loop), whereas icoFoam retains a larger fraction of runtime in transient/auxiliary operations and fixed overheads that are not accelerated by the AMGX backend.

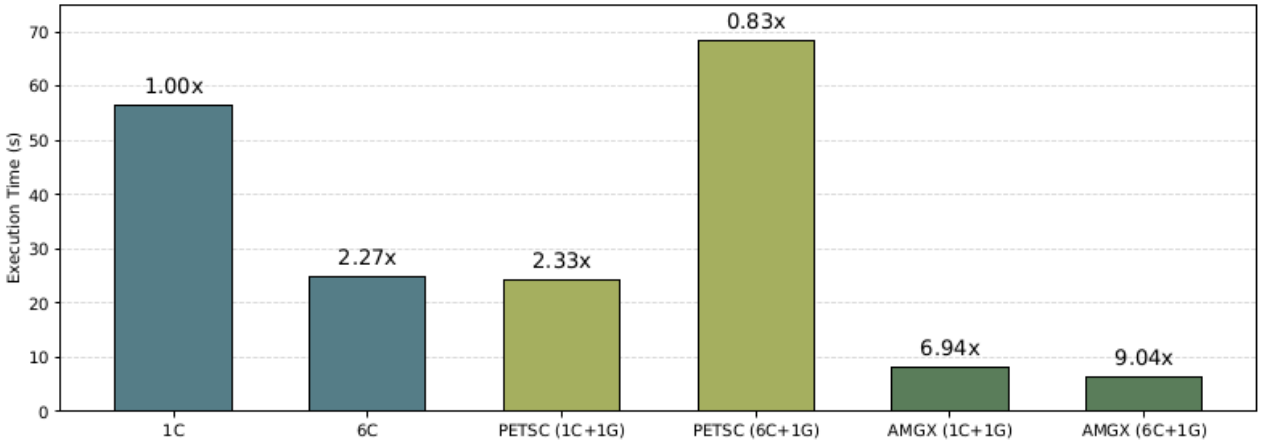

Figure 28: PETSc and AMGX -accelerated execution times of the TUDelft case when scaling MPI processes (simpleFoam solver).

In Figure 28, the speedups for both accelerated solvers are shown for 1 and 6 MPI processes, using the simpleFoam solver on the TUDelft case shown in Figure 24. The 1C CPU configuration again serves as the reference (1.00×). The solver exhibits sufficient parallelism, obtaining 2.27x speedup at scale (6C). Using PETSc with one GPU (1C+1G) provides a similar improvement (2.33×), while case partitioning when using PETSc leads to a significant slowdown. In contrast, AMGX consistently achieves high performance (6.94× speedup with 1C+1G and 9.04× with 6C+1G).

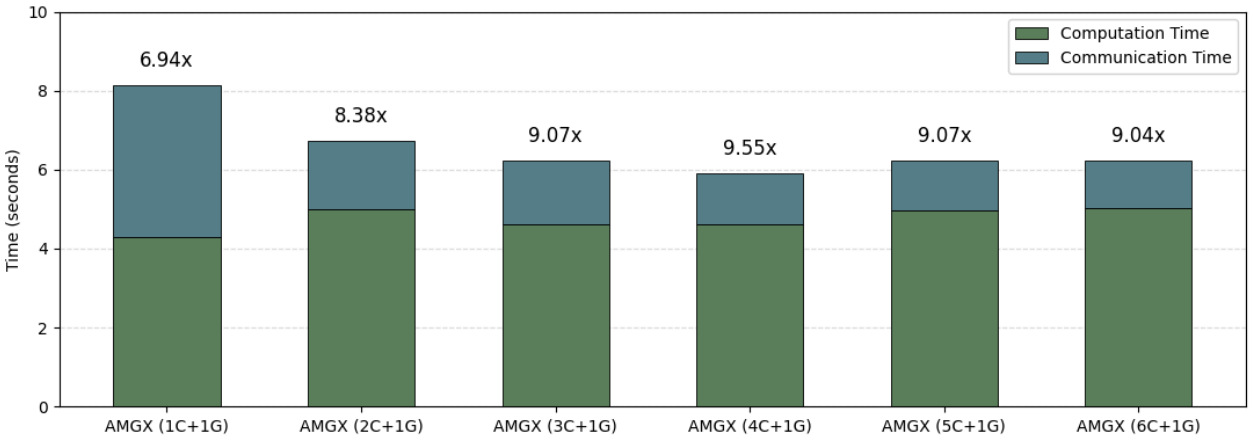

Figure 29: AMGX-accelerated execution times of the TUDelft case when scaling MPI processes. (simpleFoam solver).

Increasing MPI processes with 1 GPU yields indeterminate runtimes regarding total computation, since kernel launch and communication overheads are introduced, but the problem is partitioned across MPI ranks, and memory locality is exploited more efficiently. Host-to-device and device-to-host memory transfers (accounted for in communication) are more efficiently overlapped with computation at a greater scale, since only the working set of one MPI rank needs to be loaded on the GPU before transfers begin. However, communication overheads are introduced at scale. To investigate the net effect of the aforementioned factors, we perform a sweep of the MPI rank presented in Figure 26. The total time is obtained using the Linux *time* command, and it is partitioned into computation/communication using the ratio between total processing and system call time aggregated per MPI rank. As seen in the Figure, speedup peaks at 9.54× for 4C+1G. Beyond this point, performance slightly decreases to 9.04× (6C+1G). This finding motivates further autotuning to obtain marginal gains by increasing the MPI rank further than the number of available GPUs.

7. Conclusion

The RefMap European Project investigates the transition of aviation towards Unmanned Air Mobility and its impacts on safety and environmental concerns. A major goal of RefMap is to deliver flow prediction for drone flights within urban environments using deep-learning models trained on datasets generated from both high-resolution (LES-like) and lower-fidelity (e.g., RANS) CFD simulations. This deliverable addresses the excessive computational demands of such simulation campaigns by combining GPU acceleration and autotuning techniques that reduce time-to-solution while preserving numerical correctness. In particular, we profile and accelerate the SOD2D high-fidelity solver and develop an auto-tuning workflow that explores OpenACC parallelisation parameters to achieve robust speedups across heterogeneous systems. Complementary cross-layer analysis shows that optimal configurations depend on GPU architecture, compiler stack, precision mode, and scaling regime, motivating scale-aware tuning rather than fixed “one-size-fits-all” settings. We further perform GPU-accelerated execution for representative OpenFOAM workloads through lightweight integration with GPU-enabled libraries, reducing the runtime of the dominant pressure-solver components and accelerating the low-fidelity branch of the pipeline. Overall, these interventions improve performance portability and simulation throughput, enabling faster multi-fidelity dataset generation and more rapid iteration of RefMap’s urban-flow prediction models on modern HPC platforms. An overview of the presented work and its integration into the broader scope of REFMAP can be found in the relevant publication [38].

Future work will focus on extending autotuning to multi-GPU communication and partitioning effects for both the high and the low fidelity branch, advancing hotspot-level (Whitebox) tuning to reduce tuning cost, broadening validation across additional GPU generations and toolchains, and integrating automated configuration selection into the RefMap workflow. This is so that each platform and scale can be executed with near-optimal settings.

Acknowledgements

We acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI, hosted by CSC (Finland) and the LUMI consortium through a EuroHPC Regular Access call. The computations were enabled by the Berzelius resource provided by the Knut and Alice Wallenberg Foundation at the National Supercomputer Centre.

References

[1] RefMap (n.d.), RefMap EU project, https://www.refmap.eu/, retrieved 2026-01-23.

[2] Gasparino Ferreira da Silva, L., Spiga, F., & Lehmkuhl, O. (n.d.), Sod2D: A GPU-enabled spectral finite elements method for compressible scale-resolving simulations, Manuscript / technical description.

[3] Weller, H. G., Tabor, G., Jasak, H., & Fureby, C. (1998), A tensorial approach to computational continuum mechanics using object-oriented techniques, Computers in Physics, 12(6), pp. [pages not provided].

[4] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., & Amarasinghe, S. (2014), OpenTuner: An extensible framework for program autotuning, Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 303–316.

[5] Straubinger, A., Rothfeld, R., Shamiyeh, M., Büchter, K.-D., Kaiser, J., & Plötner, K. O. (2020), An overview of current research and developments in urban air mobility – setting the scene for UAM introduction, Journal of Air Transport Management, 87, 101852.

[6] Yim, S. H., Lee, G. L., Lee, I. H., Allroggen, F., Ashok, A., Caiazzo, F., Eastham, S. D., Malina, R., & Barrett, S. R. (2015), Global, regional and local health impacts of civil aviation emissions, Environmental Research Letters, 10(3), 034001.

[7] Mason, P. J. (1994), Large-eddy simulation: A critical review of the technique, Quarterly Journal of the Royal Meteorological Society, 120(515), 1–26.

[8] Moin, P., & Mahesh, K. (1998), Direct numerical simulation: A tool in turbulence research, Annual Review of Fluid Mechanics, 30(1), 539–578.

[9] Fischer, P., Lottes, J., & Tufo, H. (2007), Nek5000 (Technical report No. NEK5000), Argonne National Laboratory, Argonne, IL.

[10] Yan, Z., Chen, R., & Cai, X.-C. (2022), Large eddy simulation of the wind flow in a realistic full-scale urban community with a scalable parallel algorithm, Computer Physics Communications, 270, 108170.

[11] Lenz, S., Schönherr, M., Geier, M., Krafczyk, M., Pasquali, A., Christen, A., & Giometto, M. (2019), Towards real-time simulation of turbulent air flow over a resolved urban canopy using the cumulant lattice Boltzmann method on a GPGPU, Journal of Wind Engineering and Industrial Aerodynamics, 189, 151–162.

[12] Mortezazadeh, M., Wang, L. L., Albettar, M., & Yang, S. (2022), CityFFD – City fast fluid dynamics for urban microclimate simulations on graphics processing units, Urban Climate, 41, 101063.

[13] Yang, M., Oh, G., Xu, T., Kim, J., Kang, J.-H., & Choi, J.-I. (2023), Multi-GPU-based real-time large-eddy simulations for urban microclimate, Building and Environment, 245, 110856.

[14] Kirk, D. B., & Hwu, W.-m. W. (2016), Programming massively parallel processors: A hands-on approach (3rd ed.), Morgan Kaufmann, San Francisco, CA.

[15] Offermans, N., Marin, O., Schanen, M., Gong, J., Fischer, P., Schlatter, P., Obabko, A., Peplinski, A., Hutchinson, M., & Merzari, E. (2016), On the strong scaling of the spectral element solver Nek5000 on petascale systems, Proceedings of the Exascale Applications and Software Conference 2016, pp. 1–10.

[16] Markidis, S., Gong, J., Schliephake, M., Laure, E., Hart, A., Henty, D., Heisey, K., & Fischer, P. (2015), OpenACC acceleration of the Nek5000 spectral element code, The International Journal of High Performance Computing Applications, 29(3), 311–319.

[17] Otten, M., Gong, J., Mametjanov, A., Vose, A., Levesque, J., Fischer, P., & Min, M. (2016), An MPI/OpenACC implementation of a high-order electromagnetics solver with GPUDirect communication, The International Journal of High Performance Computing Applications, 30(3), 320–334.

[18] Świrydowicz, K., Chalmers, N., Karakus, A., & Warburton, T. (2019), Acceleration of tensor-product operations for high-order finite element methods, The International Journal of High Performance Computing Applications, 33(4), 735–757.

[19] Karp, M., Jansson, N., Podobas, A., Schlatter, P., & Markidis, S. (2020), Optimization of tensor-product operations in Nekbone on GPUs, arXiv preprint, arXiv:2005.13425.

[20] Vincent, J., Gong, J., Karp, M., Peplinski, A., Jansson, N., Podobas, A., Jocksch, A., Yao, J., Hussain, F., Markidis, S., et al. (2022), Strong scaling of OpenACC-enabled Nek5000 on several GPU-based HPC systems, International Conference on High Performance Computing in Asia-Pacific Region, pp. 94–102.

[21] Reguly, I. Z., & Mudalige, G. R. (2020), Productivity, performance, and portability for computational fluid dynamics applications, Computers & Fluids, 199, 104425.

[22] Medina, D. S., St-Cyr, A., & Warburton, T. (2014), OCCA: A unified approach to multi-threading languages, arXiv preprint, arXiv:1403.0968.

[23] Fischer, P., Kerkemeier, S., Min, M., Lan, Y.-H., Phillips, M., Rathnayake, T., Merzari, E., Tomboulides, A., Karakus, A., Chalmers, N., et al. (2022), NekRS, a GPU-accelerated spectral element Navier–Stokes solver, Parallel Computing, 114, 102982.

[24] Thomée, V. (2007), Galerkin finite element methods for parabolic problems (2nd ed.), Springer-Verlag, Berlin, Heidelberg.

[25] Farber, R. (2016), Parallel programming with OpenACC (1st ed.), Morgan Kaufmann, San Francisco, CA.

[26] Reguly, I. Z., & Mudalige, G. R. (2020), Productivity, performance, and portability for computational fluid dynamics applications, Computers & Fluids, 199, 104425.

[27] Pennycook, S. J., Sewall, J. D., & Lee, V. W. (2016), A metric for performance portability, arXiv preprint, arXiv:1611.07409.

[28] Geuzaine, C., & Remacle, J.-F. (2009), Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities, International Journal for Numerical Methods in Engineering, 79(11), 1309–1331.

[29] NVIDIA (n.d.), NVIDIA Nsight Systems, https://developer.nvidia.com/nsight-systems, retrieved 2026-01-23.

[30] National Supercomputer Centre (NSC), Linköping University (n.d.), Berzelius cluster, https://www.nsc.liu.se/systems/berzelius/, retrieved 2026-01-23.

[31] Koliogeorgi, K., Anagnostopoulos, G., Zampino, G., Sanchis, M., Vinuesa, R., & Xydis, S. (2024), Auto-tuning multi-GPU high-fidelity numerical simulations for urban air mobility, Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–6, doi: 10.23919/DATE58400.2024.10546549.

[32] Eleftherakis, P.-E., Anagnostopoulos, G., Kapetanakis, A., Umair, M., Vet, J.-Y., Iliakis, K., Vincent, J., Gong, J., Vinuesa, R., & Xydis, S. (2025), Poster: Performance portability in GPU-accelerated spectral finite element fluid simulations: A cross-layer exploration approach, Proceedings of the 22nd ACM International Conference on Computing Frontiers (CF '25), 228–229.

[33] NEXTFOAM Co., Ltd. (n.d.), Computational Fluid Dynamics (CFD) services and solutions, https://nextfoam.co.kr/, retrieved 2026-01-15.

[34] OpenFOAM Foundation (2024), OpenFOAM User Guide, https://www.openfoam.com/documentation/user-guide, retrieved 2026-01-23.

[35] PETSc-for-Foam Contributors (n.d.), petsc4foam, https://gitlab.com/petsc/petsc4foam, retrieved 2026-01-15.

[36] OpenFOAM (n.d.), external-solver (GitLab project), https://develop.openfoam.com/modules/external-solver, retrieved 2026-01-15.

[37] Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N., Sellappan, V., & Strzodka, R. (2015), AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods, SIAM Journal on Scientific Computing, 37(5), S602–S626.

[38] Anagnostopoulos, G., Eleftherakis, P., Maroudis, A. C., Kapetanakis, A., Iliakis, K., Guastoni, L., Umair, M., Vet, J., Vincent, J., Gong, J., Vinuesa, R., & Xydis, S. (2025), RefMap: From HPC to high fidelity CFD simulations to optimized flow prediction models for next-gen urban airspace mobility, Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2025), Lecture Notes in Computer Science, pp. 1–16.

[39] Kapetanakis, A., Ferikoglou, A., Anagnostopoulos, G., & Xydis, S. (2025), Dataflow optimized reconfigurable acceleration for FEM-based CFD simulations, Design, Automation & Test in Europe Conference (DATE), pp. 1–6, doi: 10.23919/DATE64628.2025.10992836.

Appendix I

Dataflow Optimized Reconfigurable Acceleration for FEM-based CFD Simulations[3]

This appendix presents a dataflow-optimized FPGA acceleration approach for FEM-based CFD, specifically SOD2D, targeting the Runge–Kutta loop where diffusion and convection dominate runtime [39]. The design maps element-level computations to pipelined dataflow tasks and uses explicit memory-interface management to sustain throughput across large meshes.

The CFD solver considered in this work is characterized by a deeply nested computational structure dominated by the Runge–Kutta (RK) time integration scheme, where diffusion, convection, and state update operators are repeatedly evaluated over all elements and nodes of the discretized mesh. This structure exposes a high degree of regularity, data reuse, and coarse-grained parallelism at both the element and node levels, while also imposing strict demands on memory bandwidth and data movement. The accelerator design presented in this section is driven by these algorithmic characteristics, aiming to map the most computationally intensive RK stages onto a reconfigurable FPGA architecture through task-level pipelining, dataflow execution, and explicit memory interface management. The following paragraphs detail how these principles are reflected in the proposed architecture and the associated micro-architectural optimizations.

[3] The FPGA acceleration of the high fidelity simulator SOD2D was conducted by ICCS outside the scope of T2.4. Therefore, it is summarized in this Appendix.

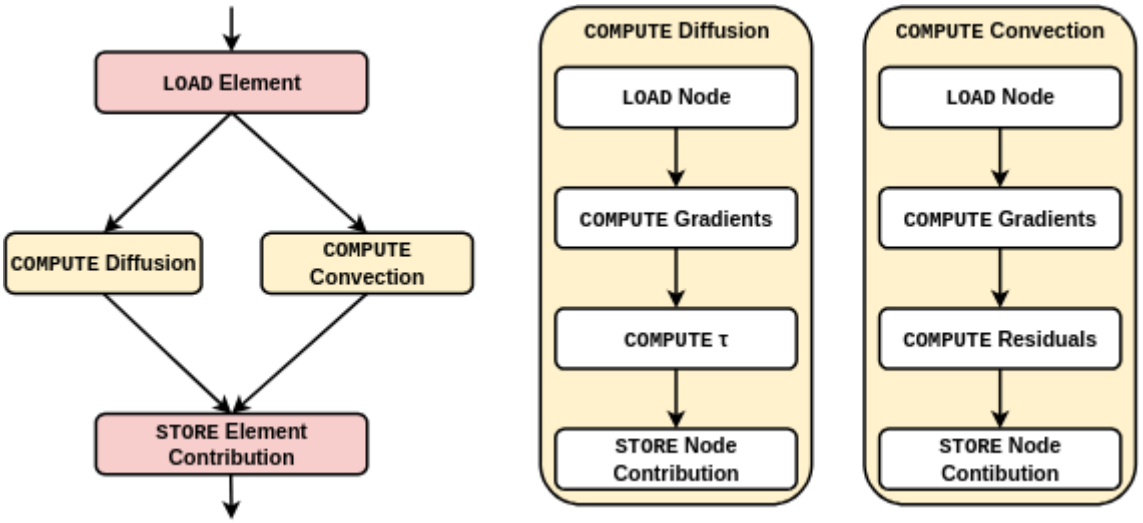

Figure 30: Dataflow Graph of Core Computation

Figure 30 presents the dataflow graph of the core FEM computation executed within each Runge–Kutta stage, capturing the sequence of operations required to evaluate the diffusion and convection terms for a single mesh element. The graph highlights the load–compute–store structure of the algorithm, where element-level data are first fetched from memory, followed by independent diffusion and convection computations that operate over the nodes of the element. At each node, the required geometric and state information is used to compute gradients, the viscous stress tensor *tau*, and the associated residuals, which are then accumulated into the element contribution. The absence of data dependencies between the diffusion and convection paths, as depicted in the graph, enables their concurrent evaluation and motivates their later fusion into a single computational module. This representation makes explicit the element-wise and node-wise parallelism inherent in the FEM formulation and directly informs the dataflow-oriented accelerator design presented in the following section.

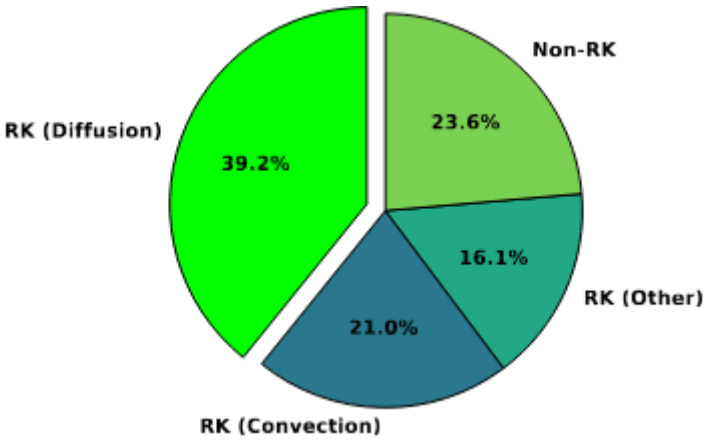

Figure 31: Breakdown of Average Execution Time

Figure 31 illustrates the average execution time breakdown of the CFD solver across multiple problem sizes, highlighting the relative computational weight of each major algorithmic component. The results show that the Runge–Kutta (RK) time integration dominates execution time, accounting for approximately 76.5% of the total runtime. Within the RK loop, the diffusion and convection kernels emerge as the primary performance bottlenecks, contributing 39.2% and 21.04% of the overall execution time, respectively. This distribution reflects the high arithmetic intensity and memory access demands of the FEM-based evaluation of these terms, which require repeated element-wise and node-wise computations at every RK stage. The remaining runtime is attributed to state updates and auxiliary computations, which exhibit comparatively regular access patterns and lower computational complexity. This profiling analysis clearly identifies the RK loop—and, in particular, the diffusion and convection operators—as the most promising targets for hardware acceleration, directly motivating the dataflow-oriented FPGA design presented in the following section.

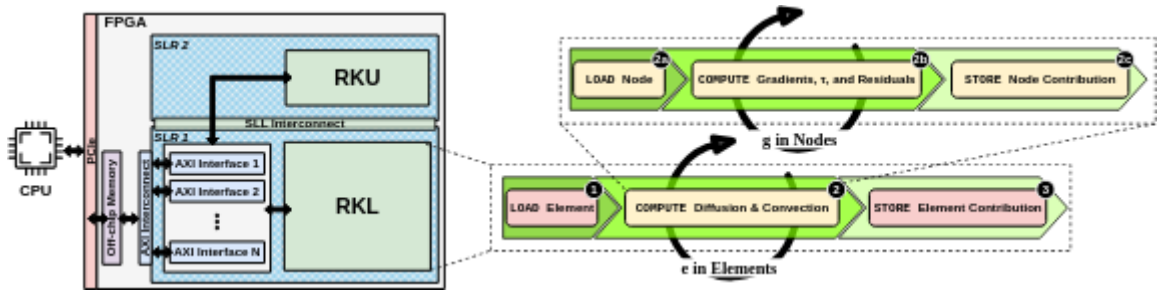

Figure 32: Overview of the Proposed Accelerator's Architecture

A high-level overview of the proposed FPGA-based acceleration architecture and its interaction with the host system is presented in Figure 32. The host CPU orchestrates the simulation flow by managing initialization, time-step control, and PCIe-based data transfers to the FPGA's off-chip memory. The accelerator itself is partitioned into two kernels mapped onto separate Super Logic Regions (SLRs) of the device: the Runge–Kutta Loop (RKL) kernel, which implements the computationally intensive FEM-based diffusion and convection operators, and the Runge–Kutta Update (RKU) kernel, which performs the state variable updates at the end of each RK stage. The figure highlights that RKL is directly connected to off-chip memory to sustain high-bandwidth, irregular access patterns, while RKU accesses the same memory through Super Long Lines (SLLs), reflecting its lower computational intensity and more regular access behavior. This spatial and functional separation enables concurrent execution, alleviates routing congestion, and facilitates efficient task-level pipelining within RKL, establishing

the architectural foundation for the dataflow and memory optimizations discussed in the subsequent subsections.

```
...
1 void readDDR() {
2   #pragma HLS INTERFACE MODE=m_axi BUNDLE=gmem_1 PORT=rho
3   #pragma HLS INTERFACE MODE=m_axi BUNDLE=gmem_2 PORT=Tem
4   #pragma HLS INTERFACE MODE=m_axi BUNDLE=gmem_3 PORT=mu_fluid
5   #pragma HLS INTERFACE MODE=m_axi BUNDLE=gmem_4 PORT=E
6   for (uint32_t inode=0; inode < NNODE; inode++) {
7     rho_elem_type rho_temp = rho[inode];
8     Tem_elem_type Tem_temp = Tem[inode];
9     mu_fluid_elem_type mu_fluid_temp = mu_fluid[inode];
10    E_elem_type E_temp = E[inode];
11  }
12 }
...
```

Figure 33: Individual AXI Interface Assignment Optimization

Figure 33 illustrates the individual AXI interface assignment strategy applied to the off-chip memory accesses of the *Load-Element* task. The figure shows how distinct arrays accessed within the same loop are explicitly mapped to separate AXI interfaces using HLS directives, enabling multiple memory transactions to be issued concurrently. By distributing these accesses across independent AXI ports, the design eliminates arbitration-induced serialization that would otherwise arise when multiple arrays share a single interface. This optimization is particularly critical for sustaining the throughput of task-level pipelining, as it ensures that memory transfers do not become the bottleneck of the pipeline initiation interval. The figure also implicitly highlights the need for careful interface management, since the number of available AXI ports is limited, motivating the controlled reuse of interfaces across tasks that are executed sequentially rather than concurrently.

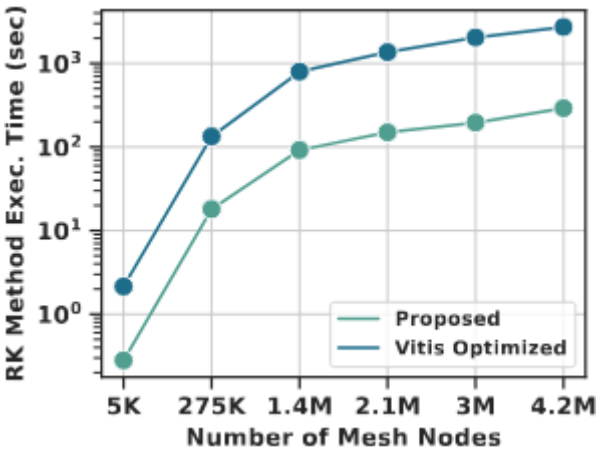

Figure 34: Execution Time for Different Numbers of Mesh Nodes

The execution time of the Runge–Kutta (RK) computation as a function of the number of mesh nodes is seen in Figure 34, comparing the proposed accelerator against the Vitis-HLS optimized design. The figure demonstrates a near-linear scaling behavior for both implementations as the mesh size increases from 1.4M to 4.2M nodes, indicating that the computational workload grows proportionally with problem size. Across all configurations, the proposed design consistently achieves significantly lower execution times than the Vitis-optimized baseline, with a uniform performance gap that widens in absolute terms for larger meshes. This behavior highlights the effectiveness of the proposed architectural partitioning and task-level optimizations, which sustain high throughput even as the problem size scales, while the Vitis-HLS optimized design remains constrained by lower operating frequency and routing congestion.

Design	FF%	LUT%	BRAM%	URAM%	DSP%
Vitis Opt.@100MHz	17.19	27.68	22.96	0.73	9.17
Proposed@150MHz	25.29	41.15	43.98	11.77	18.23

Table 3: Post P&R Resource Utilization Percentages

Table 3 reports the post place-and-route (P&R) resource utilization of the proposed accelerator and the Vitis-HLS optimized design on the AMD Alveo U200 FPGA. The results show that the proposed design incurs moderate increases in flip-flop (FF) and lookup table (LUT) utilization, approximately 1.5x higher than the Vitis-optimized baseline, reflecting the additional control and buffering logic introduced by task-level pipelining and kernel partitioning. Block RAM (BRAM) and DSP usage increase by up to 1.9x, consistent with the replication of compute units and the explicit management of on-chip data movement. A more pronounced increase is observed in URAM utilization, where the proposed design leverages URAM aggressively to stage large data structures and sustain high memory throughput. Despite this increase, overall resource usage remains well within device limits, demonstrating that the achieved performance gains are realized with controlled and targeted resource overhead rather than excessive hardware duplication.

In the work presented, we assessed a high-performance FPGA accelerator developed for solving the Navier-Stokes equations, with a focus on FEM and directly comparable to SOD2D, the high-fidelity simulator used in the rest of the deliverable. The proposed accelerator, implemented with HLS on AMD Alveo U200 FPGA, delivers 7.9x better performance than the Vitis-HLS optimized version, while compared with its software implementation counterpart running on a high-end server it reduces latency by 45% while

consuming 3.64x less power. This underscores the potential of our approach for solving the Navier-Stokes equations more efficiently, paving the way for addressing even more challenging CFD simulations.