A Systematic Performance Study of Parallel Programming Models for Stencil-Based CFD Applications on Heterogeneous Architectures

Mariana Padilha[1], Claudio Schepke[1(✉)], Tailí Petry[1], and José Rufino[2]

[1] Laboratory of Advanced Studies in Computation (LEA), Federal University of Pampa (UNIPAMPA), Alegrete, Brazil
{marianarp.aluno,claudioschepke,tailipetry.aluno}@unipampa.edu.br
[2] Research Centre in Digitalization and Intelligent Robotics (CeDRI), Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC), Instituto Politécnico de Bragança, 5300-253 Bragança, Portugal
rufino@ipb.pt
https://sites.unipampa.edu.br/lea/

Abstract. Coupled free-flow and porous media phenomena are widely stencil-based computational fluid dynamics (CFD) applications. Memory-access patterns and parallel programming abstractions strongly influence their performance. Understanding the impact of programming models on stencil computations in heterogeneous architectures, which combine multi-core CPUs and GPUs, is critical. This paper presents a systematic performance study of directive-based (OpenMP and OpenACC) and explicit-kernel (CUDA) parallel programming models applied to a structured-grid porous-media CFD solver. We perform a full refactoring of a legacy Fortran implementation and evaluate task granularity, scalability behavior, and architectural influence across distinct CPUGPU systems. Moreover, we analyze numerical reproducibility under IEEE 754 double precision and characterize the computational and memory-access behavior of the solver. Experimental results demonstrate how the abstraction level and parallel decomposition strategy impact scalability and performance limits in stencil-based CFD applications on heterogeneous architectures.

Keywords: High-Performance Computing · Computational Fluid Dynamics · Porous Media Problem · OpenMP · OpenACC · CUDA

1 Introduction

Devastating climate phenomena and environmental catastrophes have repeatedly occurred in different parts of the world [9], often involving complex interactions between free fluid flow and porous media, such as those observed in heavy rainfall events, dam failures, and mining waste overflows. The simulation of this type of

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026, LNCS 16767, pp. 35–50, 2027.
https://doi.org/10.1007/978-3-032-30491-9_3

phenomenon involves the computation of operations based on the mathematical representation of the physical forces involved [2]. These phenomena are commonly modeled using the NavierStokes and DarcyForchheimer equations. Their numerical solution requires repeated stencil evaluations across space and time, making performance sensitive to memory-access patterns and parallel decomposition strategies.

Stencil-based computational fluid dynamics (CFD) solvers are widely used to simulate coupled free-flow and porous-media phenomena in environmental and engineering applications [7]. These solvers typically operate on structured grids and rely on iterative local-neighbor computations. As a result, they generate memory-intensive workloads and require strict convergence control. Due to these characteristics, stencil-based CFD applications are suitable candidates for evaluating parallel programming models on heterogeneous systems that combine multi-core CPUs and GPUs.

A previous study introduced a porous-media flow solver for an inclined plane configuration [10]. That study focused on algorithmic restructuring to improve cache efficiency and limit vectorization. Although modest gains were achieved in a single-core context, the increasing availability of heterogeneous systems demands a deeper investigation into how such legacy scientific codes can be redesigned and evaluated under distinct parallel programming abstractions.

Porting and parallelizing legacy CFD solvers involves more than inserting directives. It requires restructuring memory allocation strategies, adapting data access patterns, controlling task granularity, and ensuring numerical reproducibility across architectures. Directive-based models, such as OpenMP and OpenACC, and explicit-kernel programming approaches, such as CUDA, follow distinct abstraction philosophies, which may lead to different scalability behaviors and performance trends in stencil-dominated workloads.

In this work, we extend a porous-media CFD solver through a complete refactoring and parallel redesign targeting heterogeneous CPUGPU systems. The base legacy code is reorganized to support modular memory management, dynamic domain parameterization, and GPU-ready execution. Rather than focusing solely on raw speedup, we conduct a systematic performance study on how different parallel programming models could influence scalability, task granularity, and architectural behavior in a stencil-based CFD application.

The main contributions of this work are: (1) a structured refactoring of a legacy porous-media CFD solver into a modular and parameterized implementation suitable for heterogeneous architectures; (2) a systematic comparative evaluation of directive-based (OpenMP and OpenACC) and explicit-kernel (CUDA) programming models applied to a structured-grid stencil workload; (3) a detailed analysis of task granularity and scalability behavior across multi-core CPU and GPU; (4) a performance characterization of the computational and memory-access behavior of the solver, providing insight into observed scalability limits; (5) a numerical reproducibility validation under IEEE 754 double-precision arithmetic across CPU and GPU executions; (6) a controlled multi-architecture

experimental study highlighting the influence of hardware characteristics on parallel efficiency.

The remainder of this paper is organized as follows. Section 2 references some related works. Section 3 shows the application's functionalities and the code structure. The description of the methodology of the experiments is in Sect. 4. Section 5 shows the performance results for the different parallel implementation approaches. Finally, Sect. 6 presents the conclusion and future work.

2 Related Works

ALHafez and Kurdi [1] present a systematic analysis of MPI, OpenMP, and CUDA parallel programming interfaces across multiple dimensions: architectural foundations, performance characteristics, domain-specific suitability, programming complexity, and recent advancements. Their study restricts to a theoretical examination of each model's strengths, weaknesses, and optimization techniques.

Khalilov and Timoveev [6] compare the performance of CUDA, OpenMP, and OpenACC on an Nvidia Tesla V100 GPU in various typical scenarios that arise in scientific programming, such as matrix multiplication, regular patterns for memory access (BabelStream benchmark), and evaluate the performance of physical simulation codes implemented using these programming models (Cloverleaf and Lulesh benchmark). Moreover, they study the performance of matrix multiplication implemented in vendor-optimized BLAS libraries for the Nvidia Tesla V100 GPU and the Math Kernel Library (MKL) for Intel Xeon processors.

In [11], Silva et al. provide OpenMP and OpenACC implementations for a grain-drying simulation, a porous media application. They evaluate native `DO CONCURRENT`, OpenMP (loop, sections, tasks, target, and teams approach), and OpenACC parallel loops, maintaining the same code structure.

Karimi and Bhattacharya [5] focuses on an application that addresses a fluid flow and solute transport problem in periodic porous media. The authors select C++ for their implementation, adopting the CUDA interface and using the cuFFT and cuBLAS standard libraries for the GPU version of their code. Data exchange between the CPU and GPU occurs only at initialization and at the end of the execution to store the results for recovery. The rest of the calculation is conducted entirely on the GPU. Two FEM-based methods are solved on a single core of an Intel Skylake CPU (2.1 GHz), while the FFT-based method is solved on both the same CPU and a NVIDIA P100 GPU with 3584 cores (1.3 GHz). The performance of the GPU version is approximately 20× faster.

The approach pursued in our work, and described in this paper, considers an entire application, as proposed in [5,11], instead of conceptual notions [1] or benchmarks [6]. However, we apply the same parallel programming interfaces as in [6]: OpenMP, OpenACC, and CUDA. We also incorporate the notion of restricted data movement between the CPU and GPU only before and after the iterative step, as in [5], and diverge by explicitly implementing all functions called by our application, instead of making calls to external libraries.

3 Numerical Model Application

The original goal of the application we have refactored is to simulate and produce a numerical solution for a coupled flow of a free fluid over a porous medium. The model data are defined using discrete quantities through the Finite Volume Method applied to a heterogeneous system [4]. The application addresses the challenges associated with modeling abrupt transitions between free flow, governed by the Navier–Stokes equations [8], and porous media, described by the Darcy–Forchheimer equation [3]. In particular, the influence of porous matrix properties on flow behavior is investigated, contributing to a better understanding and numerical modeling of such complex systems.

The application source code is organized into three main stages: i) preprocessing, ii) main computation, and iii) data output, as illustrated in Fig. 1. Initially, two data structures are read from input files. These structures contain the physical parameters associated with a given simulation and the values defining the domain decomposition sizes. Subsequently, memory allocation is performed, and the finite volume discretization scheme is applied.

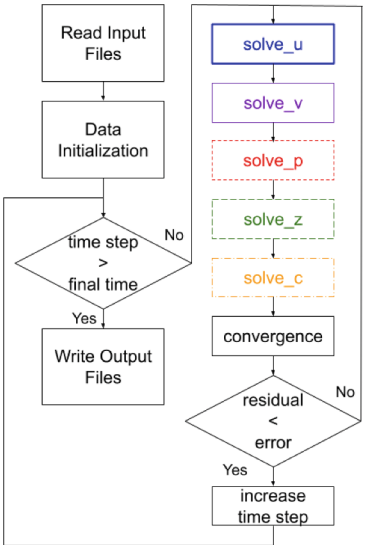

Fig. 1. Main flowchart of the numerical model. Adapted from [10].

The main computation advances over discrete time steps and iterates over a two-dimensional spatial domain. At each time step, the physical variables are updated iteratively until a convergence criterion is satisfied. Within this inner iteration loop, the first stage solves the momentum equations in the x and y directions. Next, the continuity equation is enforced by solving a Poisson equation associated with mass conservation. The third stage computes energy transport (temperature), followed by the calculation of concentration through the

solution of a scalar advection–diffusion equation. At the end of each inner iteration, selected variables are updated and the convergence metric is evaluated. The iterative process terminates either when the convergence criterion is reached or when the maximum number of iterations is exceeded. After each time-step increment in the outer loop, additional variables are updated accordingly.

At the end of the execution, all computed physical properties are written to separate output files, one for each variable. These outputs can subsequently be post-processed and visualized using external tools, such as Python scripts.

Figure 2 illustrates the subroutines invoked during the iterative solution process. For all solver routines—`solve_u`, `solve_v`, `solve_p`, `solve_z`, and `solve_c`—there are associated subroutine calls responsible for data computation and boundary condition treatment. For instance, in the pressure solver `solve_p` (highlighted in red), the `resp` subroutine computes intermediate data, while `bcp` applies the corresponding boundary conditions. The same structure is adopted for the remaining solver routines.

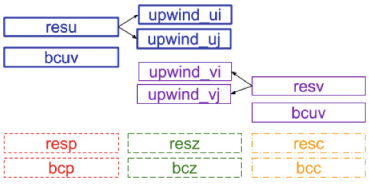

Fig. 2. Subroutines invoked by each solver during the iterative step. Box border types and font colors correspond to the operations shown in Fig. 1.

The routines responsible for data computation and boundary condition evaluation are invoked three times within each solver routine. This behavior results from the adoption of a second-order Runge–Kutta time integration scheme. In this approach, two intermediate estimates are required to compute the final approximation, which is obtained as a weighted average of the evaluated slopes. Additionally, the routines `upwind_ui`, `upwind_uj`, `upwind_vi`, and `upwind_vj` are employed to handle the simplified and distinct computation of grid points located near the domain boundaries, specifically in the second and penultimate rows and columns of the discrete mesh.

The convergence metric is computed from the residual vectors produced by `solve_u`, `solve_v`, and `solve_p`, using the `maxval` operator (denoted as max, below), which returns the maximum absolute value within each residual array:

$$residual_u = \max(|res_u|), \tag{1}$$

$$residual_v = \max(|res_v|), \tag{2}$$

$$residual_p = \max(|res_p|), \tag{3}$$

$$error = \max(residual_u, residual_v, residual_p). \tag{4}$$

The various implementations developed, based on the OpenMP, OpenACC, CUDA, and `DO CONCURRENT` parallel programming interfaces, are public[1].

The code organization is mainly governed by temporal discretizations. Spatial discretizations are controlled externally by the convergence loop, while internally, the operations are divided into five functions based on the physical calculations involved. Each function is organized into three loops. Each loop operates on one of the steps of the second-order Runge-Kutta method over all points.

The computation is distinct for the different types of discrete points. The vast majority of the elements follow a stencil processing pattern, including neighboring point values. For instance,

$$P_{ij} = V_{i-1,j} + V_{i+1,j} + V_{i,j-1} + V_{i,j+1}, \tag{5}$$

where P is a physical property and V abstracts distinct computations involving the processing of neighboring element values. The elements at the edges have simpler calculations according to the definition of boundary conditions:

$$\begin{array}{llll} u(x,y) = 0, & v(x,y) = 0 & \text{within the porous medium} \\ u(x,y) = 0, & v(x,y) = v_i & \text{in the free flow region,} \end{array} \tag{6}$$

where v_i is the injection speed.

To avoid a conditional test per computed point, points are grouped according to the type of processing applied to each group. That is, the internal points are computed in a block of rows and columns. The external points from the first and last rows and columns are computed separately, each in a specific loop.

4 Evaluation Methodology

In this section, we describe the computer architecture, software, domain discretization, and case studies targeted in this work.

4.1 Testbed

The tests were performed on a bare-metal Linux runtime, using three systems (nodes) of different architectures, with distinct numbers of cores, CPU frequencies, cache sizes, and different co-processors (GPUs) attached:

- $Node1$: 2 × Intel Xeon CPU E5-2650 octa-core processors, 2 threads per core, and 20 MB of L3, 256 KB (per core) L2, and 32 KB (per core) L1 caches; 1 × NVIDIA Quadro M5000 GPU with 2048 CUDA cores, 8 GB of global memory, and caches of 2 MB for L2 and 48 KB (per SM) for L1;
- $Node2$: 2 × Intel Xeon CPU E5-2420 hexa-core processors, 2 threads per core, and 15 MB of L3, 256 KB (per core) L2, and 64 KB (per core) L1 cache; 2 × NVIDIA TITAN V GPUs with 5120 CUDA cores, 12 GB of global memory, and caches of 4.5 MB for L2 and 96 KB (per SM) for L1;

[1] https://anonymous.4open.science/r/simulacao-de-meios-porosos-636E.

– *Node*3: 1 × Intel i9-13900HX CPU with 8 performance-cores (clock frequency limited to 4 GHz, 2 threads per core), 16 efficient-cores, and 36 MB of L3, 2 MB (per core) L2, 80 KB (per core) L1, 4 MB (per module) E-Core L2, and 96 KB (per core) E-Core L1 cache; 1 × NVIDIA GeForce RTX 4060 GPU with 3072 CUDA cores, 8 GB of global memory, and caches of 24 MB for L2 and 128 KB (per SM) for L1.

The compilers used were `nvfortran` and `nvcc` from NVIDIA HPC_SDK version 25.5, with the compilation flags `-O3` to allow aggressive compiler optimizations for all environments and input sizes, `-mp` when running in parallel using OpenMP threads, and `-stdpar=gpu -acc -Minfo=accel` for OpenACC.

4.2 Domain and Case Studies

Figure 3 illustrates the problem domain. The simulation considers a surface of 12 × 20 cm. The open fluid flow extends 15 cm in the positive y-axis direction, and the porous media extends 5 cm in the negative y-axis direction. Special operations occur in a transition zone between fluid flow and porous media, such as soil, for example. For the boundary conditions, there are also special operations.

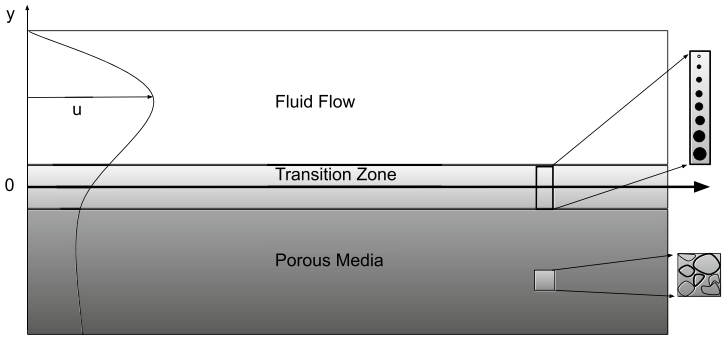

Fig. 3. Domain size scheme for a y-axis direction open fluid flow.

We defined 4 case studies. Each one is four times larger than the previous one. The number of points for each case after the discretization by the finite volume of the domain illustrated in Fig. 3 is: $Case1$ - $51 \times 84 = 4,284$; $Case2$ - $102 \times 169 = 17,238$; $Case3$ - $153 \times 254 = 38,862$; $Case4$ - $204 \times 339 = 69,156$.

The simulation covered the time period from 0 to 1 s using a Δt of 0.005 s, which is equivalent to 200 steps of the external loop. The convergence criteria requires less than 10^{-4} for the residual vectors of `solve_U`, `solve_V`, and `solve_P`. If the stopping criteria are not reached in 800,000 iterations for a specific time step, the algorithm ends, indicating convergence problems.

Each experiment was executed 10 times to collect the mean execution time for initialization, iterative steps, and post-processing. The standard deviation

observed was less than 1.5% of the average execution time. The pre- and post-processing times are low in relation to the total execution time. Table 1 presents the initialization and post iterative step times for $Node1$ for all four case studies.

Table 1. Pre- and Post-Processing execution time in seconds for $Node1$

Time	Case1	Case2	Case3	Case4
Pre-processing	0.03	0.07	0.16	0.32
Post-processing	0.06	0.17	0.48	1.62

The GPU memory usage observed, given by the `nvidia-smi` command for the four case studies, in order of domain size, was 312, 322, 340, and 368 MiB.

We also ensured strict numerical equivalence: all codes produce identical output results. This comparison and verification was done using Python scripts.

5 Evaluation Results

This section presents the experimental results obtained from executing the refactored CFD solver on three distinct heterogeneous architectures. We analyze the convergence behavior of the numerical method, evaluate the performance of multicore parallel implementations (OpenMP, OpenACC multicore, and `DO CONCURRENT`), and compare them against GPU-accelerated versions using OpenACC and CUDA. The analysis focuses on scalability, the impact of domain size on performance gains, and the relative efficiency of each programming model across different hardware configurations.

5.1 Numerical Convergence

Figure 4 presents the number of iterations necessary to attain the values for convergence at each time step (1 to 200) in $Case1$. Each time step converges when all partial residual values for `solve_u`, `solve_v`, and `solve_p` methods are smaller than 10^{-4} (stop criterion). The number of iterations decreases as the time step advances. That occurs because a time step computation takes the results from the previous step. After the 80th time step, the number of iterations reduces to 1 in most steps. A similar behavior appears in the other case studies.

Figure 5 displays the values achieved for the residual of the quantity of movement in x (res_u) and y (res_v), as well as of the continuity (res_p), also for $Case1$. The residual res_p is the last to achieve a value smaller than 10^{-4} in most cases, while res_u and res_v have already reached the 10^{-8} to 10^{-9} range.

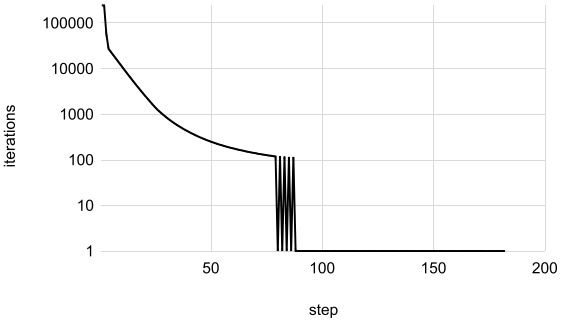

Fig. 4. Iterations needed for the convergence (log. scale, $Case1$).

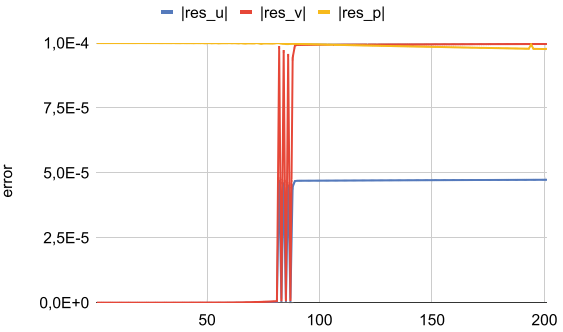

Fig. 5. Values of the residual error at the moment of convergence ($Case1$).

5.2 Multi-core Performance

Figure 6 shows the speedup for the $Case1$ in $Node2$, evaluating multi-core parallel programming interfaces against a sequential code version. We range the number of threads from 2 to 12, in steps of 2. Note that for each parallel programming interface, it is explicitly necessary to set the exact number of threads.

The results are relatively similar for OpenMP and OpenACC. The OpenACC has the worst results, perhaps limited by the multithreading generated by the compiler. The $Case1$ is the smallest test, and because of this, the performance gain is limited. For example, we tested `DO CONCURRENT`, and the speedup was 3.25, which is the maximum speedup of OpenACC.

Figure 7 presents the speedup for the three case studies. As before, we range the number of OpenMP threads from 2 to 12, in steps of 2, now comparing that version with the `DO CONCURRENT` Fortran version for multi-core processing and the OpenACC version for GPU computation. As the case study sizes increase, the speedup improves due to the larger granularity.

The relationship between the performances of the `DO CONCURRENT` and OpenMP variants improves with larger domains. For example, the `DO CONCURRENT` variant in $Case1$ has a speedup equivalent to that of 4 OpenMP

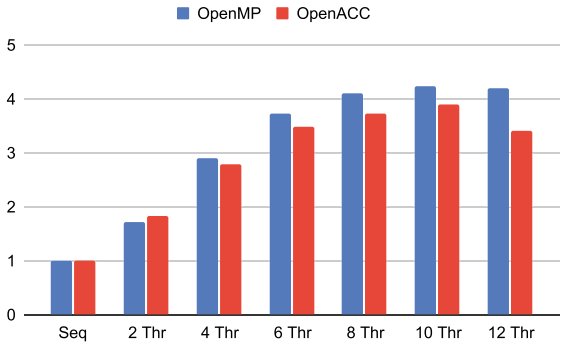

Fig. 6. Speedup of Multicore OpenMP and OpenACC variants ($Case1$ in $Node2$).

threads, while in $Case2$ there is a speedup between 6 and 8 OpenMP threads, and the speedup exceeds 8 OpenMP threads for $Case3$.

The OpenACC variant running on the GPU on $Node1$ achieved a speedup of 9.27 to the sequential single-core implementation, for $Case3$. This result is similar to using OpenMP with 12 threads.

To gain additional insight into the GPU-based OpenACC implementation, we also ran with $Case4$. In this scenario, the sequential version takes $314,241.68s$, while the OpenACC variant demands just $12,807.16s$, yielding a speedup of $24.53\times$, improving over the value ($20\times$) presented in [5].

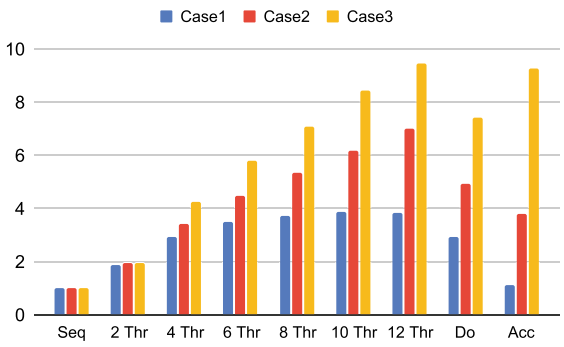

Fig. 7. Speedup of the Multicore OpenMP and `DO CONCURRENT` variants, and of the GPU-based OpenACC variant ($Case1$, $Case2$, and $Case3$ in $Node1$).

5.3 GPU Performance

We also provide a detailed analysis of the GPU-accelerated implementations using OpenACC and CUDA. We evaluate their performance on the $Node3$

architecture, compare the efficiency of directive-based versus explicit kernel programming, and employ the Roofline model to identify performance bottlenecks. The analysis focuses on the results obtained with the RTX 4060 Laptop GPU, which represents a modern and energy-efficient architecture where the trade-offs between programming effort and performance are particularly relevant.

CUDA Implementation and Profiling. The main CUDA implementation choices are summed up in Table 2, and profiling results are provided in Table 3.

We use 2D thread blocks of 16×16 threads to map the structured grid onto the GPU execution model. This configuration provides a good balance between thread occupancy and memory access efficiency for stencil-based computations.

Table 2. CUDA kernel configuration used in the GPU implementation.

Parameter	Value
Thread block size	16×16
Thread mapping	2D structured grid (1 thread per cell)
Kernel organization	27 kernels (one per solver stage)
CUDA streams	Used for boundary condition kernels
Memory access pattern	Strided global memory accesses in 2D arrays

A dedicated CUDA kernel receives each major computation state of the solver. This design enables fine-grained control over GPU execution and different solver stages to be scheduled independently on the device.

We use CUDA streams to enable the concurrent execution of independent kernels whenever data dependencies permit it. However, several solver stages require sequential execution due to dependencies between intermediate fields. As a result, full kernel concurrency is not always possible, and streams are applied only to operations to achieve safe overlap.

The mapping between grid points and threads preserves spatial locality. However, due to the layout of the 2D arrays in memory, the fastest varying thread dimension does not perfectly align with the contiguous memory axis, resulting in partially strided global memory accesses.

The profiling results (see Table 3) revealed that the majority of the runtime is concentrated in the velocity solver kernels. Together, the `solve_u` and `solve_v` kernels account for approximately 70% of the total kernel execution time, reflecting the higher computational complexity of the momentum equations. The scalar transport kernels (`solve_z` and `solve_c`) represent about 26% of the runtime, while the pressure solver (`solve_P`) contributes only 4%. Additionally, only 1.9% of the total execution time corresponds to host-side overhead, indicating that the solver computation is effectively offloaded to the GPU (Fig. 8).

Table 3. Runtime distribution among the main CUDA kernel.

Kernel	Total Time (s)	Percentage
solve_u	2200.7	34.77%
solve_v	2249.3	35.53%
solve_p	255.0	4.03%
solve_z	793.0	12.53%
solve_c	832.2	13.15%
Total kernel time	6330.3	100%
Total execution time	6453.0	—
Overhead	122.8	1.90%

CPU-Based vs. GPU-Based Variants on Node3. Figure 9 presents the speedup achieved by the different parallel implementations (CPU- and GPU-based) on $Node3$ for $Case1$, $Case2$, and $Case3$, all relative to the original sequential CPU version. Several important observations emerge from these results.

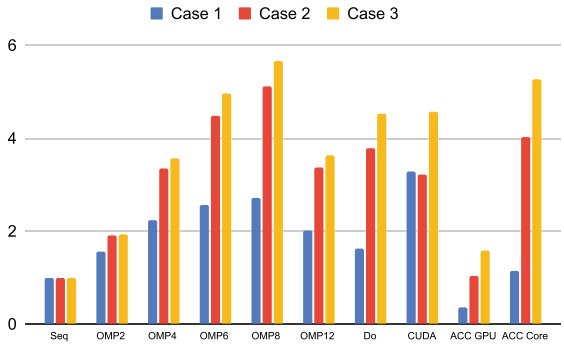

Fig. 8. Speedup of the OpenMP, `DO CONCURRENT` and OpenACC Multicore variants, and of the CUDA and OpenACC GPU variants ($Case1$, $Case2$, and $Case3$ in $Node3$).

First, the absolute speedup values on Node3 are lower than those observed on Node1 and Node2. That is primarily due to the high single-core performance of the Intel $i9-13900HX$ processor. Because the sequential baseline executes faster on this modern CPU, the speedup achieved through parallelization - whether on multiple CPU cores or on the GPU - is naturally reduced.

For the multi-core implementations, the results follow the expected behavior. The OpenMP version shows increasing speedup as the number of threads grows, scaling up to 8 threads. Beyond this point, performance gains begin to plateau due to the limitations of granularity. Even for Case 3, the problem size does not provide sufficient workload to utilize additional threads efficiently. Both the `DO CONCURRENT` and multi-core OpenACC implementations also deliver good

speedup when using the maximum number of threads, achieving performance comparable to OpenMP.

Regarding GPU-accelerated execution, two key observations stand out. First, the CUDA version, evaluated here for the first time, achieves speedup results similar to those obtained with the best OpenMP multi-core configuration. For *Case*3, the maximum OpenMP speedup (using 12 threads) is approximately 9.0×, while CUDA achieves 9.1×. That indicates that, for these domain sizes and on this specific architecture, the GPU does not significantly outperform a fully utilized multi-core CPU. The CPU's high single-thread performance, combined with the GPU's memory bandwidth limitations (as further discussed in the Roofline analysis), contributes to this parity.

Second, the OpenACC GPU implementation produces significantly smaller gains compared to the CUDA version. While CUDA achieves a 9.1× speedup for *Case*3, OpenACC on the GPU reaches only 5.3×. This performance gap of approximately 1.7× across all domain sizes is substantial and warrants further investigation. The consistency of this gap—it does not decrease significantly with larger workloads—suggests that the difference is not merely due to runtime overhead, but rather reflects how effectively each programming model utilizes the GPU's computational and memory resources. The Roofline analysis in the next section provides further insight into this behavior.

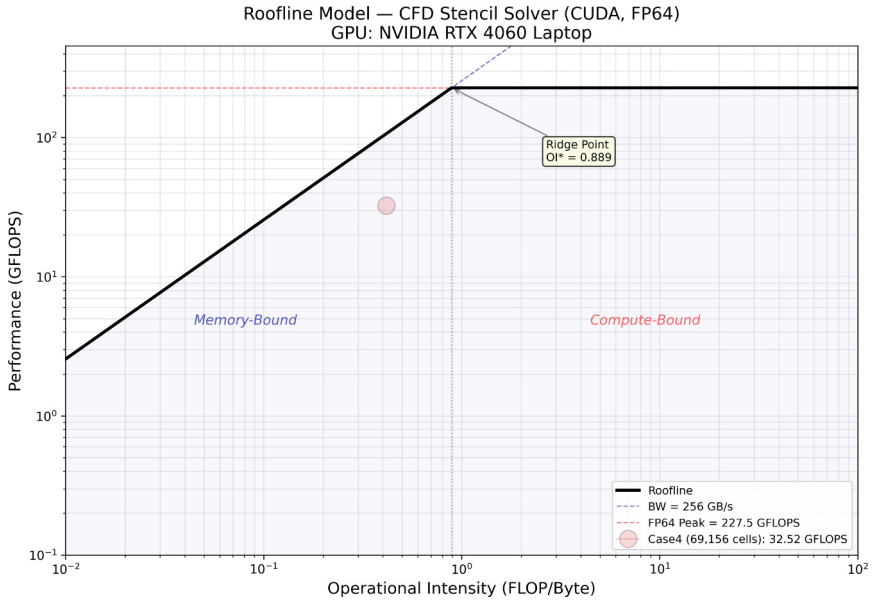

Fig. 9. Roofline model for the CUDA implementation of the CFD solver on the RTX 4060 Laptop GPU (Node3). The solver operates in the memory-bound region with an operational intensity of approximately 0.42 FLOP/Byte.

Roofline Model Analysis on Node3. To quantitatively understand the performance characteristics of the GPU implementation, the Roofline performance model is employed for the largest problem size ($Case4$) on the $Node3$ architecture. Figure 9 presents the Roofline plot for the CFD solver, allowing us to determine whether the application is limited by memory bandwidth or computational throughput.

The RTX 4060 GPU in $Node3$ has a theoretical peak double-precision performance of approximately 227.5 GFlop/s and a peak memory bandwidth of 256 GB/s. This results in a ridge point—where the application transitions from memory-bound to compute-bound—at an arithmetic intensity of approximately 0.89 Flop/Byte. The operational intensity of the solver was estimated by counting the floating-point operations and memory transfers performed in the main stencil kernels. Considering all solver stages, including the QUICK discretization, central-difference computations, and RungeKutta time integration, the overall operational intensity is approximately 0.42 Flop/Byte.

Since this value lies well below the ridge point, the application is clearly memory-bound. This behavior is characteristic of stencil-based CFD solvers, where each grid point update requires loading multiple neighboring values from memory while performing a relatively small number of arithmetic operations.

Based on the Roofline model, the maximum attainable performance for this operational intensity is approximately 106.8 GFlop/s, determined by the memory bandwidth limit (256 GB/s $\times$ 0.42 Flop/Byte). The measured performance of the CUDA implementation is approximately 32.5 GFlop/s, corresponding to about 30% of the memory-bandwidth-limited Roofline performance. This result confirms that the solver is primarily constrained by memory traffic rather than computational throughput. Consequently, further performance improvements are unlikely to be achieved through floating-point optimizations alone.

From an optimization perspective, since the application is memory-bound, further improvements should focus on increasing arithmetic intensity and improving memory access efficiency. Potential strategies include (1) increasing data reuse through techniques such as loop tiling or temporal blocking, (2) ensuring fully coalesced memory accesses in GPU kernels, and (3) leveraging the large L2 cache available on the RTX 4060 to reduce DRAM traffic. Given the intrinsic characteristics of stencil-based CFD workloads, the current performance already reflects the fundamental memory bandwidth limitations of the architecture.

6 Conclusion and Future Work

This paper improves an application for simulating free flow parallel to a porous plane. This application can solve problems such as soil's rain absorption and flood simulation. We described the code organization and proposed changes to parallelize it. An analysis of convergence for the timestep iterations and the relationship between the residual errors, was also presented.

We discussed several implementations using the parallel programming interfaces OpenMP, OpenACC, CUDA, and Fortran `DO CONCURRENT` directive. The

results show performance gains of 9.43× using 12 OpenMP threads for $Case3$, on $Node1$. The multi-core OpenACC version presents results alongside OpenMP. The `DO CONCURRENT` version shows that the results are more distant from OpenMP.

For GPUs, CUDA and OpenACC are good alternatives. CUDA can explicitly specify parallel operations, whereas OpenACC depends on directive injection and compiler decisions. The speedup achieved for $Case4$ with OpenACC was 24.

More running tests are needed to establish other performance relationships. For example, it is relevant to evaluate larger case studies, especially for GPU executions or faster cores. We intend to implement an OpenMP target for GPU in future work to establish a third GPU approach. The target approach allows the selection of the parallel environment at compilation time.

Acknowledgments. This work was supported by FCT - Fundação para a Ciência e Tecnologia, I.P. by projects: CeDRI, UID/05757/2025 (DOI: 10.54499/UID/05757/2025) and UID/PRR/05757/2025 (DOI: 10.54499/UID/PRR/ 05757/2025); SusTEC, LA/P/0007/2020 (DOI: 10.54499/LA/P/0007/2020); by CNPq: 10/2023 Universal (Project N° 407827/2023-4); and by FAPERGS: PROBIC scientific initiation scholarship.

References

1. ALHafez, N., Kurdi, A.: Parallel paradigms in modern HPC: a comparative analysis of MPI, OpenMP, and CUDA. arXiv preprint arXiv:2506.15454 (2025)
2. Civan, F.: Porous Media Transport Phenomena. Wiley, Hoboken, NJ, United States (2011)
3. Cornelissen, P.: Coupled free-flow and porous media flow: a numerical and experimental investigation. Ph. D. thesis, Faculty of Geosciences - Utrecht University (2016)
4. Ferziger, J.H., Perić, M., Street, R.L.: Finite Volume Methods, pp. 81–110. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-319-99693-6_4
5. Karimi, M., Bhattacharya, K.: Accelerated computational micromechanics for solute transport in porous media. Comput. Methods Appl. Mech. Eng. **426**, 116976 (2024)
6. Khalilov, M., Timoveev, A.: Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. J. Phys. Conf. Ser. **1740**(1), 012056 (2021)
7. Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., Sadayappan, P.: Effective automatic parallelization of stencil computations. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pp. 235–244. Association for Computing Machinery, New York, NY, USA (2007)
8. Kundu, P., Cohen, I., Hu, H.: Fluid Mechanics. Academic Press (2002)
9. Liritzis, I.: Disasters and climatic phenomena today and in the past. Proc. Eur. Acad. Sci. Arts **2**(1) (2023). https://doi.org/10.4081/peasa.22, https://www.peasa.eu/site/article/view/22

10. Schepke, C., Spigolon, R.A., Rufino, J., Cristaldo, C.F.D.C., Pizzolato, G.L.: Contributions to accelerating a numerical simulation of free flow parallel to a porous plane. In: 2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 226–233 (2025). https://doi.org/10.1109/PDP66500.2025.00038
11. Silva, H.U.D., Lucca, N., Schepke, C., Oliveira, D.P.D., Cristaldo, C.F.D.C.: Parallel OpenMP and OpenACC porous media simulation. J. Supercomput. **79**(8), 8425–8446 (2023).https://doi.org/10.1007/s11227-022-05004-2