POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Porting an HPC Multiphysics Finite Element Method code to GPU through Openacc

Laurea Magistrale in High Performance Computing Engineering

Author: Riccardo Selis

Advisor: Prof. Luca Formaggia

Co-advisor: Guillaume Hozeaux

Other contributors: Carlos Alvarez Martin, Yacine Rouis, Filippo Spiga

Academic year: 2024-2025

1. Introduction

1.1. Motivation

High-performance simulation is increasingly executed on *heterogeneous* supercomputing nodes, where general-purpose CPUs orchestrate the workflow while GPUs provide the bulk of throughput for data-parallel kernels. This shift is particularly relevant for finite-element multiphysics applications: realistic geometries, unstructured meshes, and tightly coupled physics push computational cost and energy-to-solution, making GPU enablement a strategic requirement for production codes.

This thesis is developed in the context of the Alya project at the Barcelona Supercomputing Center (BSC), within a team that targets scalable, robust, and maintainable HPC simulation. Alya represents a mature multiphysics finite-element platform where performance is determined not by a single kernel, but by the interaction between mesh/graph services, assembly, and iterative solvers. Porting such a code to GPUs is therefore not only about accelerating loops, but about preserving correctness and maintainability while adapting to the device execution and memory constraints. To address this challenge without fragmenting the codebase into architecture-specific versions, this work adopts a directive-based offload approach based on OpenACC. OpenACC enables incremental porting and step-by-step validation: compute regions can be moved to the device progressively, while controlling data residency explicitly to avoid transfer-dominated execution. In large Fortran applications, this is a practical path to performance portability because it allows GPU acceleration while retaining a single source tree and a CPU fallback execution path.

1.2. Contributions

The work presented here originates from an eight-month internship within the Alya development environment, where the host team identified a concrete engineering need: to advance GPU readiness in the *modern* Alya code line (modular and OOP-oriented) through a representative mini-application, and to extract reusable patterns that lower the cost and risk of future ports. The main contributions are:

1. **Reference GPU port of an implicit ADR mini-application:** OpenACC offload of the core assembly workflow in a compact unit driver that exercises essential Alya services (mesh, graph/CSR, assembly, and solver integration), providing a val-

idated baseline for future module ports.

2. **Profiling-driven performance characterization:** a structured methodology to measure, profile, and interpret GPU performance, including the impact of pack-based execution and tuning parameters (e.g., `VECTOR_SIZE`) on occupancy, memory behavior, and synchronization overheads.
3. **Reusable porting patterns for modern Fortran/OOP:** practical guidelines to manage derived types, pointers, and device data lifetimes with OpenACC, aimed at making GPU execution predictable and maintainable for subsequent Alya developments.

2. Alya

Alya is a massively-parallel multiphysics engineering simulation code developed at the Barcelona Supercomputing Center (BSC) since 2004 [20]. Designed from the outset for supercomputers, rather than retrofitted from a sequential legacy solver, it targets realistic engineering-grade workloads—complex geometries, unstructured meshes, and tightly coupled physics—while maintaining portability across modern architectures, including heterogeneous CPU/GPU systems [21]. With about one million lines of Fortran code, Alya supports a broad portfolio of PDE-based applications e.g. turbulent fluid dynamics, non-linear solid mechanics, heat transfer, reactive transport and chemistry, electrophysiology, and particle transport. Its relevance is demonstrated by strong scaling up to 100,000 processes on leadership-class systems for demanding multi-physics test cases, and by its inclusion in PRACE's UEABS benchmark suite, making it a representative platform for exascale-oriented multiphysics simulation with a strong emphasis on robustness, scalability, and performance portability, correctness and maintainability while adapting to the device exe- cution and memory constraints [1].

2.1. HPC computational workload organization

Alya achieves performance by *hierarchically* mapping the same simulation workload onto the parallel resources of the target machine. As illustrated in Fig. 1, the global mesh is first partitioned into MPI subdomains (distributed-memory parallelism). From this common decomposition, the execution path diverges: on CPUs, each MPI subdomain is further split for shared-memory parallelism and then *packed* into small homogeneous batches to maximize SIMD vectorization; on GPUs, the workflow skips the thread-subdomain stage and directly builds *large packs* that expose massive SIMT parallelism, with explicit data management and asynchronous streams to overlap transfers and kernel execution [6][4]. This design also makes communication costs predictable: assembly is largely local once the subdomain is defined, while the solver phase concentrates the unavoidable synchronization through halo exchanges (SpMV) and global reductions (dot-products). In this thesis, the work focuses exclusively on the GPU level of this hierarchy—optimizing the OpenACC/SIMT pack execution and related data-movement behavior—without addressing multi-GPU scaling with MPI (i.e., no MPI+GPU distributed runs were available in the experimental setup).

2.2. ADR model and Alya implicit implementation

We consider a scalar advection–diffusion–reaction (ADR) problem on a bounded domain $\Omega \subset R^d$, $d \in \{1, 2, 3\}$, with $\partial\Omega = \Gamma_D$. The unknown is $u(\mathbf{x})$ and the (dimension-agnostic) problem reads:

$$\begin{cases} \nabla\cdot(\rho\,\mathbf{a}\,u) - \nabla\cdot(k\,\nabla u) + r\,u = s & \text{in} \quad \Omega \\ u = u_D \text{ on} \quad \Gamma_D \end{cases}$$

To ensure robustness when advection dominates diffusion, Alya adopts a stabilized continuous Galerkin formulation. In this work the unit tests enable a SUPG-family method with a Codina definition of the stabilization parameter τ, built at Gauss points and used to weight residual and operator terms [5].

In Alya, the implicit ADR route converts into a finite-element formulation, leading to a sparse linear system $\mathbf{Ax} = \mathbf{b}$. Because advection generally yields a *non-symmetric* operator, the baseline solver used in this thesis is restarted GMRES with left preconditioning and a diagonal (Jacobi) preconditioner, which is robust and inexpensive for verification and profiling.

The implementation follows the standard element-kernel pipeline: (i) *gather* nodal coordinates/fields into element-local arrays, (ii) compute Jacobians, volumes and Cartesian derivatives at Gauss points, (iii) evaluate coefficients $\{\rho, \mathbf{a}, k, r, s\}$ (iv) build elemental matrix and RHS contributions , and (v) *scatter* into

the global CSR matrix and RHS. For validation, a manufactured solution is imposed through Dirichlet conditions on Γ_D, producing the dimension-consistent fields shown in Fig. 3 (2D and 3D), and providing an end-to-end check of boundary handling, assembly, and solver convergence within the same implicit ADR workflow.

3. Experimental Platform and Porting Methodology

The GPU experiments in this thesis were carried out on the Accelerated Partition of the MareNostrum 5 `MN-5` exascale supercomputer [19] at the Barcelona Supercomputing Center, using ACC [18] nodes that combine Intel Sapphire Rapids CPUs [8] with NVIDIA Hopper H100 GPUs [9] in a hybrid CPU–GPU environment. This platform provides both high compute throughput and high device memory bandwidth, making it suitable to evaluate directive-based offload under realistic HPC constraints. Within this setup, the porting strategy is to expose the fine-grain parallelism of the implicit ADR workflow and to keep its key data structures resident on the device, so that performance is not dominated by host–device transfers. Accordingly, this section first establishes a CPU-only baseline of the end-to-end ADR pipeline, then summarizes the OpenACC programming model adopted in Alya, and finally describes the incremental workflow used to reach a correct and tunable GPU execution path (from Unified Memory prototyping to explicit data management and profiling-guided optimization).

3.1. Performance of sequential `adr` solver on CPU

A CPU-only baseline was measured before introducing any GPU offload in order to understand where time is spent in an implicit ADR run and to motivate the porting priorities. Two Alya unit drivers were used, one for 2D and one for 3D, both solving a manufactured-solution problem on structured Cartesian meshes with the same workflow: mesh generation with boundary tagging, CSR graph construction and sparse matrix allocation, element and boundary assembly, and solution of the resulting linear system with restarted GMRES and diagonal preconditioning. Figure 3 reports the runtime breakdown into three macro phases (mesh, assembly, solve) together with GMRES iteration counts. Although the number of unknowns is matched between 2D and 3D (approximately 1M, 2M, 4M, and 8M), the profiles differ substantially. In 2D the solve phase dominates the runtime at all sizes, while in 3D the runtime is more balanced and preprocessing and assembly remain visible fractions of the total. The main driver of this difference is the iteration count. As shown in Fig. 3 (right), the 2D case requires one to two orders of magnitude more GMRES iterations than the 3D case

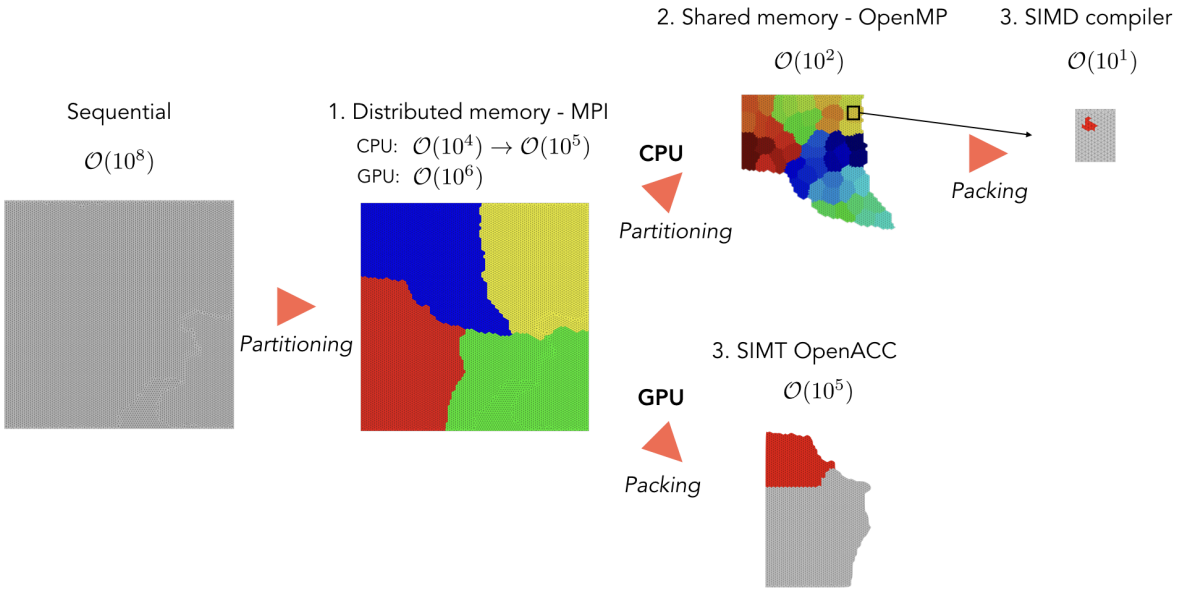

Figure 1: **Alya workload mapping on CPUs and GPUs.** Alya first partitions the mesh into MPI subdomains. CPU runs further split each subdomain for shared-memory parallelism and build small packs for SIMD vectorization, while GPU runs aggregate work into larger packs to expose SIMT parallelism. Adapted from [4].

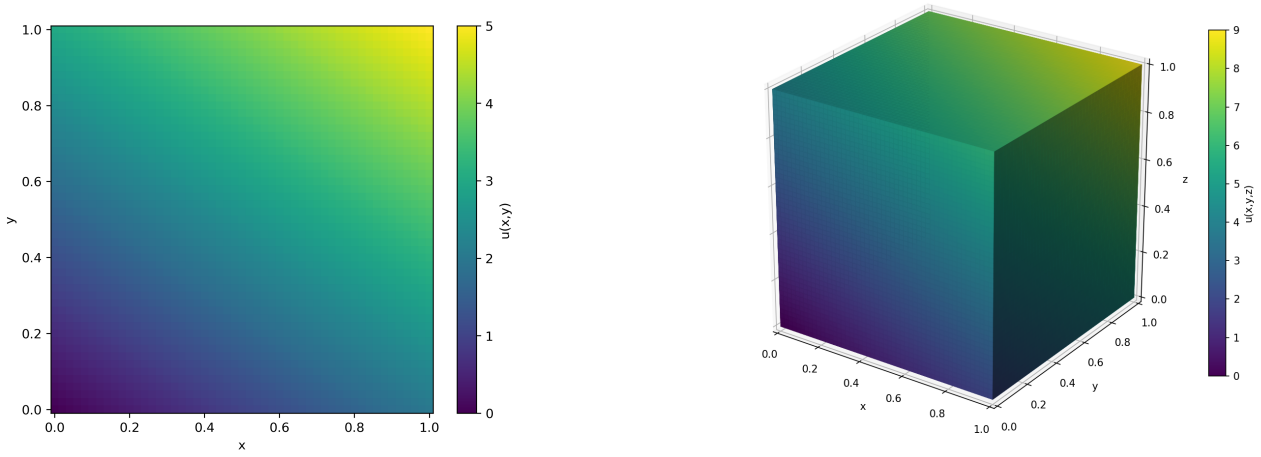

Figure 2: **Manufactured ADR solution on Cartesian grids.** Manufactured scalar field used to validate the implicit ADR setup with Dirichlet data prescribed from the analytical solution, $u(\mathbf{x})$ on $\Omega = [0, 1]^d$ where $N = 50$.

at comparable unknown counts. Since each iteration performs the same sparse linear algebra kernels, the higher iteration count translates directly into a much larger solver time, explaining why the 2D stacked bars are almost entirely solve time. In 3D, the iteration count remains relatively low, so the solver does not overwhelm the end-to-end runtime. A second effect is that, for the same number of unknowns, 3D incurs higher per-unknown costs in mesh/graph construction and in assembly. The 3D discretization produces a denser sparsity pattern and more expensive element-level work, which increases both the CSR construction effort and the assembly time. This explains why, in Fig. 3 (left), 3D retains a significant mesh and assembly share even at the largest problem size.

3.2. `OpenAcc` library

OpenACC is a directive-based programming standard for accelerating C/C++ and Fortran codes on GPUs. Instead of rewriting kernels in CUDA, the programmer annotates existing loops and regions (e.g., `!$acc parallel loop`) and an OpenACC-capable compiler generates the device code and runtime calls. OpenACC follows a simple operational model: (i) offload compute regions (`parallel`/`kernels`) and parallelize loops (ii) define data residency on the device (`enter data`/`data` and mapping clauses), (`loop` with optional `gang`/`vector`/`workers`), and (iii) correctness for shared updates (`atomic`) and explicit transfers when needed (`update`) [16] [15]. For performance, the most important principle is to keep frequently used arrays resident on the GPU across iterations and time steps, minimizing host–device transfers. OpenACC exposes a hierarchical parallel model: `gang` (coarse-grain independent groups) and `vector`/`workers` (fine-grain lanes, typically mapping to GPU SIMD/SIMT). In many cases the compiler can choose a mapping automatically, while explicit `gang`/`vector` clauses give more control at the cost of reduced portability.

3.3. Porting workflow

The porting followed an incremental, measurement-driven workflow. First, the ADR mini-application was offloaded using OpenACC directives with `OPENACC_MANAGED=ON`, relying on CUDA Unified Memory to obtain a working GPU execution path with minimal refactoring [2]. Next, NVIDIA Nsight Systems [12], instrumented with NVTX ranges, was used to validate offload coverage and to visualize unintended behavior such as host–device page migrations, explicit transfers, or fallback execution on the CPU. After establishing correctness, Unified Memory was progressively disabled and data management was made explicit: long-lived arrays were allocated on the device, transfers were restricted to the few required updates, and `present` policies were enforced. Finally, Nsight Compute and Nsight Systems (with NVTX) were used to analyze kernel performance and guide tuning [11].

4. Porting and Results

This section presents the core GPU porting outcomes and the associated performance results

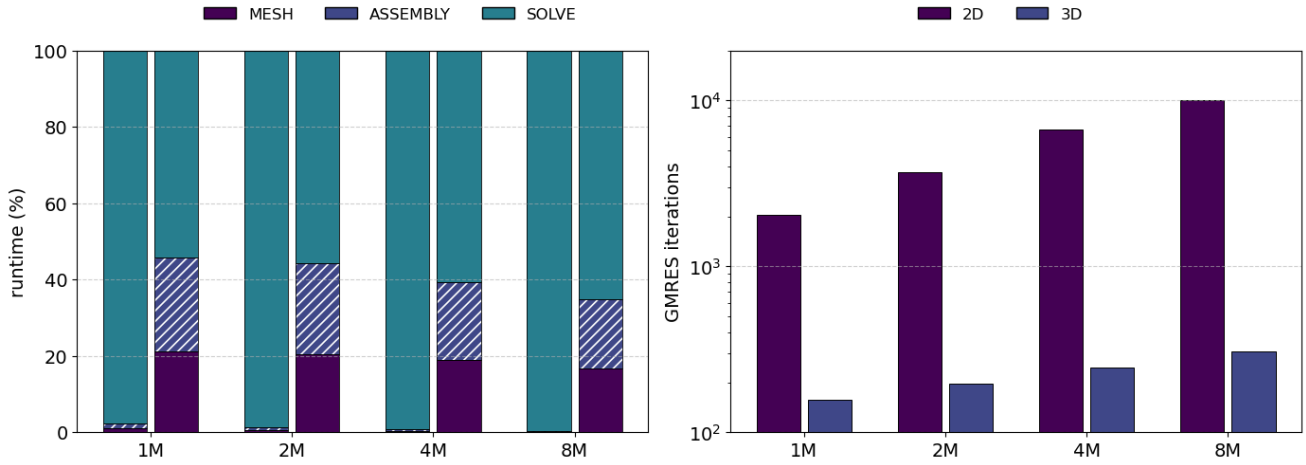

Figure 3: **Runtime breakdown and iters count on CPU.** Implicit ADR (GMRES, $\varepsilon = 10^{-6}$) runtime breakdown (left) and GMRES iteration counts (right) for 2D vs 3D at matched sizes (1–8M unknowns).

for the implicit ADR mini-application. The focus is on the assembly phase, since it concentrates the gather/compute/scatter workload and is strongly affected by GPU mapping choices and data-management decisions. We first describe Alya's pack-based assembly strategy and explain how the pack width parameter `VECTOR_SIZE` exposes fine-grain parallelism for both CPU SIMD and GPU SIMT execution. We then summarize the key portability challenge introduced by Alya's modern object-oriented design, namely the need to make pointer-based object graphs valid on the device through explicit deep-copy and `attach` patterns. Finally, we report and interpret the measured assembly speedups on 2D and 3D Cartesian test cases, highlighting why `VECTOR_SIZE` acts as a dominant tuning knob in 2D but shows weaker sensitivity in 3D, and relating these trends to profiler evidence collected within NVTX-instrumented ranges.

4.1. Parallelization strategy for Assembly

To keep a single implementation portable across CPUs and GPUs, Alya-ADR organizes element and boundary assembly in `packs` (batches) of fixed width, controlled by compile-time directive `VECTOR_SIZE`. Rather than assembling one element at a time, the code processes `VECTOR_SIZE` elements simultaneously following the usual gather/compute/scatter pattern. The lane index `ivect=1..VECTOR_SIZE` identifies the element handled by that lane. On CPUs, this lane loop is SIMD-friendly due to unit-stride lane-first storage; on GPUs, the same lane loop is mapped to threads with OpenACC, enabling a unified kernel structure. Varying `VECTOR_SIZE` directly changes the amount of fine-grain parallelism and the working-set footprint of per-pack temporaries. On GPUs, increasing `VECTOR_SIZE` typically improves throughput because it increases the number of active threads per kernel invocation, helping hide memory latency and amortize launch overhead; if too small, the kernel may underutilize the device [17]. On CPUs, excessively large `VECTOR_SIZE` can degrade performance by inflating temporary arrays and increasing cache/register pressure, while moderate values may improve auto-vectorization thanks to the unit-stride lane dimension. In practice, the best `VECTOR_SIZE` is therefore architecture-dependent: larger for GPUs to maximize occupancy and kernel's calls, smaller for CPUs to preserve locality.

Algorithm 1 Assembly controlled by `VECTOR_SIZE`

Require: Element packs `pack(ipack).l`, pack width `VECTOR_SIZE`
Ensure: Global RHS b and sparse matrix A updated

```
for ipack = 1...npack do
    list ← pack(ipack).l[1:VECTOR_SIZE]
    for all ivect = 1...VECTOR_SIZE in parallel do
        ielem ← list[ivect]
        if ielem = 0 then
            continue
        end if
        Gather: read nodal coords/fields of ielem
        Compute: build elrhs, elmat
        Scatter: add contributions to global b and A
    end for
```

This section quantifies how the OpenACC pack size (`VECTOR_SIZE`) affects GPU assembly performance, and why the response is markedly different in 2D and 3D. We report assembly

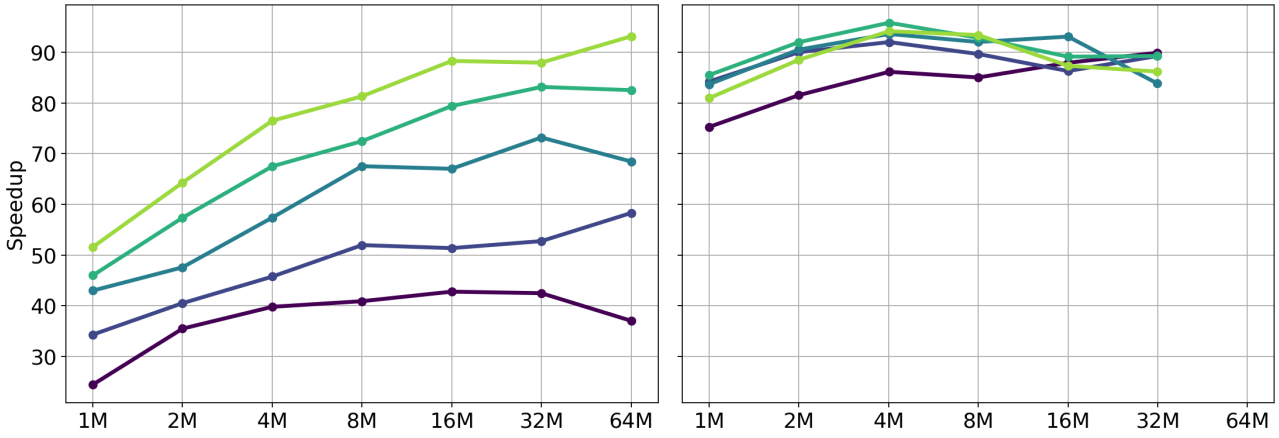

Figure 4: **Assembly speedup vs. problem size and pack size.** Assembly speedup (CPU time / GPU time) as a function of problem size for the 2D (left) and 3D (right) configurations under a `VECTOR_SIZE` sweep (64k–1024k). In 2D, increasing `VECTOR_SIZE` yields a clear and mostly monotonic speedup gain that becomes stronger as the mesh grows. In 3D, speedup is already high for small `VECTOR_SIZE` and shows only a weak dependence on packing, indicating earlier saturation of the dominant assembly kernels.

speedups from the internal timers and interpret the trends using qualitative evidence from Nsight Compute (NCU) collected within the assembly NVTX range.

4.2. Object-oriented data structures and OpenACC: manual deep copy with `attach`

Alya-ADR relies on object-oriented Fortran with nested derived types and pointer members (e.g., a solver object pointing to a mesh object, which in turn points to coordinates and connectivity arrays). On GPUs, a naïve `copyin` of a derived type is typically *shallow*: pointer components may still contain host addresses, and dereferencing them inside a device kernel can trigger illegal-address failures. To make the object graph valid on the device, the port uses a manual deep-copy strategy [13] [14] based on two steps: (i) create/copy the *leaf* data buffers (arrays) on the device, and (ii) apply `attach` to repair device-side pointer values so that device-resident parent objects reference device-resident children. Because `attach` does not move bytes, it must be executed *after* the target buffer is present on the device and in a bottom-up order (leaves → parents → root), recursively for nested objects (e.g., boundary data). This pattern preserves a single CPU/GPU codebase and provides deterministic correctness: kernels always observe valid device addresses while data residency can be optimized independently through explicit transfers.

4.3. Summary of results and explanation

Figure 4 shows that `VECTOR_SIZE` is a strong performance knob in 2D but a secondary one in 3D. In 2D, larger packs consistently reduce assembly time and the benefit grows with problem size, leading to a wide separation between curves. In 3D, all pack sizes achieve similarly high speedup and the curves remain close, with only small non-monotonic variations at large `VECTOR_SIZE`.
NCU analysis supports a batching-driven interpretation. Increasing `VECTOR_SIZE` reduces the number of pack iterations and therefore the number of kernel invocations inside the assembly NVTX range, effectively amortizing launch/runtime overheads. This effect matters more in 2D because assembly kernels are shorter (lighter per-element work), so fixed per-launch and runtime costs represent a larger fraction of the end-to-end assembly time; with larger packs, kernels execute in a more stable regime with improved sustained DRAM/memory throughput. In 3D, per-element work is heavier and dominant kernels are already long enough to operate near a throughput-limited steady state even at smaller packs; therefore, further increases in `VECTOR_SIZE` mainly reduce overheads that are already amortized, yielding diminishing returns. Residual differences in 3D can be explained by second-order effects observed in NCU (e.g., changes in register pressure/achieved occupancy and scatter contention), which can offset the modest gain from fewer launches.

References

[1] Application performance on accelerators. Technical Report D7.1.2, PRACE-1IP, 2012.

[2] *NVIDIA HPC Compilers User's Guide*, 2025.

[3] Ricard Borrell, J. C. Cajas, Daniel Mira, Alberto Taha, Seid Koric, Mariano Vázquez, and Guillaume Houzeaux. Parallel mesh partitioning based on space filling curves. *Computers & Fluids*, 173, 2018.

[4] Ricard Borrell, Alberto Taha, Damien Dosimont, Marta Garcia-Gasulla, Guillaume Houzeaux, Mariano Vázquez, Eduard López, Josep Jorba, and Pilar Quiles. Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture. *Future Generation Computer Systems*, 107, 2020.

[5] Ramon Codina. Stabilization of incompressibility and convection through orthogonal sub-scales in finite element methods. *Computer Methods in Applied Mechanics and Engineering*, 190, 2000.

[6] Guillaume Houzeaux, María Luisa de la Cruz, and Mariano Vázquez. Parallel uniform mesh subdivision in Alya. Technical Report WP197, PRACE, 2011. PRACE White Paper.

[7] Guillaume Houzeaux, Mariano Vázquez, Raymond Aubry, and José María Cela. A massively parallel fractional step solver for incompressible flows. *Journal of Computational Physics*, 228(17), 2009.

[8] Intel. Intel Xeon Platinum 8460Y+ — Product Specifications (ARK), 2026.

[9] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture (Hopper) — In-depth overview, 2022.

[10] NVIDIA. *CUDA C++ Best Practices Guide*, 2025.

[11] NVIDIA. *NVIDIA Nsight Compute: Profiling Guide*, 2025.

[12] NVIDIA. *NVIDIA Nsight Systems: User Guide*, 2025.

[13] OpenACC Organization. *Complex Data Management in OpenACC Programs*, 2014.

[14] OpenACC Organization. *Deep Copy Attach and Detach*, 2016.

[15] OpenACC Organization. Openacc best practices guide (porting and portability guide). GitHub repository, 2025.

[16] OpenACC-Standard.org. *The OpenACC Application Programming Interface (Version 3.3)*, 2022.

[17] Heath Owen, Dominik Ernst, Thomas Gruber, Oriol Lehmkuhl, Guillaume Houzeaux, Giancarlo Gasparino, and Gerhard Wellein. Alya towards exascale: Optimal OpenACC performance of the navier–stokes finite element assembly on GPUs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024.

[18] RES (Red Española de Supercomputación). MareNostrum 5 ACC — Node specification (ACC compute node), 2026.

[19] TOP500. MareNostrum 5 ACC — System entry, 2026.

[20] Mariano Vázquez, Guillaume Houzeaux, Seid Koric, Albert Artigues, Javier Aguado-Sierra, Rafael Arís, Daniel Mira, Hadrien Calmet, Fabiano Cucchietti, Heath Owen, Alberto Taha, Erin Dering Burness, José María Cela, and Mateo Valero. Alya: Multiphysics engineering simulation towards exascale. *Journal of Computational Science*, 14, 2016.

[21] Mariano Vázquez, Antonio Rubio, Guillaume Houzeaux, M. González, J. Giménez, and V. Beltrán. Xeon phi performance for HPC-based computational mechanics codes. PRACE technical report / dataset record, 2014.