POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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

TESI DI LAUREA MAGISTRALE IN
HIGH PERFORMANCE COMPUTER ENGINEERING

Author: **Riccardo Selis**

Student ID: 293375
Advisor: Prof. Luca Formaggia
Co-advisors: Guillaume Hozeaux, Carlos Alvarez Martin
Academic Year: 2024-205

# Abstract

Modern HPC systems increasingly rely on heterogeneous CPU/GPU nodes, making performance portability a central requirement for production-grade simulation codes. This thesis addresses the GPU acceleration of the implicit Advection–Diffusion–Reaction (ADR) workflow in Alya, a large-scale multiphysics finite-element code developed at the Barcelona Supercomputing Center. The work targets the assembly stage, which is a major cost driver in many implicit runs due to its gather/compute/scatter structure on unstructured data and its sensitivity to memory bandwidth and write conflicts. A single-source porting strategy based on OpenACC is presented, designed to preserve Alya's modular architecture and object-oriented, pointer-rich data structures. Assembly is reorganized around packs of homogeneous elements: a coarse-grain loop over packs exposes scalable parallel work, while a fine-grain lane dimension provides the portability bridge, mapping to SIMD on CPUs and to SIMT threads on GPUs. To enable correct device execution with nested derived types, the thesis details a practical deep-copy workflow based on explicit data regions and systematic pointer attachment. Performance is evaluated on MareNostrum 5 accelerated nodes with NVIDIA H100 GPUs. Results show that the OpenACC pack parameter (`VECTOR_SIZE`) acts as a granularity knob: in 2D, larger packs reduce launch/runtime overheads and yield strong assembly speedup gains, while in 3D assembly quickly becomes bandwidth-oriented and further increases provide diminishing returns and may trigger resource-pressure effects.

**Keywords:** GPU, OpenACC, FEM, HPC, Porting.

# Abstract in lingua italiana

I moderni sistemi HPC adottano sempre più nodi eterogenei CPU/GPU, rendendo la portabilità prestazionale un requisito chiave per i codici di simulazione in produzione. Questa tesi affronta l'accelerazione su GPU del workflow implicito di Advection–Diffusion–Reaction (ADR) in Alya, codice multifisico ad elementi finiti sviluppato al Barcelona Supercomputing Center. L'attenzione è rivolta alla fase di assembly, spesso dominante nei run impliciti per via della struttura gather/compute/scatter su dati irregolari e della forte dipendenza da banda di memoria e conflitti in scrittura. Viene proposta una strategia single-source basata su OpenACC, pensata per mantenere l'architettura modulare di Alya e le sue strutture dati object-oriented ricche di puntatori. L'assembly viene riorganizzato in pack di elementi omogenei: un livello esterno sui pack espone parallelismo coarse-grain, mentre una dimensione interna di lane funge da ponte di portabilità, mappandosi a SIMD su CPU e a thread SIMT su GPU. Per garantire la correttezza con tipi derivati annidati, la tesi descrive un workflow di deep copy basato su regioni dati esplicite e su una procedura sistematica di attach dei puntatori. Le prestazioni sono valutate su nodi accelerati di MareNostrum 5 con GPU NVIDIA H100. I risultati mostrano che il parametro di pack OpenACC (`VECTOR_SIZE`) controlla la granularità: in 2D pack più grandi riducono overhead di launch/runtime e aumentano significativamente lo speedup dell'assembly; in 3D l'assembly entra rapidamente in un regime limitato dalla banda e ulteriori incrementi offrono benefici marginali, talvolta penalizzati da effetti di pressione sulle risorse.

**Keywords:** GPU, OpenACC, FEM, HPC, Porting.

# Contents

# 1 | Introduction

This thesis addresses the GPU enablement of a representative implicit finite-element workflow within Alya, a production multiphysics simulation code developed at the Barcelona Supercomputing Center (BSC). The motivation stems from the widespread adoption of heterogeneous CPU–GPU nodes in modern HPC systems: while GPUs offer high throughput for data-parallel kernels, large multiphysics codes must achieve these gains without sacrificing robustness, maintainability, or portability. Within this context, the thesis focuses on performance engineering aspects that determine whether GPU acceleration is effective in practice, with particular emphasis on the element and boundary assembly stage and its interaction with data movement and solver execution.

The scope of the work is intentionally constrained. The study targets single-node GPU execution using OpenACC directive-based offload, prioritizing a unified code path that remains valid on CPU-only platforms. The porting and analysis are conducted on a compact ADR mini-application that exercises Alya services (mesh, graph/CSR, assembly, solver integration) while remaining small enough for iterative debugging and profiling. Multi-GPU MPI+GPU scaling and algorithmic redesign of the underlying numerical methods are outside the scope; the objective is to produce a correct, tunable, and reusable GPU reference implementation and to extract practical patterns for future ports in the modern OOP-oriented Alya code line.

## 1.1. Motivation: heterogeneous HPC and performance portability

High-performance computing (HPC) is undergoing a structural transition driven by two converging pressures: the increasing computational intensity of scientific and engineering workloads, and the growing importance of energy efficiency and throughput per watt. In this context, modern leadership systems are rarely built around a single type of processor. Instead, they adopt *heterogeneous nodes* that combine general-purpose CPUs with throughput-oriented accelerators, most commonly GPUs. For simulation codes targeting realistic geometries, large unstructured meshes, and multi-physics coupling, this shift turns GPU enablement into a strategic requirement: the ability to exploit accelerators effectively is increasingly intertwined with the

ability to remain competitive on flagship platforms. At the same time, heterogeneous hardware exacerbates a long-standing tension between *portability* and *performance*. Large production codes must evolve across multiple generations of systems, vendors, and programming environments, while preserving long-term maintainability and numerical robustness. This motivates a development approach that aims not only to run across platforms, but also to deliver a meaningful fraction of available performance without fragmenting the codebase into architecture-specific variants. In other words, the relevant goal is *performance portability*, rather than portability alone.

### 1.1.1. Heterogeneous nodes as the mainstream for HPC simulation

The current HPC landscape reflects a broad adoption of accelerator-based architectures for top-tier systems. Heterogeneous nodes typically assign distinct roles to CPUs and GPUs: CPUs orchestrate control flow, handle latency-sensitive tasks, and manage the global workflow, while GPUs provide massive parallel throughput for compute- and memory-intensive kernels. This division aligns well with many building blocks of implicit PDE solvers—sparse linear algebra, stencil-like operators, and element-level computations—provided the code can expose enough parallelism and sustain high memory throughput. However, heterogeneity also changes the practical rules for performance. On GPUs, the performance envelope is often limited by memory behavior (bandwidth, latency hiding, and access regularity) and by the ability to keep large working sets resident on the device. Consequently, end-to-end speedup cannot be assumed from simply offloading compute loops: it depends on restructuring hot paths to increase concurrency, reduce unnecessary host–device transfers, and avoid synchronization bottlenecks. For production simulation codes, these changes must be introduced while preserving numerical results, stability, and the ability to run efficiently on CPU-only platforms.

### 1.1.2. Performance portability in large multiphysics codes

In large multiphysics applications, performance portability is not a single technique but an engineering objective that must be pursued across software layers: numerical kernels, data structures, parallel runtime, and build/deployment environment. Conceptually, performance portability requires two properties: (i) the application supports a set of target platforms without extensive rewrites, and (ii) it achieves a non-trivial level of performance on each supported platform. A key challenge is that different architectures favor different trade-offs. For example, optimizing for CPU cache locality and SIMD vectorization can conflict with GPU requirements for massive threading, latency hiding, and coalesced memory access patterns. When combined with irregular access patterns typical of unstructured discretizations, the gap between a “cor-

rect" port and a *high-performance* port can be substantial. Because qualitative statements about portability are easy to make and hard to compare, the literature has also proposed quantitative ways to reason about performance portability across platforms. These approaches emphasize that portability should be evaluated on an explicit set of architectures and that performance claims should be grounded in measurable baselines. For a thesis grounded in production code, this perspective is useful: it encourages a methodology where baseline measurements, profiling, and controlled tuning are used to separate algorithmic limitations from implementation bottlenecks.

### 1.1.3. Thesis focus: GPU-level optimization with directive-based offload

Within this broader context, this thesis focuses on GPU execution of an implicit finite-element workflow, with the goal of improving performance while preserving a unified source code that remains viable on CPU-only systems. The adopted strategy relies on directive-based accelerator programming, which enables *incremental porting*: compute regions can be offloaded step-by-step, correctness can be validated at each stage, and the same code can remain compilable across heterogeneous environments. This approach is particularly appealing for large Fortran codebases where a full rewrite into a low-level programming model would be costly and difficult to sustain. The focus of the work is therefore not on introducing new numerical algorithms, but on addressing the performance-engineering aspects that determine whether GPU acceleration is effective in practice: mapping fine-grain parallelism to GPU threads, selecting appropriate batch sizes for kernel execution, and controlling data residency and movement to avoid transfer-dominated execution. The resulting analysis and optimizations aim to produce both (i) concrete improvements on the target GPU platform and (ii) general lessons on how directive-based offload interacts with the performance characteristics of implicit assembly workflows.

## 1.2. Alya as a production multiphysics platform

This thesis is developed within the context of *Alya*, a production-grade multiphysics simulation code designed for large-scale HPC environments. Alya is representative of an important class of scientific software: long-lived, continuously evolving, and required to deliver both numerical robustness and high performance across multiple generations of supercomputing architectures. In such codes, the main difficulty is rarely confined to a single kernel. Instead, performance emerges from the interaction between discretization, data structures, mesh/graph services, assembly routines, and iterative solvers. For this reason, Alya provides an ideal setting to study GPU acceleration under realistic constraints, including maintainability, portability, and compat-

ibility with an existing ecosystem of modules.

### 1.2.1. Alya in context: production code, robustness, scalability

Alya is a massively-parallel code aimed at engineering and applied-science workloads that involve complex geometries, unstructured meshes, and tightly coupled physical models. Its design emphasizes scalability and robustness: the code is intended to run efficiently on distributed-memory machines, while also exploiting node-level parallelism on modern CPU and GPU nodes. Beyond its scientific scope, Alya is particularly relevant as a *platform*: it incorporates mature infrastructure for mesh handling, partitioning, matrix graph construction, I/O, solver services, and multi-physics coupling. These capabilities are precisely what differentiates production simulation codes from simplified research prototypes, and they also shape what is feasible when introducing GPU offload. From the performance-engineering perspective, Alya is interesting because it embodies a hierarchical view of parallelism. The global problem is decomposed into subdomains for distributed-memory execution, and within each subdomain the dominant workloads (e.g., element and boundary assembly, sparse linear algebra) must be mapped efficiently onto the node architecture. This hierarchy becomes more challenging on heterogeneous systems: the same high-level algorithmic workflow must remain valid, while the best-performing low-level mapping may differ substantially between CPUs and GPUs. Consequently, GPU enablement in Alya is not merely a question of rewriting compute loops; it requires addressing data residency, memory-access patterns, synchronization, and the interaction between the offloaded kernels and the surrounding runtime services.

### 1.2.2. ADR implicit workflow as a representative target workload

The specific workload considered in this thesis is an implicit advection–diffusion–reaction (ADR) route, selected as a representative proxy for a broader class of finite-element PDE solvers. The ADR pipeline exercises the core building blocks that often dominate end-to-end runtime in implicit simulations: (i) mesh and graph preparation, (ii) element and boundary assembly into global sparse structures, and (iii) iterative solution of the resulting linear system. In particular, assembly follows the canonical finite-element pattern *gather/compute/scatter*. Field values and coordinates are gathered into element-local arrays; quadrature-point computations evaluate operators and stabilization terms; and the resulting local contributions are scattered to global vectors and matrices. This workflow is an effective target for GPU studies for three reasons. First, it features abundant fine-grain parallelism across elements and quadrature points, which in principle maps well to SIMT execution. Second, it also exposes the classical GPU pain points of production FEM codes: irregular memory access in the gather phase and po-

tential write conflicts in the scatter phase (often requiring atomic updates or conflict-avoidance strategies). Third, it links naturally to the solver stage: even if assembly is accelerated, the overall benefit depends on how much time is ultimately spent in sparse linear algebra kernels and on whether data can remain resident on the device across the full solve loop. For these reasons, an implicit ADR mini-application is not only useful for validation, but also for performance characterization and for extracting guidelines that can generalize to other Alya modules with similar assembly structure.

### 1.2.3. Context of this work: internship-driven objective within the Alya team

The work presented in this thesis originates from an eight-month internship carried out within the Alya development environment, where the host team identified a concrete performance-engineering need: to advance GPU readiness in a way that is technically sound, maintainable, and reusable for future ports. In this context, two internal code lines co-existed during development. A *legacy* version features a more monolithic organization, heavier reliance on global state, and limited use of object-oriented abstractions. A more *modernized* version adopts a cleaner modular structure and object-oriented programming (OOP) practices. While GPU ports of legacy-style codes can sometimes be more straightforward (because data is flatter and more explicit), extending GPU support to the modern OOP-based design is strategically important for long-term evolution and maintainability. A major practical obstacle is that directive-based GPU programming interacts non-trivially with OOP and pointer-rich data structures. Derived types with pointer members, dynamic allocation patterns, and deep object graphs require careful data mapping and pointer attachment strategies to ensure that device-side references are valid and that the runtime does not fall back to unintended host transfers. This difficulty motivated the internship task: to port and stabilize a compact but representative ADR mini-application in the modernized code line, and to document the patterns required to make OOP and pointers work reliably with OpenACC offload. The resulting contribution to the research group is therefore twofold. First, the ADR mini-application provides a *reference implementation* for future ports: a self-contained path that exercises mesh services, assembly, and solver integration while remaining small enough to iterate quickly during debugging and profiling. Second, the work consolidates a set of *practical porting patterns* for the modern Alya infrastructure, including strategies to manage derived types, pointer attachment, and device data lifetimes. Together, these outcomes reduce the risk and effort of porting additional modules by offering a tested baseline, a troubleshooting guide grounded in real failures, and a validated example of how to reconcile OpenACC offload with a modular OOP-oriented Fortran code structure.

## 1.3. Technical challenges in GPU porting of implicit assembly workflows

Porting an implicit finite-element assembly workflow to GPUs is rarely a matter of "moving loops to the device". In production codes, assembly sits at the intersection of three difficult concerns: (i) irregular data access dictated by mesh connectivity, (ii) concurrent updates to shared global structures, and (iii) device data management across multi-stage implicit pipelines (assembly + iterative solve). This section summarizes the main technical challenges that shape the design choices in the remainder of the thesis.

### 1.3.1. Gather/compute/scatter, irregular access, and atomic updates

Element and boundary assembly follow a common pattern: *gather* degrees of freedom and geometric data into element-local arrays, *compute* operator terms at quadrature points, and *scatter* the resulting local contributions into global vectors and sparse matrices. While the compute phase is typically regular and arithmetic-heavy, gather and scatter are driven by unstructured connectivity (indices in `lnods`, local-to-global maps, CSR row/column structures). On GPUs, this translates into memory access patterns that are difficult to coalesce and that tend to be latency-dominated. As a consequence, achievable throughput is often limited by memory behavior rather than floating-point capability. The scatter phase introduces an additional, fundamental complication: multiple elements can contribute to the same global row/entry, creating write conflicts. Traditional CPU strategies to mitigate this include reordering, coloring, and blocking schemes, but these approaches can increase preprocessing complexity and may reduce flexibility in a modular codebase. A common pragmatic approach on GPUs is to use atomic operations for the conflicting updates, ensuring correctness under massive parallelism. Atomics, however, introduce serialization at hot global locations, potentially creating performance cliffs when contention becomes significant. The severity of this effect depends on mesh topology, element type, sparsity structure, and the chosen parallel mapping (how many threads concurrently target the same rows). Finally, correctness requirements in implicit workflows amplify the impact of scatter semantics. Assembly is typically repeated across time steps and non-linear iterations, and it must preserve numerical stability and convergence properties of the solver. Therefore, any GPU strategy must ensure that parallel updates remain mathematically consistent with the CPU reference, while accepting that floating-point non-associativity and different reduction orders may yield small round-off differences.

### 1.3.2. Pack-based parallelism and VECTOR_SIZE trade-offs on GPUs

Alya organizes assembly into packs (batches) of elements processed in a unified kernel structure, controlled by a pack width parameter (VECTOR_SIZE). This design is attractive because it provides a single implementation that can target both CPU vectorization and GPU SIMT execution by mapping the "lane" dimension to the most appropriate hardware parallelism. On GPUs, VECTOR_SIZE becomes a first-order performance parameter because it directly controls the amount of exposed fine-grain parallelism per kernel invocation, and therefore influences occupancy, latency hiding, and the amortization of kernel launch overhead. The trade-off is that larger packs increase the working-set footprint of per-pack temporaries (element-local arrays, quadrature intermediates, stabilization terms) and may increase register pressure. High register pressure can reduce occupancy, counteracting the benefit of additional threads. Conversely, packs that are too small may underutilize the device, leaving memory latency insufficiently hidden and making performance sensitive to launch overheads and runtime scheduling effects. In practice, the optimal choice depends on the balance of (i) memory intensity of gather/scatter, (ii) compute intensity at quadrature, and (iii) the overhead and contention introduced by atomic scatter updates. A second, closely related aspect is *data layout*. Pack-based designs typically store element-local data with the lane dimension as the unit-stride index to favor SIMD on CPUs. On GPUs, the same layout can be beneficial if consecutive threads access consecutive lane entries, enabling more regular memory transactions. However, when global indices dominate (e.g., gather via indirect addressing), the benefit may be partially obscured. For this reason, VECTOR_SIZE tuning must be interpreted together with hardware-centric metrics such as achieved memory bandwidth, occupancy, and stall reasons. The thesis therefore treats VECTOR_SIZE not as a pure "knob" but as an embodiment of an architectural mapping decision, whose effects differ between problem dimensions and mesh configurations.

### 1.3.3. Data residency and Fortran/OOP mapping constraints

For implicit workflows, accelerating the assembly kernel alone is insufficient if data is repeatedly moved between host and device. GPUs deliver performance when large, frequently used arrays remain resident across iterations and time steps. This shifts attention to the *data lifetime* of mesh arrays, coefficient fields, CSR structures, and solver vectors. A directive-based approach makes it possible to introduce GPU regions incrementally, but it also requires disciplined control of data mapping so that the runtime does not perform implicit transfers that negate performance gains.

In modern Fortran codebases, the challenge is compounded by object-oriented programming and pointer-rich structures. Derived types may contain allocatable members and pointers that

reference dynamically allocated storage. Even when top-level objects are present on the device, internal pointers may still reference host addresses unless explicit attachment or deep-copy strategies are applied. This can lead to failure modes that are subtle: code may compile successfully yet produce incorrect results or trigger unintended host access at runtime. Ensuring correct device-side object graphs therefore requires explicit policies for allocation, data region scoping, and pointer attachment, along with a clear separation between *host-only orchestration* and *device-resident compute data*. Within the scope of this thesis, these constraints are not incidental: they motivate the use of a compact ADR mini-application as a reference for future ports. By forcing the port to coexist with modular services (mesh, graph construction, solver interfaces) while relying on OOP abstractions, the mini-application becomes a realistic stress test of data management practices. The outcome is a set of patterns that make the execution path predictable: data remains resident across the implicit loop, updates are explicit and minimal, and the device view of OOP structures is validated as part of the correctness workflow.

## 1.4. Contributions

The work presented in this thesis is the outcome of an eight-month internship carried out within the Alya development environment, with the objective of producing results that are directly useful to the host team's GPU porting roadmap. Rather than introducing a new numerical method, the thesis delivers a set of *engineering contributions* that improve GPU readiness and provide a concrete reference for future module ports—especially in the context of modern Fortran code organization with object-oriented constructs and pointer-rich data structures.

### 1.4.1. OpenACC GPU offload of ADR assembly in Alya unit drivers

A first contribution is the GPU enablement of a compact implicit ADR mini-application, designed to exercise the essential services required by a PDE module (mesh handling, graph/CSR structures, assembly, and solver integration) while remaining small enough to iterate rapidly during debugging and profiling. This mini-app was developed as a *reference path* for the modern Alya code line, which emphasizes modularity and object-oriented programming.

Concretely, the work provides:

1. a consistent OpenACC offload strategy for the ADR assembly workflow, aligned with Alya's pack-based execution model;
2. a correctness-oriented integration that maintains a CPU fallback path and supports step-by-step verification (e.g., manufactured-solution tests and end-to-end convergence checks);

3. a practical resolution of GPU-related issues arising from modern Fortran features (derived types, allocatables, pointers), ensuring that device-side data views are valid and stable throughout the implicit workflow.

From the group perspective, the mini-app acts as a controlled, reproducible test vehicle that demonstrates how OpenACC can be applied to a modern Alya module without reverting to legacy-style global-state designs.

### 1.4.2. Profiling-driven tuning and performance characterization

A second contribution is a systematic methodology to quantify and explain performance on the target GPU execution path. The thesis establishes CPU baselines, introduces incremental offload, and then applies profiling-driven tuning to identify limiting factors and performance opportunities. Particular attention is devoted to the architectural meaning of the pack width parameter `VECTOR_SIZE`, which directly controls fine-grain parallelism, kernel occupancy, and the working-set footprint of per-pack temporaries.

This contribution includes:

- an end-to-end runtime breakdown of the implicit ADR pipeline (mesh/graph, assembly, solve) to prioritize optimization effort under realistic execution;
- a controlled exploration of `VECTOR_SIZE` on GPUs, connecting observed speedups/slowdowns to measurable effects such as kernel launch configuration, occupancy, memory behavior, and the overhead of synchronization (e.g., atomics in scatter updates);
- a profiler-guided interpretation workflow that links kernel-level metrics to application-level outcomes, enabling performance conclusions that are diagnostic ("why it happens") rather than only descriptive ("what happens").

The resulting characterization is intended to be reusable by the team as a template for evaluating other modules: baseline first, controlled changes second, and profiler evidence to support tuning decisions.

### 1.4.3. Practical guidelines and lessons learned for future Alya GPU work

A third contribution is the consolidation of practical porting guidelines that reduce friction for future GPU work in the modern Alya code line. The focus is on reliability and predictability: avoiding unintended host–device transfers, ensuring correct device data lifetimes, and managing pointer-based structures in a way that remains maintainable within a modular OOP-oriented architecture.

The thesis distills:

- a set of data-residency rules for implicit workflows (what must remain resident across iterations and time steps, what can be transient, and what transfers must be explicit);
- a set of repeatable patterns for handling derived types and pointers in OpenACC contexts (including device allocation scope, attachment practices, and debugging checks to detect invalid device-side references early);
- a troubleshooting and profiling playbook for iterative porting, emphasizing incremental validation and the interpretation of profiling evidence to distinguish memory limits, synchronization limits, and occupancy-related bottlenecks.

Overall, these guidelines are meant to turn the ADR mini-app into more than a single successful port: it becomes a reference implementation and a practical knowledge base that supports porting additional Alya modules with similar gather/compute/scatter structure and comparable data-management complexity.

## 1.5. Thesis organization

This thesis is structured to first introduce the GPU programming background, then present the Alya software context, and finally describe the target platform, the ADR problem, and the profiling/porting results.
Chapter 2 introduces the GPU programming background needed to interpret the design choices and performance results, covering the CUDA execution model, the NVIDIA GPU memory hierarchy, and the OpenACC offload approach adopted in this work. Chapter 3 introduces Alya and the specific execution workflow relevant to this thesis, with emphasis on the code architecture and on how the computational workload is organized for HPC systems. Chapter 4 defines the experimental context and the target problem: the MareNostrum 5 ACC node architecture, the implicit ADR mini-application used for validation, and the CPU-side profiling baseline. Chapter 5 presents the GPU porting strategy and the performance results, focusing on pack-based assembly, the effect of `vector_size`, and the interpretation supported by profiling evidence. Chapter 6 concludes the thesis by summarizing the contributions and main findings, and by outlining the most relevant future directions, including multi-GPU and multi-node extensions and energy-oriented comparisons.
Readers primarily interested in GPU programming concepts may focus on Chapter 2 and then proceed directly to Chapters 4 and 5 for the experimental setup and results. Readers primarily interested in Alya may start from Chapter 3 and then continue with Chapters 4 and 5, using Chapter 2 as background when clarifications on the GPU execution/memory model or on Ope-

nACC are needed. Readers mainly interested in performance outcomes may start from Chapter 5 and use Chapters 2–4 as reference material for the programming model, the Alya workflow context, and the problem/platform description.

To support reproducibility and future extensions, the experimental setup and measurement methodology are documented where they are most relevant. Chapter 4 reports the target platform configuration and the adopted toolchain, together with the validation workflow and the CPU profiling baseline used as reference. Chapter 5 specifies the problem configurations used in the performance campaigns (problem sizes, tolerances, and tuning parameters such as `VECTOR_SIZE`) and defines the timing and speedup metrics used for comparison. Where appropriate, implementation-relevant details (offloaded regions, data-residency decisions, and profiling collection procedures) are described alongside the corresponding results so that the reasoning behind the conclusions is traceable.

# 2 | GPU Parallel Programming and OpenAcc

This chapter introduces the GPU hardware and parallel programming concepts needed to interpret the design choices and performance results presented later in the thesis. The focus is on the execution and memory organization of modern NVIDIA GPUs: the CPU–GPU offload model, kernel launch and thread hierarchy, the Streaming Multiprocessor (SM) as the unit of throughput and latency hiding, and the memory hierarchy that often dominates performance in PDE workloads. Although this thesis uses OpenACC, CUDA terminology is adopted as a common reference vocabulary, because directive-based offload ultimately maps to the same underlying grid–block–warp execution model and the same SM resource constraints.
The scope is deliberately *hardware-oriented* and *application-driven*. Rather than providing an exhaustive survey of GPU programming, the chapter highlights the mechanisms that most directly impact implicit finite-element workflows: SIMT execution and divergence, warp scheduling and latency hiding, occupancy limits driven by registers and shared memory, and memory-system effects (coalescing, caches, bandwidth, and atomics). These concepts are then connected to OpenACC's abstraction levels (gang/worker/vector) to clarify how the offloaded Alya-ADR kernels are shaped and why profiling metrics (issue rate, eligible warps, achieved bandwidth, and stall reasons) are meaningful for diagnosing and tuning performance.

## 2.1. GPGPU and Compute Unified Device Architecture

General-purpose GPU computing (GPGPU) is based on a simple, but powerful, division of labor: a latency-optimized CPU orchestrates the application, while a throughput-optimized GPU executes data-parallel kernels with massive concurrency. Although modern GPU platforms provide increasingly integrated software stacks, the underlying execution model remains conceptually *heterogeneous*: the host and the device expose different strengths, different memory hierarchies, and different scheduling mechanisms. Understanding the hardware organization is therefore essential to interpret both performance opportunities and bottlenecks in real PDE workloads. Figure 2.1 summarizes the core hardware components that recur throughout this the-

sis: the CPU host with its main memory, a DMA-capable connection to the GPU device memory, a global execution queue on the GPU side, an array of streaming multiprocessors (SMs, historically also called multiprocessors), and a memory hierarchy anchored by a device-wide L2 cache and off-chip global memory. Inside each SM, warps are scheduled and dispatched onto execution pipelines using a large register file, shared memory, and specialized units [14, 15].

### 2.1.1. CPU vs GPU execution model

A CPU core is designed to minimize the latency of sequential or lightly threaded execution. It typically features sophisticated control logic (branch prediction, out-of-order execution), large multi-level caches, and high single-thread performance. This design is well suited to the irregular control flow and system-level responsibilities of production simulation codes: mesh and input parsing, dynamic memory management, complex data-structure traversals, and the orchestration of multi-stage workflows.

A GPU, by contrast, is designed to maximize throughput by executing a very large number of threads concurrently. Instead of investing silicon area primarily in complex control logic per core, GPUs replicate simpler execution resources many times and rely on massive parallelism

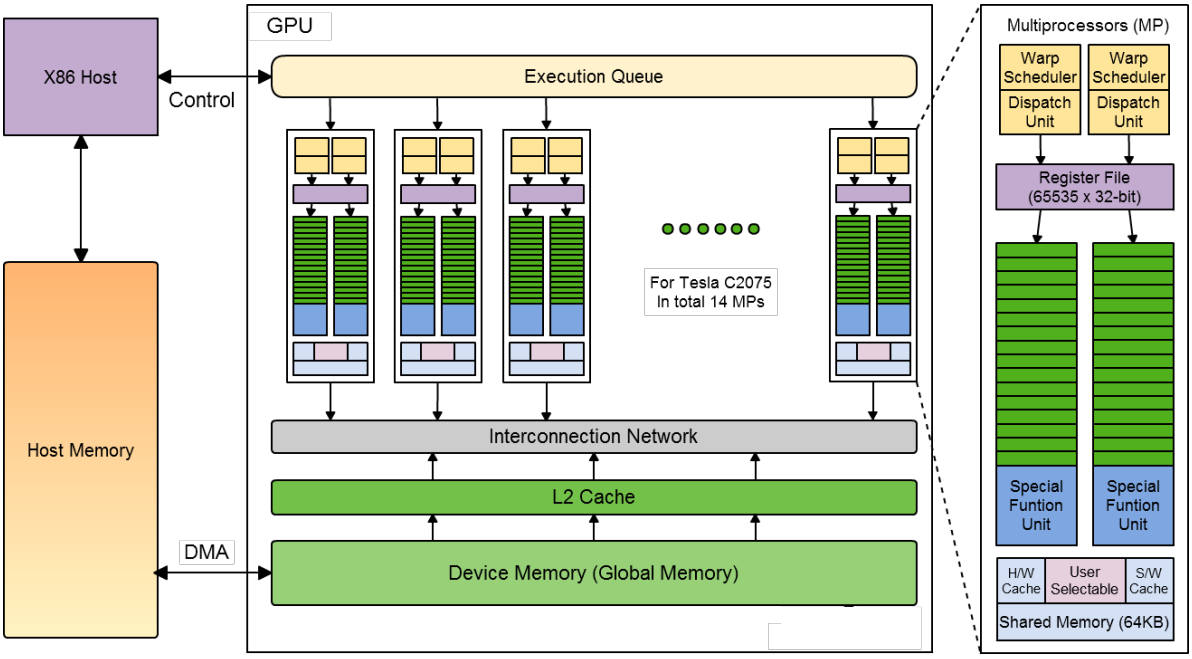


Figure 2.1 **GPU Hardware overview.** Hardware-oriented view of the CPU–GPU execution model and a representative NVIDIA SM. The host CPU issues work to the GPU through a launch/command interface; data is transferred between host memory and device global memory through DMA. The GPU schedules work from an execution queue across multiple SMs. Each SM contains warp schedulers and dispatch units, a register file, on-chip shared memory, and execution units (including special function units). A device-wide L2 cache and global memory back the computation. Taken from [21].

to hide memory and pipeline latencies. The performance model is therefore shaped by three hardware facts:

- **Concurrency is the first-class resource.** High utilization requires exposing enough independent work to keep many warps resident and schedulable across all SMs.
- **Memory behavior dominates for many HPC kernels.** Even when arithmetic intensity is moderate, sustained performance depends on effective use of caches, locality, and regular access patterns that can be served efficiently from the memory hierarchy.
- **Synchronization is expensive at scale.** Operations that serialize execution (e.g., contended atomics or frequent global barriers) can negate the benefit of increased parallelism.

The heterogeneous execution model is operationalized through an offload pipeline that is visible in Fig. 2.1. The CPU host remains responsible for the global program flow and issues GPU work to an *execution queue* on the device. Before the GPU can compute, the required data must be available in device address space. On discrete GPUs, host memory and device memory are physically separate, and data movement occurs through explicit transfers that are typically executed by DMA engines over a host–device interconnect (commonly PCIe). From an application perspective, these transfers may be synchronous or asynchronous, but from a hardware perspective they represent movement across a much higher-latency boundary than on-device loads and stores. For this reason, a consistent theme in GPU performance engineering is to minimize host–device traffic and to maximize the fraction of the workflow that operates on data resident in global memory and caches. Once data is resident, the GPU executes *kernels*, i.e., device functions launched by the host. Kernels are not executed by a single "GPU core". Instead, the GPU decomposes kernel work into independent thread blocks that can be distributed across the available SMs. Each SM then runs many threads concurrently using on-chip state. Fig. 2.1 highlights the most relevant SM-local resources:

- **Register file.** Registers are the fastest storage for thread-private variables. High register usage increases per-thread state and can reduce the number of concurrently resident warps.
- **Shared memory.** Shared memory is an explicitly managed on-chip scratchpad shared by threads in a block. It enables low-latency data reuse, but it is limited in capacity and competes with occupancy.
- **Warp schedulers and dispatch units.** The SM selects ready warps and issues their instructions to execution units. Latency hiding is achieved by switching between warps when some are stalled.
- **Special function units and execution pipelines.** These units handle arithmetic, load/s-

tore, and specialized operations. The practical performance limit often depends on how well the instruction mix and memory accesses can keep these pipelines busy.

From the perspective of implicit PDE solvers, this CPU–GPU split has an immediate interpretation. The CPU is well suited for irregular orchestration phases and for the parts of the workflow that do not parallelize deeply. The GPU is best used for the phases that can be expressed as large collections of independent operations: element-level assembly, vector updates, and sparse linear algebra kernels. However, not all "parallel" phases accelerate equally: if a kernel is dominated by indirect memory access (e.g., gathers through connectivity) or by synchronization (e.g., scatters with write conflicts), then the GPU's throughput advantage may be limited by the memory system and contention rather than by the raw compute capability. The remainder of this chapter builds the hardware vocabulary needed to reason about those effects.

### 2.1.2. CUDA programming model: kernel launch and thread hierarchy

CUDA provides a programming model that mirrors the GPU hardware organization while maintaining an abstract, scalable view of parallelism. Even when a code is written using directive-based offload (e.g., OpenACC), the generated device code ultimately follows the same hierarchy: the host launches work, the work is decomposed into blocks and warps, and blocks are scheduled across SMs. For this reason, CUDA terminology is used in this thesis as the reference vocabulary to describe how work is mapped onto the hardware. A kernel launch specifies a *grid* of thread blocks. Each block contains a fixed number of threads (organized conceptually as 1D/2D/3D), and each thread executes the same kernel code but with different thread indices. The crucial hardware property is that a thread block is the unit of scheduling onto SMs: blocks are assigned to SMs dynamically, depending on resource availability. Importantly, the order in which blocks execute is not defined, and blocks can run concurrently or sequentially. This execution freedom is what makes the CUDA model scalable: the same kernel launch can adapt to GPUs with different numbers of SMs, because blocks are required to be independent except through global memory.

This scheduling model directly matches the design goal of throughput machines: keep all SMs busy by providing a large pool of blocks to draw from. When many blocks are available, the GPU can spread the work across SMs, and when some blocks stall (e.g., due to memory), the scheduler can issue other blocks or switch between warps within a block. The practical implication for performance engineering is that *kernel configuration matters*: the number of blocks, the number of threads per block, and the per-thread resource usage together determine how many blocks and warps can be resident on each SM. Inside an SM, threads are grouped into *warps*, typically of 32 threads. The warp is the execution granularity for instruction issue: threads in a warp proceed in lockstep for a given instruction, and control-flow divergence causes the warp

to serialize different paths. Although a more detailed discussion of SIMT and warp scheduling is provided in Section 2.2, it is already useful here to connect the CUDA hierarchy to the hardware view in Fig. 2.1. A thread block corresponds to multiple warps that share the SM-local resources allocated to that block: a portion of the register file and a portion of the shared memory. These allocations constrain how many blocks can be simultaneously resident. For example, a kernel that uses many registers per thread may reduce the number of resident warps, lowering the GPU's ability to hide latency. Likewise, heavy shared-memory usage may limit block residency even when the kernel has abundant parallel work. CUDA also defines memory and synchronization scopes that reflect hardware constraints. Threads within a block can cooperate through shared memory and can synchronize using barriers; this is feasible because all threads of a block are co-resident on the same SM. In contrast, synchronization across blocks is not directly available within a single kernel launch (outside of specialized mechanisms), because blocks may run in any order and may not be simultaneously active. This constraint reinforces the importance of designing kernels as collections of block-local work units whose only global interaction is through memory operations that do not require strict ordering.

For PDE-oriented kernels, the grid–block–thread abstraction offers multiple natural mapping choices. Element-wise assembly, for instance, can assign one element (or one quadrature point) to a thread, or it can assign a small batch of elements to a block and use threads to process lanes within that batch. The appropriate choice depends on how data is laid out, how much reuse can be captured in shared memory, and how much contention is generated during scatter updates. In the workflow considered in this thesis, the implementation adopts a pack-based design (introduced in Chapter 1 and detailed later), which can be viewed through the CUDA lens as a controlled way of selecting the amount of fine-grain parallelism exposed to the GPU. In hardware terms, this ultimately influences block size, the number of resident warps, register/shared memory pressure, and the balance between memory stalls and useful instruction issue.

In summary, the CPU vs GPU execution model and the CUDA thread hierarchy provide the foundational hardware picture for the rest of the GPU chapter. They explain why GPU acceleration is not simply "more cores", but a different execution regime where performance depends on exposing parallelism at the right granularity and on respecting the constraints of the SM resource model and the device memory hierarchy shown in Fig. 2.1.

## 2.2. Multithreaded Multiprocessor Architecture and Streaming Multiprocessor

The fundamental compute building block of NVIDIA GPUs is the *Streaming Multiprocessor* (SM). From a hardware perspective, an SM can be interpreted as a highly multithreaded core

designed to keep its execution pipelines busy under long-latency events (primarily memory operations) by rapidly switching among many ready-to-run thread groups. This organization is the key reason why GPUs can sustain high throughput on data-parallel workloads: instead of relying on complex out-of-order logic to hide latency (as CPUs do), the SM relies on a large number of resident threads and hardware warp scheduling [7]. Figure 2.2 provides a hardware-oriented view of a modern SM implementation. Two aspects are especially important for the performance engineering perspective adopted in this thesis. First, the SM is *partitioned* into multiple processing blocks, each with its own warp scheduler, dispatch logic, and private slice of on-chip resources such as the register file. Second, the SM integrates a hierarchy of caches and on-chip memories that define the cost of data movement: instruction caches feed the front-end, while a shared L1 data cache / shared memory region supports low-latency access and intra-block cooperation. These hardware features motivate the vocabulary used throughout the rest of this chapter—warps, divergence, scheduling, occupancy—and they explain why seemingly minor code-level decisions (loop structure, temporary arrays, atomics, data layout) can have large consequences at scale.

### 2.2.1. SIMT execution: warps and control-flow effects

GPUs implement a *Single-Instruction Multiple-Threads* (SIMT) execution model. Threads are created by the programming model in large numbers, but the hardware schedules and issues instructions at the granularity of a *warp*, i.e., a fixed-size group of threads that share a program counter and execute in lockstep for a given instruction. In NVIDIA GPUs, the warp is the fundamental unit of scheduling and instruction issue: the warp scheduler selects a warp, and the dispatch logic issues an instruction from that warp to the appropriate execution pipelines. SIMT is often described as "SIMD-like", but the distinction is important for performance analysis. Each thread in a warp has its own register state and can access memory at different addresses. This makes SIMT convenient for irregular workloads (e.g., indirect gathers), but it also means that hardware efficiency depends on how uniform the warp's behavior remains during execution. Two forms of non-uniformity are particularly relevant.

If threads in the same warp take different branches of a conditional, the warp must *serialize* the execution of the different paths. Conceptually, the warp executes one path with a subset of threads active (the others masked off), then executes the other path. This reduces the effective instruction throughput for that region and increases the instruction count observed by the warp scheduler. Divergence is therefore most harmful when it occurs in inner loops or in frequently executed branches. In implicit PDE workloads, divergence often arises not from algorithmic branching but from mesh- and boundary-driven logic: boundary elements follow different paths than interior elements; different element types may trigger different operator terms; stabilization


Figure 2.2 **Streaming Multiproceprocessor architecture overview.** Illustrative NVIDIA Streaming Multiprocessor (SM) organization. The SM is partitioned into multiple processing blocks, each containing its own warp scheduler and dispatch unit, a slice of the register file, and execution resources (integer and floating-point pipelines, load/store units, special function units, and tensor cores). The front-end is fed by instruction caches, while the memory path includes on-chip L1 data cache/shared memory and a device-wide cache hierarchy (not shown here). This partitioned design is central to SIMT execution, latency hiding, and occupancy constraints. Taken from [21].

terms may be enabled or disabled; and sparse data structures can introduce conditional logic in scatter paths. Even when these branches appear small at source level, they can fragment warp execution and reduce the fraction of cycles spent issuing useful work. For this reason, later chapters interpret profiler stall and issue-rate metrics with an awareness that "low issue" can come from control-flow serialization as well as from memory latency.

Even without control divergence, a warp can suffer from *memory divergence*: threads execute the same load/store instruction but access addresses that are widely separated or poorly aligned. GPUs can combine nearby accesses into fewer memory transactions when addresses exhibit spatial locality (often described as coalescing). Indirect addressing patterns driven by unstructured connectivity tend to weaken this locality. The consequence is not only lower effective bandwidth, but also longer latency per instruction and increased pressure on caches and memory pipelines. For gather-heavy phases, this effect can dominate overall kernel time even when the arithmetic content is modest.

The hardware organization in Fig. 2.2 clarifies why these effects matter. Each processing block contains multiple execution pipelines, but those pipelines are only productive when the warp scheduler can continuously feed them with eligible instructions. Divergence increases the number of instructions required for a given amount of work, and irregular memory access increases the time each instruction spends waiting on the memory system. Both reduce the number of cycles in which the scheduler can issue work and therefore reduce achieved throughput.

### 2.2.2. SM scheduling and latency hiding

Unlike CPUs, GPUs generally do not attempt to hide long-latency events through complex out-of-order execution on a single thread. Instead, SMs hide latency by maintaining a large pool of resident warps and rapidly switching among them. This is the purpose of the warp scheduler and dispatch units highlighted in Fig. 2.2. When one warp stalls (e.g., waiting for a global memory load), the scheduler can issue instructions from another warp whose dependencies are resolved. If enough warps are ready often enough, the SM can sustain near-continuous instruction issue, and the latency of individual operations is amortized by concurrency.

This mechanism leads naturally to the profiler concepts of *active warps* and *eligible warps*, which will be used later when interpreting Nsight Compute results:

- **Active warps** are warps that are resident on the SM: their state is allocated (registers and other context), and they have not yet completed execution.
- **Eligible warps** are the subset of active warps that are ready to issue their next instruction. A warp can be active but not eligible if it is waiting on an unresolved dependency or a synchronization condition.

The practical reason this distinction matters is that high occupancy (many active warps) does not guarantee high throughput. If most active warps are waiting on the same type of dependency, the scheduler may frequently find that no warp is eligible, leaving issue slots unused. Typical causes include:

- **Memory dependencies:** warps waiting on results from global memory loads, cache misses, or heavily contended memory pipelines;
- **Execution dependencies:** long dependency chains in registers (e.g., reductions, recurrence-like computations), limiting instruction-level parallelism within a warp;
- **Synchronization:** barriers within a block, or serialization effects due to atomic operations and shared updates.

In a finite-element assembly context, these stall sources map to recognizable code patterns. Gather phases frequently generate memory dependencies due to indirect loads through connectivity arrays. Compute phases can generate execution dependencies if the computation is written as a deep chain of dependent operations that limits instruction-level parallelism. Scatter phases can introduce synchronization-like behavior when many threads contend on the same global locations through atomics, effectively serializing parts of the kernel. The role of scheduling is therefore not merely a hardware detail; it provides a causal link between algorithm structure (gather/compute/scatter) and the performance signatures later observed in profiler reports.
The partitioned SM design in Fig. 2.2 also clarifies how this scheduling scales. Each processing block has its own scheduler and dispatch logic, meaning instruction issue is distributed across multiple scheduling domains. In practice, this increases the aggregate issue capability of an SM, but it also means that performance can be limited by bottlenecks local to a partition: register pressure, limited eligible warps, or local pipeline contention can reduce issue efficiency even if other partitions are better utilized. This hardware reality is one reason why kernel-level profiling often reports metrics per scheduler or per SM sub-partition, and why interpreting those metrics requires a mental model of the SM's internal structure.

### 2.2.3. Occupancy and resource constraints

*Occupancy* is the term commonly used to describe how many warps (or threads) are resident on an SM compared to the architectural maximum. Occupancy is not a performance metric by itself; rather, it is a capacity metric that sets an upper bound on the number of warps available for latency hiding. Understanding occupancy is nevertheless essential because it is directly controlled by kernel configuration and resource usage.
The reason occupancy is constrained is visible in Fig. 2.2: each resident thread requires reg-

ister state, and each resident block requires a slice of shared memory and other bookkeeping resources. For a given kernel, the number of concurrently resident blocks and warps on an SM is limited by multiple constraints, including:

- **Registers per thread.** The register file is finite and is partitioned among resident warps. High register usage increases per-thread state and reduces how many threads can be resident at once.
- **Shared memory per block.** Shared memory (or the shared portion of the L1/shared region) is allocated per block. Heavy shared-memory usage reduces how many blocks can coexist on an SM.
- **Threads per block and block limits.** Even if registers and shared memory are sufficient, the architecture imposes maximums on resident blocks, resident warps, and resident threads per SM.

These constraints create practical trade-offs that appear repeatedly in GPU performance engineering:

- Increasing the amount of work per thread (e.g., by introducing larger local temporaries or unrolling computations) may reduce instruction count and increase arithmetic intensity, but it can also increase register pressure and reduce occupancy.
- Using shared memory to improve locality (e.g., caching element-local data) can reduce global memory traffic, but it may limit block residency and reduce the number of warps available for latency hiding.
- Choosing larger blocks can improve the ability to amortize overheads and to cooperate via shared memory, but it can also increase per-block resource demand, reducing the number of resident blocks.

A central message for the remainder of the thesis is therefore the following: *occupancy is necessary but not sufficient*. A kernel with very low occupancy often struggles to hide memory latency, but increasing occupancy does not automatically improve performance if the kernel is limited by other factors. For example, a kernel can achieve high occupancy yet remain slow if it is memory-bound with poor access locality, if it is dominated by atomic contention in scatter updates, or if instruction issue is limited by dependencies that prevent warps from becoming eligible. Conversely, kernels with moderate occupancy can still perform very well if they expose enough instruction-level parallelism and achieve high effective bandwidth.
For this reason, later chapters interpret occupancy together with other scheduler-centric metrics (active vs eligible warps, issue rate) and memory-centric metrics (achieved bandwidth, cache behavior, serialization indicators). The SM organization in Fig. 2.2 provides the hardware basis

for that interpretation: it shows where per-thread state lives (registers), where intra-block cooperation can occur (shared memory), and where the scheduler draws from the pool of warps to hide latency.

## 2.3. NVIDIA GPU Memories

Outside of the GPU cores themselves, the memory subsystem is often the main determinant of overall performance for HPC kernels. Data-parallel workloads generate extremely high bandwidth demand: large vector and matrix updates, neighborhood-style accesses, indirect gathers, and frequent load/store traffic for intermediate results all contribute heavily to memory pressure. Modern GPUs are therefore engineered primarily as *throughput* machines: the memory system is optimized to sustain very high streaming rates, even if the latency of individual accesses remains high. This design choice is central for interpreting profiler evidence in sparse and finite-element workloads: sustained performance depends less on the latency of a single load and more on the ability to keep many memory operations in flight and to convert them into efficient DRAM transactions.

### 2.3.1. HBM-backed DRAM organization and bandwidth scaling

Discrete NVIDIA GPUs pair the SM array with off-chip high-bandwidth memory (HBM). While the physical packaging differs from DDR, the fundamental DRAM constraints remain: DRAM is organized into banks, rows, and columns, and accesses are governed by timing constraints (e.g., row activation followed by high-rate column access while the row remains open). The key system-level difference is that HBM provides very wide interfaces and high aggregate bandwidth, allowing the GPU to sustain enormous streaming rates when access patterns are favorable.

To scale bandwidth, the GPU memory subsystem is organized into multiple memory partitions. Each partition includes an independent memory controller and a dedicated slice of the external memory system, and addresses are interleaved across partitions so that consecutive cache-line-sized regions are distributed across controllers. This fine-grain interleaving balances traffic and lets the device approach the ideal scaling where total bandwidth grows with the number of partitions. In practice, a high number of independent requesters (many SMs and many warps) continuously inject memory operations, creating a large pool of concurrent requests. Memory controllers exploit this concurrency to schedule requests efficiently (e.g., by favoring row-buffer locality when possible), which improves bus utilization but implies that latency is tolerated via concurrency rather than minimized.

### 2.3.2. Caches, streaming behavior, and locality

GPU cache hierarchies must sustain extremely high bandwidth, but they are small relative to typical working sets in simulation and assembly workloads. Rather than aiming for very high hit rates (as many CPU codes do), GPU caches support a streaming execution style in which hits and misses are interleaved while the machine continues to make progress using other warps. Device-wide caches (notably L2) reduce pressure on off-chip HBM when reuse exists, and they help smooth bursts of traffic generated by many SMs. However, for irregular access patterns (e.g., unstructured gathers through connectivity), reuse is limited and the effective performance limit often collapses to the achievable HBM bandwidth plus the efficiency with which requests can be formed and served.

Modern GPUs also support virtual memory, meaning that kernels generate virtual addresses that are translated to physical addresses via TLBs and page tables. For performance engineering, the key practical implication is that irregular memory access can increase TLB pressure and page-walk activity, adding overhead on top of already expensive off-chip loads. This is one reason why compact, contiguous data layouts and reuse-friendly ordering can matter even when arithmetic intensity is modest.

### 2.3.3. Memory spaces and synchronization scopes

The programming model exposes multiple memory spaces that correspond to different physical and logical organizations. Global memory resides in external HBM and provides a large, device-wide address space used for communication among thread blocks that may execute on different SMs at different times. Because blocks are scheduled dynamically, the model does not define a global execution order among blocks. Within a block, stronger ordering can be enforced via barrier synchronization; at device scope, atomics and explicit memory-ordering primitives provide coordination for shared updates.

Shared memory is allocated per thread block and is only visible to threads within that block. Its lifetime is tied to the block execution, so it is implemented on-chip and can provide very high bandwidth and low latency. Shared memory is therefore a primary mechanism to increase reuse and to reduce global-memory traffic, but it can suffer from bank conflicts when access patterns map many threads to the same bank, which serializes part of the traffic and reduces effective bandwidth.

Local memory is per-thread private storage that is logically addressable (unlike registers), but it is typically backed by global memory and only cached on-chip opportunistically. Excessive register pressure can trigger spills to local memory, which is especially harmful in bandwidth-bound kernels because it adds additional global traffic. Constant (and read-only) memory spaces

can be beneficial when many threads read the same locations, enabling broadcast-like behavior through specialized caches.

### 2.3.4. Coalescing: the critical rule of thumb

General-purpose load/store instructions are issued per thread, but the hardware services memory at warp granularity. To improve bandwidth utilization, the memory system coalesces per-thread requests from a warp into a small number of aligned memory transactions when the addresses are contiguous and properly aligned. A practical rule of thumb is therefore:

- Unit-stride, contiguous access is ideal: consecutive threads access consecutive addresses, producing few full transactions and high effective bandwidth.
- Indirect or strided access is costly: threads in a warp touch far-apart addresses (as in gather through a connectivity array), generating many transactions, lowering effective bandwidth, and increasing latency and cache pressure.

This coalescing behavior is one of the main reasons why "lane-first" layouts in pack-based assembly can matter: when the lane dimension is stored contiguously, threads mapped to lanes tend to access adjacent memory, improving transaction efficiency.

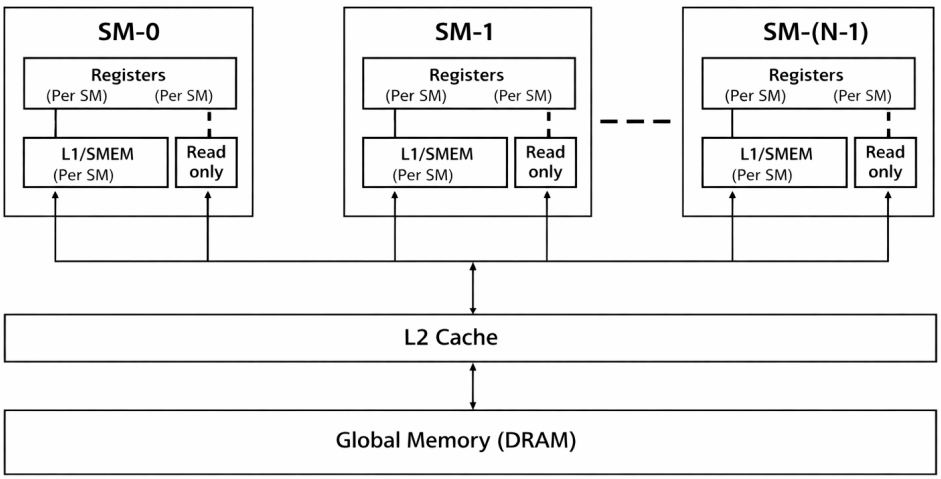


Figure 2.3 **GPU memory hierarchy and access granularity.** Black-and-white schematic of the GPU memory hierarchy, highlighting the per-SM private resources (registers, shared memory/L1, and read-only cache) and the device-wide levels (L2 cache and global DRAM). Low-latency, high-bandwidth accesses are confined within each SM, whereas inter-SM data traffic is mediated by the shared L2 and off-chip global memory. Warp-level memory requests are coalesced into aligned transactions when addresses are contiguous; irregular access patterns increase transaction count and limit effective bandwidth, a critical aspect in sparse and finite-element workloads. Adapted from [14].

### 2.3.5. Atomics and contention in scatter updates

In many PDE kernels, the gather phase is read-dominated and primarily limited by memory locality. The scatter phase is different: multiple threads may update the same global locations (shared nodes, shared rows, shared DOFs), which introduces write conflicts. When conflicts cannot be avoided, correctness requires atomic read–modify–write operations. Atomics serialize updates to the same address, and high contention can therefore turn a nominally parallel update into a partially serialized bottleneck. From a performance perspective, scatter contention is often visible as reduced achieved bandwidth, elevated memory/serialization stalls, and diminished scaling with additional parallelism. These effects are central when interpreting assembly profiling: improving coalescing in gather is necessary but not sufficient if scatter contention dominates the end-to-end time.
The memory principles above translate directly into actionable performance expectations for the gather/compute/scatter structure:

- Gathers are bandwidth/latency-limited: indirect connectivity-driven loads weaken coalescing and cache reuse, so effective bandwidth (not peak bandwidth) becomes the limiter.
- Compute is often secondary: unless the element operator is very expensive, assembly kernels tend to remain memory-oriented because each FLOP is paired with multiple loads/stores and address calculations.
- Scatters can be synchronization-limited: atomic updates needed for shared nodes/DOFs may dominate time when contention is high, especially for boundary contributions or dense coupling patterns.
- Data layout is a first-order knob: contiguous lane-first layouts improve coalescing and reduce transaction count, which directly improves achieved bandwidth.
- Batching helps when overhead dominates: larger packs reduce the number of launches and can improve steady-state streaming, but they do not remove the fundamental limits imposed by memory locality and atomic contention.

## 2.4. OpenACC: Survey on the library

OpenACC [20] is a directive-based programming standard for accelerator computing, designed to enable GPU offload without rewriting applications in low-level programming models such as CUDA. The model is incremental: the developer annotates existing loops and regions (pragmas in C/C++ or special comments in Fortran), and an OpenACC-capable compiler generates device

code and the required runtime calls. The same source can still compile and run on CPU-only systems because directives can be ignored when OpenACC is not enabled. This single-source, step-by-step porting path is a practical fit for large HPC code bases, where correctness must be preserved while progressively moving the dominant kernels to the GPU.
A typical usage pattern is to identify a compute-intensive loop nest, annotate it for parallel execution, and then control data movement explicitly so that arrays remain resident on the device across multiple kernels. For example:

```
#pragma acc parallel loop gang vector present(a,b,c)
for (int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
}
```

**Listing 2.1** OpenACC loop offload in C/C++.

```
!$acc parallel loop gang vector default(present)
do i = 1, n
  c(i) = a(i) + b(i)
end do
```

**Listing 2.2** OpenACC loop offload in Fortran.

### 2.4.1. Execution model and mapping to GPU parallelism

OpenACC exposes a hierarchical execution model with four levels: `gang`, `worker`, `vector`, and `seq` [19]. The intent is to provide a portable description of parallel structure while allowing the compiler/runtime to map it onto the hardware [16]. On NVIDIA GPUs, the mapping is implementation-dependent, but the following correspondence is a useful mental model for performance engineering:

- `gang`: coarse-grain parallelism. A gang typically maps to a CUDA thread block (CTA) scheduled on a single SM. Different gangs execute independently and may be scheduled in any order.
- `vector`: fine-grain, SIMT-style parallelism. Vector lanes typically map to CUDA threads, executed in warps. The vector length influences the number of threads used to execute the loop on the device.
- `worker`: intermediate granularity within a gang. On NVIDIA targets, worker often corresponds to a warp-like grouping or an internal subdivision used by the compiler to structure execution and memory usage. Because this mapping is not fixed across compilers, `worker` is best used when it reflects a clear nested-parallel structure in the code.

- `seq`: explicit sequential execution of a loop level on the device, used to keep outer loops parallel while executing inner loops in-register (or to preserve dependencies).

In practice, performance tuning typically starts by making the outermost independent loop level `gang`-parallel and mapping the innermost, unit-stride dimension to `vector`. This mirrors CUDA best practice: expose many thread blocks for occupancy (`gang`) and use enough threads per block to hide memory latency (`vector`). When the code contains nested parallelism (e.g., elements and quadrature points), `worker` can be used as an additional degree of freedom, but it should be introduced only when it reflects a stable structure and improves achieved occupancy or memory throughput on the target compiler. The key portability trade-off follows directly from this hierarchy. If the programmer leaves loop mapping unspecified, the compiler chooses a mapping that is generally reasonable but not necessarily optimal for a specific kernel. Adding explicit `gang`, `worker`, and `vector` clauses improves predictability and can unlock higher performance, but it also constrains the compiler and may reduce portability across devices and compilers.

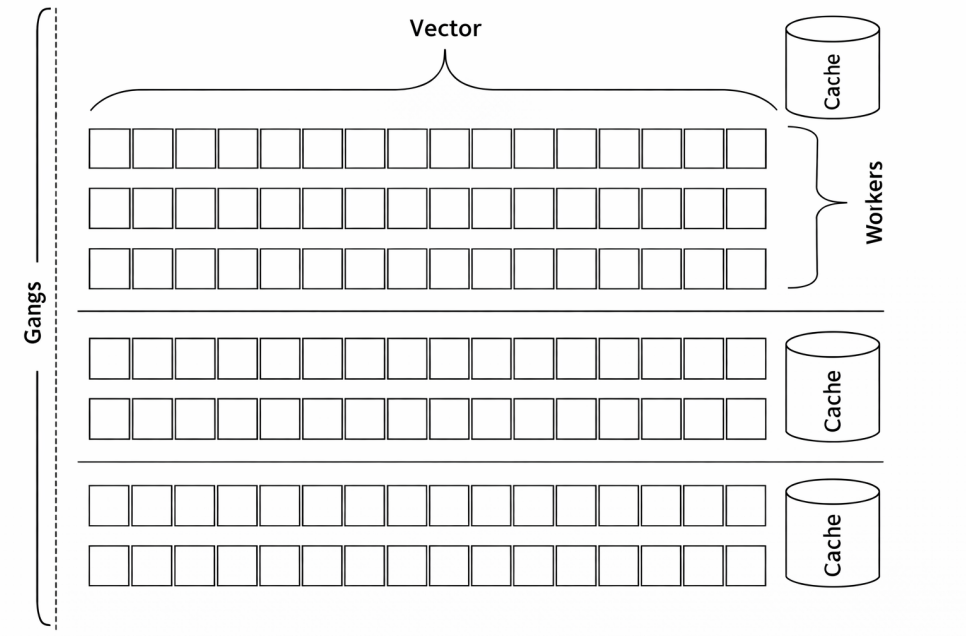


Figure 2.4 **Hierarchical gang–worker–vector execution model.** Scheme of a hierarchical parallel execution model in which computation is organized into *gangs*, each composed of multiple *workers*, which in turn execute vector lanes over contiguous data elements. The vector dimension exposes fine-grained data parallelism, typically mapped to SIMD lanes or warp threads, while workers coordinate a subset of the iteration space within a gang. Memory accesses are most efficient when vector lanes operate on contiguous addresses, enabling aligned and coalesced transactions; irregular access patterns reduce effective bandwidth and limit scalability in memory-bound kernels. Adapted from [19].

### 2.4.2. Data model, data movement, and correctness on accelerators

OpenACC separates compute offload from data residency. Compute directives (`parallel`, `kernels`, `loop`) control where loops execute, while data directives and clauses control where arrays live and when they are transferred. For GPU performance, the central goal is to avoid implicit host–device transfers by keeping long-lived arrays resident on the device and using `present`/`default(present)` to enforce that kernels operate on device-resident data. When intermediate results must be consumed on the host or when host-side initialization is required, explicit `update` operations provide controlled transfers.

For Fortran applications with derived types and pointer components, accelerator correctness requires extra care: copying a parent object to the device is typically shallow with respect to pointer members, so the device-side object may still contain host addresses. In such cases, the project uses explicit `enter data` for leaf buffers and `attach` to fix device-side pointers so that dereferences inside kernels refer to valid device allocations. This requirement is central for large OOP code bases, where mesh, boundary, and solver objects form multi-level pointer graphs.

Synchronization is relevant primarily when kernels perform updates to shared data. In assembly-style workloads, the gather phase is read-dominated, while the scatter phase may require coordination because multiple threads can update the same global locations. OpenACC provides atomic directives for correctness in these conflict patterns, at the cost of potential serialization under contention. The performance impact of atomics is kernel- and mesh-dependent and must be evaluated empirically; however, expressing atomic updates is often unavoidable for correctness in unstructured scatter operations.

### 2.4.3. Portability and performance trade-offs

OpenACC improves productivity and portability by delegating low-level scheduling, code generation, and device management to the compiler/runtime. This abstraction can limit the expressivity of device-specific optimizations that are straightforward in CUDA, and it can produce suboptimal code when the compiler cannot infer data lifetime, dependence structure, or an efficient mapping for a given loop nest. In practice, high performance with OpenACC typically requires making two aspects explicit: (i) the execution mapping of key loops (gang/worker/vector choices that match the kernel's structure), and (ii) the data lifetime and pointer validity (persistent device residency, `present` usage, and explicit `attach` for pointer-rich structures). These two controls preserve the portability benefits of directives while addressing the most common sources of performance loss: excessive data movement and weak kernel parallelization.

# 3 | Alya: Code Architecture and Workload Organization

Alya is a multi-physics engineering simulation code developed at the Barcelona Supercomputing Center (BSC) since 2004 [28]. Unlike legacy solvers that were originally conceived as sequential applications and later parallelized, Alya was designed from the beginning as a massively parallel code targeting large supercomputers and, more recently, heterogeneous CPU/GPU systems [5]. Its development has been driven by the requirements of engineering-grade simulations: complex geometries, unstructured meshes, strongly heterogeneous workloads, multi-physics couplings (often problem-specific), and the need to incorporate new models rapidly without sacrificing robustness and scalability (e.g., large Reynolds-number flows, reactive transport, multi-material solid mechanics, and bioengineering settings). Alya addresses the numerical solution of discretized partial differential equations arising in a broad range of engineering and scientific domains. Representative applications include incompressible and compressible turbulent flows, solid and structural mechanics (including non-linear regimes), heat transfer, species transport and chemistry, electrical propagation (e.g., electrophysiology), and particle-based transport models.

This thesis focuses on the ADR capability in Alya and on performance engineering of the implicit assembly workflow on GPU-enabled systems. For this reason, it is useful to summarize (i) how Alya is structured as a modular codebase and (ii) how it organizes and maps the computational workload across the hierarchy of modern HPC platforms. The first aspect clarifies where ADR integrates and which shared services it depends on. The second aspect clarifies why packing, data layout, and topology-aware mapping are first-order design constraints for GPU acceleration and scalable time-to-solution.

## 3.1. Alya overview

Alya primarily relies on variational discretizations, in particular finite element formulations, and it is well suited to unstructured meshes composed of multiple element types (e.g., tetrahedra, hexahedra, prisms, pyramids, with linear or higher-order variants). This flexibility enables

the representation of intricate geometries and mesh features typical of industrial and biomedical problems. For stabilization and accuracy in convection-dominated regimes, Alya adopts the Variational Multiscale Finite Element Method (VMS-FEM) as a core ingredient of several modules.

From a time-integration and algorithmic perspective, the code supports both explicit and implicit schemes, selecting the appropriate path depending on stiffness, coupling strength, and target accuracy. The codebase is also designed to accommodate both monolithic and staggered strategies; in practice, staggered schemes with coupling iterations are often preferred for large-scale multi-physics simulations due to flexibility and parallel efficiency.

Alya is widely used as a vehicle to demonstrate that realistic engineering workloads can scale to very high processor counts on general-purpose supercomputers. A performance study reports scalability up to 100,000 MPI processes on Blue Waters using three engineering-relevant test cases: incompressible flow in a human respiratory system, low-Mach combustion in a kiln furnace, and coupled electro-mechanics of the heart [28]. These cases are notable not merely for core counts, but for the combination of characteristics that tend to challenge scalability: complex geometries, unstructured meshes, explicit–implicit algorithmic mixes, strong coupling in multi-physics settings, and solver convergence considerations at scale. The same work emphasizes that Alya relies on in-house algebraic solvers tightly integrated with its parallel data structures, avoiding dependence on third-party solver frameworks when pursuing extreme scalability.

## 3.2. Software organization

Alya is engineered as a modular multiphysics platform rather than a monolithic solver, so that new physical models and numerical capabilities can be added without rewriting the execution backbone. The codebase is organized around three pillars: a kernel that orchestrates the simulation workflow, physics modules that implement specific PDEs and their element-level operators, and services that provide shared infrastructure (geometry, data structures, solvers, I/O, and runtime utilities) reused by all modules. This separation keeps the physics code focused on model formulation and element operators, while concentrating performance-critical infrastructure into a small number of shared libraries.

Figure 3.2 provides a build/dependency view of this modular design. In this thesis context, the red node identifies the application module developed/extended (Alya-ADR), while the green nodes indicate the core infrastructure targets that were adapted to support the workflow and performance changes, mainly in the solver stack and the core data/connectivity/memory services.

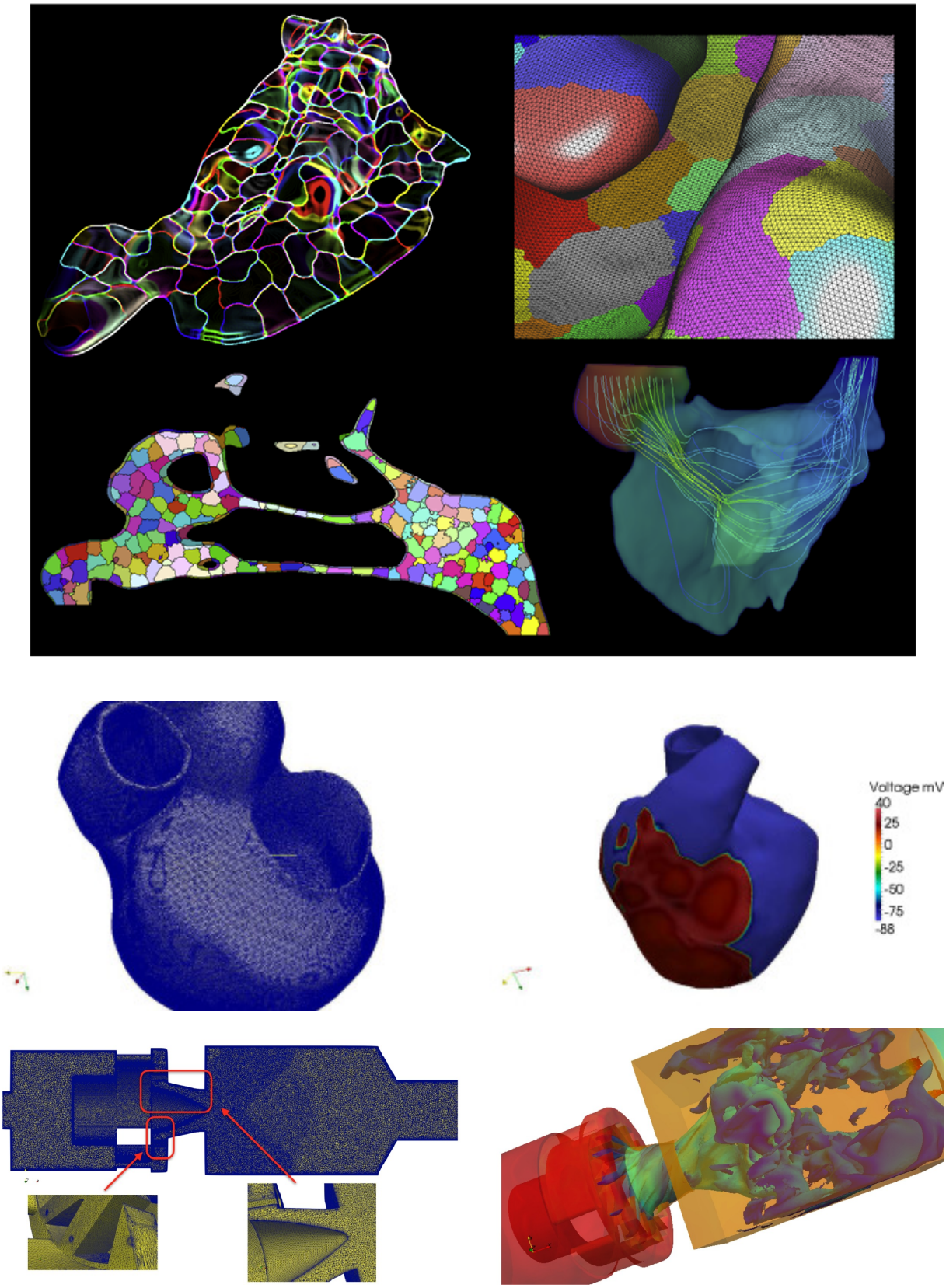


Figure 3.1 **Large-scale applications with Alya.** Examples of realistic multi-physics simulations on unstructured meshes: (top) nasal airflow benchmark with domain decomposition, and (middle/bottom) representative large-scale workloads in bio-mechanics (heart) and combustion. These cases illustrate Alya's ability to handle complex geometries and very large meshes while preserving scalability through parallel domain decomposition and execution.

### 3.2.1. Kernel

The kernel is responsible for global concerns that should not be duplicated across physics: reading and validating configuration, constructing the discrete problem, managing mesh and geometry metadata, driving the global workflow (including coupling iterations when present), triggering I/O, and invoking algebraic solvers. It provides well-defined integration points that modules implement or specialize, and it acts as the coordinator in coupled runs by enforcing call

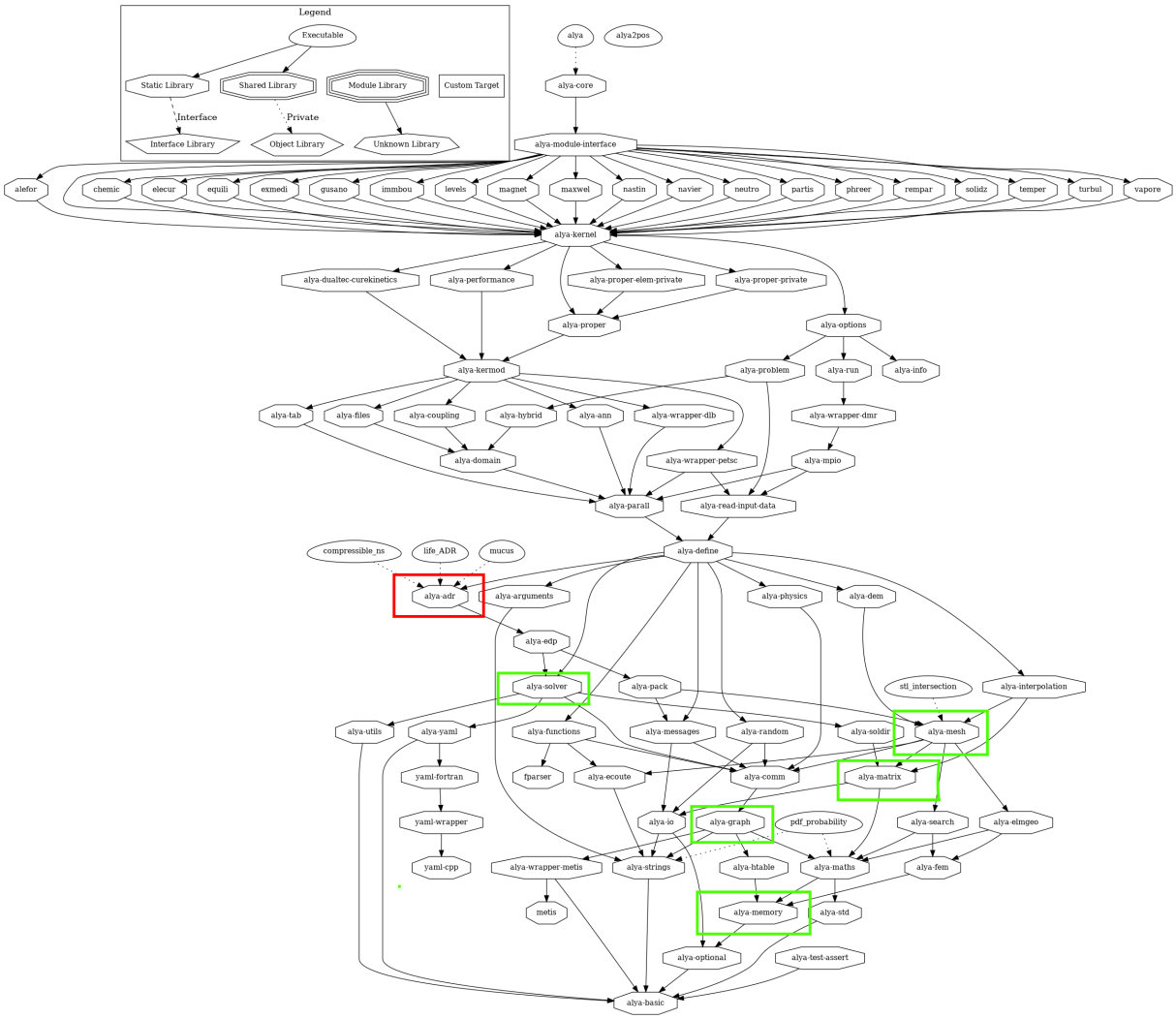


Figure 3.2 **Alya target-dependency graph.** Build/dependency overview of the Alya code stack, highlighting where this work integrates and what parts were adapted. The red node marks the `alya-adr` module (the application module developed/extended in this thesis) and its integration into the common Alya kernel and execution workflow. Green nodes indicate the core infrastructure targets that were modified to support the new workflow and performance-oriented changes, spanning the solver and the main data/graph services (mesh, matrix, graph/connectivity, and memory management). Overall, the figure emphasizes the modular design of Alya: physics modules plug into a shared backbone while a small set of central libraries concentrates most cross-cutting performance and functionality adaptations.

order, handling field exchanges, and applying convergence checks consistently across physics.

### 3.2.2. Physics modules

Physics modules encapsulate the model-specific discretization and assembly logic for a given PDE system (or a tightly coupled subset of PDEs). A typical module includes model parameters and material laws, element and boundary operators, and routines that assemble global algebraic forms. Modules can be activated *à la carte* depending on the application: a single-physics run enables one module and its dependencies, while multiphysics runs activate several modules and a coupling workflow coordinated by the kernel. This design lowers the cost of extension: adding a new model primarily means adding a new module that plugs into the existing kernel workflow and reuses the shared services.

### 3.2.3. Services

Services implement reusable capabilities that are not tied to a specific physics model. The central ones for most workflows are the mesh service (topology and geometry metadata), matrix service (sparse algebraic storage), solver service (iterative solvers and preconditioners), graph/-connectivity service (adjacency and connectivity structures), and memory service (allocation and memory management utilities). These services form the substrate for assembly and linear solves: element operators rely on mesh/connectivity to gather inputs; assembly targets sparse global data structures; solvers operate on those structures and require consistent handling of interface data. Consolidating these concerns into services improves maintainability and performance, because optimizations are implemented once in a shared component and automatically benefit all modules.

### 3.2.4. Where ADR fits in this thesis

Inside a physics module, the core computational pattern is the construction of discrete operators by looping over elements and boundary entities and assembling local contributions into global arrays. The standard workflow is to gather global fields and geometry into element-local storage, compute element matrices/vectors from the weak form (including stabilization, sources, and material terms), and scatter/accumulate the result into global sparse operators and right-hand-side vectors. Boundary conditions and interface terms follow the same logic through dedicated boundary assembly routines that evaluate fluxes, tractions, or constraints on boundary entities. This structure cleanly separates physics-specific computations (element operators) from infrastructure concerns (global indexing, sparse storage, and consistency of shared entities).

In this thesis, the ADR module is integrated into Alya's implicit pipeline and relies on shared services for (i) mesh and connectivity access, (ii) sparse matrix storage and insertion, and (iii) iterative linear solvers and preconditioning. The GPU porting work builds on the existing service interfaces: the objective is to accelerate the dominant assembly kernels and reduce data-motion overheads while preserving the same numerical workflow and solve interface.

## 3.3. Workload organization for HPC systems

Alya's performance on modern supercomputers is primarily driven by how it organizes the computational workload across nested parallelism layers and how it reshapes data to match the execution model of the target hardware. Rather than relying on a single form of parallelism, Alya adopts a hierarchical strategy that combines distributed-memory parallelism for inter-node scaling, shared-memory parallelism for intra-node concurrency, and fine-grain SIMD/SIMT execution for throughput-oriented kernels. This hierarchy is particularly visible in element and boundary assembly, which dominates runtime in many engineering workloads and is a primary target of CPU/GPU optimization in this thesis.

### 3.3.1. Distributed-memory decomposition: MPI subdomains and interface semantics

The top-level decomposition assigns disjoint mesh subdomains to MPI processes [4]. This is the fundamental scale-out mechanism: each rank owns a portion of the mesh and advances local unknowns concurrently with other ranks. In Alya, this partition is not merely a preprocessing artifact, but the backbone of runtime organization: local assembly, local sparse operator storage, and local solve kernels are all expressed in terms of the MPI subdomain.

The partition must satisfy two competing requirements: balance the computational work across ranks and minimize communication by reducing the size of subdomain interfaces. In finite element workloads, interface size directly impacts communication volume in halo exchanges and can strongly influence scalability when the solver phase becomes communication bound. To enable inter-rank consistency, Alya duplicates nodes located on interfaces between neighboring subdomains. These interface nodes appear in multiple subdomains, and their associated degrees of freedom require synchronization at specific points of the algorithm. This duplication is a deliberate design choice: it allows element assembly to proceed locally without remote memory accesses, while reducing communication to well-defined exchange operations with explicit schedules.

A key implication is that element and boundary assembly are largely communication-free once the subdomain is defined. Element contributions can be computed with local data, and interface

values required by local kernels are already present through node duplication. This pushes unavoidable communication to the linear algebra phase, where it is intrinsic to distributed sparse operators.

### 3.3.2. Intra-node parallelism on CPUs: shared memory and conflict management

On CPU nodes, each MPI subdomain is further decomposed to exploit shared-memory parallelism. Alya uses loop/task-based strategies to distribute work among threads. In assembly, the shared-memory strategy is particularly sensitive to write conflicts because multiple elements may contribute to the same global degrees of freedom. The practical objective is to maximize thread-level throughput while keeping synchronization overhead low, especially in the scatter/accumulate stage.

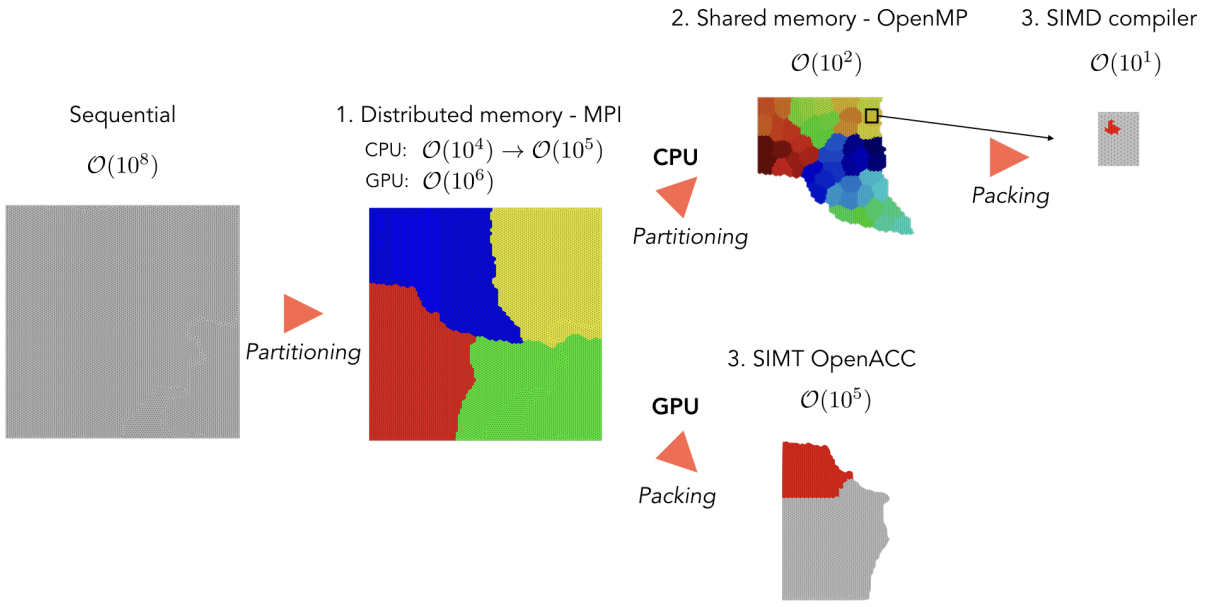


Figure 3.3 **Hierarchical workload management in Alya for CPU and GPU execution.** Overview of how Alya organizes and maps the computational workload across heterogeneous architectures. The global problem is first decomposed using distributed-memory parallelism, assigning mesh subdomains to MPI processes [9]. From this common partitioning, two execution routes are followed. On CPUs, each MPI subdomain is further subdivided to exploit shared-memory parallelism and then reorganized into compact data packs to enable efficient SIMD execution. On GPUs, the workload is directly aggregated into large packs that expose massive thread-level parallelism through SIMT execution. This architecture-aware decomposition allows Alya to adapt the granularity of the computation to the target hardware while preserving a unified programming concept (packs) and enabling heterogeneous execution within the same MPI environment. Adapted from [5].

This thread-level decomposition is designed to be compatible with the next level (SIMD/vectorization). Alya stacks concurrency layers rather than treating them independently: MPI provides coarse-grain parallelism across nodes; threading distributes work across cores; vectorization extracts throughput from the innermost loops.

### 3.3.3. Packing for regularity and SIMD efficiency

After MPI and shared-memory decomposition, Alya performs a data reorganization step—packing—to enable efficient SIMD execution. Packing groups elements into compact batches that share structural properties (same element type and integration rule). The purpose is to transform an irregular loop over heterogeneous elements into a regular loop over dense packs of homogeneous work items.

This pack-based layout has two immediate benefits on CPUs. First, it improves cache behavior by storing related element data contiguously, reducing latency and increasing effective bandwidth. Second, it exposes explicit vector lanes to the compiler: the pack dimension becomes a natural SIMD dimension, and the same arithmetic kernels are applied across multiple elements in a uniform manner. In practice, the pack size is tuned to match vector unit characteristics and kernel register pressure. The result is that the gather–compute–scatter pattern is executed over a small batch of elements at a time with predictable memory access patterns.

### 3.3.4. GPU execution path: packs as a SIMT mapping unit

On GPUs, Alya follows a different path after MPI partitioning. Rather than subdividing the MPI subdomain into thread-level subdomains, Alya aggregates work into large packs that expose massive thread-level parallelism for SIMT execution. The motivation is architectural: GPUs sustain throughput when there is enough parallel work to hide memory latency and keep many warps resident. For assembly, this translates into processing many elements concurrently. The pack abstraction provides a portability bridge across CPU and GPU: the pack dimension is a SIMD-friendly lane dimension on CPUs and becomes a thread-parallel SIMT dimension on GPUs. This unifies the kernel structure while allowing the pack size and its mapping parameters to be tuned to the device.

From a memory-system perspective, the GPU path requires explicit attention to device residency and data motion. Assembly involves many arrays and indirect accesses; relying on implicit transfers can dominate end-to-end time. Pack processing also enables a practical memory-capacity strategy: rather than requiring all data for the full MPI subdomain to be resident at once, Alya can process packs sequentially and keep only the working set on the GPU, staying within device memory limits while sustaining throughput [22].

The pack structure can enable pipelined implementations in which data staging and kernel execution overlap. In this thesis, packs are primarily exploited as the unit of work and as the unit of locality control (working-set containment and contiguous layout), while overlap opportunities are discussed in terms of what the abstraction enables and where runtime overheads can be reduced.

### 3.3.5. Where communication is unavoidable: solver kernels and synchronization

Alya's workload organization is also defined by when communication happens. Assembly is designed to be communication-free within a phase, but the solver introduces unavoidable communication because of distributed sparse linear algebra. Two patterns dominate: (i) point-to-point exchanges for halo/interface data needed by distributed sparse matrix–vector products (SpMV), and (ii) collective reductions for global dot-products and norms required by Krylov solvers.
SpMV requires that each rank has up-to-date values for interface nodes; this is implemented through explicit exchanges of the relevant vector entries. Dot-products require global reductions across ranks. Both operations imply synchronization, which means that load imbalance at the MPI level can translate into idle time at synchronization points. Consequently, partition quality and load balance directly influence scalability once the solve phase dominates.
Alya mitigates solver overheads by overlapping communication with computation where possible. In distributed SpMV, inner-node work can proceed while interface exchanges are in flight, reducing exposed communication time. This interaction between kernel speed, communication structure, and scheduling motivates the profiling focus adopted later: accelerating local kernels is necessary, but synchronization and communication can become the limiting factor at scale.

### 3.3.6. Assembly of `VECTOR_SIZE`

Throughout the thesis, `VECTOR_SIZE` denotes the fixed width of Alya's element "pack" (vector/batch) used by the assembly kernels. A pack is a list of element identifiers of length `VECTOR_SIZE`; the lane index `ivect=1..VECTOR_SIZE` selects the element handled by that lane. If the last pack of a category is not full, unused lanes are padded with a sentinel (typically `0`) and skipped by a validity check. The value of `VECTOR_SIZE` therefore controls the granularity of the gather/compute/scatter workflow: larger values imply fewer packs (hence fewer pack iterations and kernel invocations) while increasing the work performed per pack.
In this work, `VECTOR_SIZE` is a compile-time constant selected through the build system via preprocessor definitions. Values of `VECTOR_SIZE` are set through CMake/compiler flags and the code resolves the effective `VECTOR_SIZE` depending on whether OpenACC offload is en-

abled.
The same pack abstraction is used on CPUs and GPUs, but it maps to different architectural constraints:

- On CPUs, the lane loop is a SIMD-friendly dimension when arrays are stored with the lane index as the fastest-varying dimension. In this regime, `VECTOR_SIZE` influences cache footprint and register pressure: increasing it increases the amount of temporary data processed per iteration and may affect vectorization and locality.
- On GPUs, the lane loop is mapped to SIMT threads under OpenACC. In this regime, `VECTOR_SIZE` mainly acts as a batching parameter: it determines how many elements are processed per kernel launch (and thus how many launches are needed), trading off launch/runtime overheads against per-launch resource usage (registers, occupancy) and scatter contention.

When this thesis discusses a `VECTOR_SIZE` sweep, it refers to recompiling and running the same single-source assembly kernels with different pack widths to evaluate how this granularity parameter reshapes kernel execution and end-to-end assembly time.

### 3.3.7. Heterogeneous execution and multi-process GPU usage

Alya is designed so that CPU and GPU routes can coexist within the same MPI environment. Conceptually, the MPI subdomain remains the common unit of work: regardless of whether a subdomain is processed on a CPU or on a GPU, it remains a well-defined portion of the mesh with consistent interface semantics. This design enables heterogeneous execution strategies (including CPU–GPU co-execution in mixed allocations), but it shifts the main challenge to load balancing across devices: equal-sized subdomains do not imply equal execution time when device throughput differs.
At the node level, multiple MPI processes may target the same GPU if desired. In those cases, a multi-process GPU scheduling layer (e.g., CUDA MPS) can be used to manage concurrent GPU work streams from different ranks. In this thesis, the primary focus is on accelerating the ADR assembly kernels and controlling their locality and overheads; multi-process sharing and device-aware load balancing are relevant capabilities in the broader Alya execution model and become important when mapping many MPI ranks onto a limited number of accelerators.

# 4 | The Architecture and the Problem

This chapter defines the experimental and methodological foundation of the thesis. It first characterizes the target platform used for the GPU campaigns—MareNostrum 5 in its Accelerated Partition—documenting the ACC node configuration and its main components (CPUs, memory, GPUs, network interfaces, and local storage). It then introduces the reference implicit ADR setup used throughout the work and establishes a CPU-only baseline to quantify where time is spent (mesh/graph, assembly, solve) and to motivate the porting priorities. Finally, to make the remainder of the thesis self-contained, the chapter includes a short finite-element and assembly primer that defines the terminology used later (elements/DOFs, quadrature, sparse CSR operators, and the gather/compute/scatter pattern).
The scope is pragmatic and experiment-driven. The chapter focuses on (i) the MN5 ACC node as a concrete execution environment for directive-based offload, (ii) the baseline profiling workflow that serves as a reference for speedups and bottleneck attribution, and (iii) the key implementation constraints that shape the GPU strategy—pack-based assembly with `VECTOR_SIZE` and object-oriented Fortran data structures that require an explicit deep-copy/`attach` policy to make pointer trees valid on the device.

## 4.1. Description of the Architecture

This section documents the hardware platform used for the experimental evaluation in this thesis. The goal is purely descriptive: to provide a clear, reproducible reference of the compute-node configuration (CPU, memory, GPUs, network interfaces, and local storage) associated with the presented results.

### 4.1.1. MN5: system overview

MareNostrum 5 (MN5) is a EuroHPC system hosted at the Barcelona Supercomputing Center [3, 26]. The platform is organized as a multi-partition supercomputer, combining a CPU-oriented block and an accelerated block intended for heterogeneous execution. The user-facing system characterization distinguishes a General Purpose Partition (GPP), primarily CPU-based, and an Accelerated Partition (ACC), where nodes are equipped with multiple GPUs

The experiments in this thesis were executed on the ACC partition. Documentation reports that the ACC partition comprises 1,120 nodes and that each node provides four ConnectX-7 NDR200 network interfaces (200 Gb/s each, 800 Gb/s aggregate injection bandwidth per node)


Figure 4.1 **MareNostrum 5 ACC node topology.** Hardware-locality map (`hwloc/lstopo`) of a MareNostrum 5 Accelerated Partition (ACC) compute node used for the experimental runs. The node integrates two Intel Xeon Platinum 8460Y+ sockets (40 cores per socket, 80 cores total) and 512 GiB of DDR5 memory exposed as four NUMA domains (128 GiB each). Each core exposes private caches (L1i 32 KiB, L1d 48 KiB, L2 2 MiB), while each socket provides a shared L3 cache of 105 MiB. The PCIe/I/O layout includes four NVIDIA H100 GPUs (cuda0–cuda3, 64 GB HBM per device; 132 SMs and 40 MiB L2 per device as reported by the topology tool) and four ConnectX-7 NDR200 InfiniBand HCAs (mlx5_0–mlx5_3 mapped to ib0–ib3). A local NVMe device (`nvme0n1`, 480 GB) is available for node-local scratch. Taken from [3].

together with 480 GB of local NVMe storage.

### 4.1.2. Hardware description: ACC node

This subsection summarizes the hardware configuration of the ACC compute node used for the experiments. A representative topology view produced with `hwloc/lstopo` is shown in Fig. 4.1, which reports the cache hierarchy, NUMA domains, and the PCIe attachment of GPUs and network devices.

Each ACC node is built around two Intel Xeon Platinum 8460Y [10]+ processors. The measured `lscpu` output on the target node reports 40 cores per socket with SMT enabled (2 threads per core), for a total of 80 physical cores and 160 logical CPUs. The base frequency is 2.0 GHz, with a maximum turbo frequency up to 3.7 GHz. The shared last-level cache is 105 MiB per socket, consistent with the topology report.

MN5 documentation states that ACC nodes provide 512 GB of DDR5 main memory (16×32 GB DIMMs at 4800 MHz). In the topology report, memory is exposed as four NUMA domains. The figure also shows that PCIe devices are distributed across the two sockets.

Each ACC node integrates four NVIDIA Hopper H100 [11] GPUs with 64 GB of HBM per device. On the target node, `nvidia-smi` reports four GPUs (IDs 0–3), each with 65,247 MiB of on-device memory, with ECC enabled and MIG disabled. The GPUs are identified as cuda0–cuda3 in the topology view.

Each node is equipped with four ConnectX-7 NDR200 InfiniBand interfaces, visible as mlx5_0–mlx5_3 (mapped to ib0–ib3). A local NVMe device (`nvme0n1`, 480 GB) is available for node-local scratch space [23, 26].

## 4.2. Finite-Element Background and Terminology

This appendix provides the minimum finite-element (FEM) background and terminology needed to read the porting and performance chapters. The main text focuses on GPU offload and performance engineering; here we summarize the numerical objects that appear in the implementation (elements, degrees of freedom, quadrature, sparse matrices) and the standard assembly pipeline commonly referred to as gather/compute/scatter.

### 4.2.1. Mesh, elements, and degrees of freedom

A FEM discretization starts from a bounded domain $\Omega \subset \mathbb{R}^d$ and a mesh $\mathcal{T}_h$ that partitions $\Omega$ into a set of elements $e \in \mathcal{T}_h$ (e.g., triangles/quadrilaterals in 2D and tetrahedra/hexahedra/prisms in 3D). Each element has a small set of nodes, and the discrete unknown is represented in terms of

degrees of freedom (DOFs) associated with those nodes (for this thesis, a scalar DOF per node for the ADR field). The global vector of unknowns is denoted by $\mathbf{x} \in \mathbb{R}^n$, where $n$ is the total number of nodal DOFs.
Two properties are central for performance discussions: (i) element computations are local, involving only a small number of nodes per element, (ii) elements share nodes, so many local contributions update the same global DOFs.

### 4.2.2. From weak form to element-local contributions

In continuous Galerkin FEM, the PDE is expressed in a variational (weak) form and approximated in a finite-dimensional space. Operationally, this leads to a standard element-loop kernel structure:

- On each element $e$, evaluate geometric quantities (Jacobian, element volume, derivatives of basis functions in physical coordinates).
- Evaluate the coefficients and fields needed by the PDE operator (e.g., $\rho$, $\mathbf{a}$, $k$, $r$, $s$).
- Accumulate element-local contributions to an element matrix $\mathbf{K}^e$ and an element right-hand side $\mathbf{f}^e$.

The element contributions are typically computed by numerical quadrature. In code, this appears as a loop over Gauss points (`pgaus`) and loops over local nodes (`pnode`). This is the numerical origin of the Gauss-point loops in the ADR assembly kernels.

### 4.2.3. Assembly as gather/compute/scatter

Global operators are obtained by assembling the element contributions into global data structures. For a scalar problem, assembly produces a sparse linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{4.1}$$

where $\mathbf{A}$ is sparse because each element only couples the DOFs of its own nodes. In practice, assembly follows a common three-stage pattern:

- Gather: load the element nodal coordinates and (if needed) nodal fields from global arrays into element-local storage using the connectivity (local-to-global mapping).
- Compute: evaluate quadrature contributions and build $\mathbf{K}^e$ and $\mathbf{f}^e$.
- Scatter: accumulate $\mathbf{K}^e$ and $\mathbf{f}^e$ into the global matrix/vector at the corresponding global indices.

Algorithm 4.1 Generic FEM assembly (gather/compute/scatter)

**Require:** Mesh connectivity `lnods`, coordinates `coord`, field $\mathbf{x}$, CSR matrix $\mathbf{A}$, RHS $\mathbf{b}$

1: **for** $e \leftarrow 1$ to $n_{\text{elem}}$ **do**
2: Gather:
a. $\mathbf{g} \leftarrow \texttt{lnods}(:, e)$
b. $\mathbf{x}^e \leftarrow \mathbf{x}[\mathbf{g}]$
c. $\mathbf{X}^e \leftarrow \texttt{coord}[:, \mathbf{g}]$
3: Compute:
a. $(\mathbf{K}^e, \mathbf{f}^e) \leftarrow \textsc{ElementKernel}(\mathbf{X}^e, \mathbf{x}^e)$
4: Scatter:
a. $\mathbf{A} \leftarrow \mathbf{A} \oplus (\mathbf{g}, \mathbf{K}^e)$
b. $\mathbf{b}[\mathbf{g}] \leftarrow \mathbf{b}[\mathbf{g}] + \mathbf{f}^e$
5: **end for**

A compact pseudocode representation is:

This pattern explains why finite-element assembly is challenging on GPUs: gather and scatter use indirect addressing through `lnods`, and scatter may update the same global entries from different elements, creating write conflicts.

### 4.2.4. Sparse matrices and irregular structure

FEM global matrices are sparse because each row corresponds to a node (DOF) that interacts only with nodes connected through elements in the mesh. In Alya, sparse matrices are stored in compressed sparse row (CSR) format, defined by: (i) a row pointer array (`rowptr`), (ii) a column-index array (`colind`), and (iii) a value array (`values`). Assembly inserts element-local contributions into the appropriate CSR locations.

On structured Cartesian meshes, the sparsity pattern has a relatively regular structure. On unstructured meshes, connectivity is irregular and the set of neighbors per node varies across the domain. This irregularity is a key reason why memory access patterns become less predictable and why performance can be limited by indirect addressing and cache behavior.

### 4.2.5. Why scatter causes conflicts

Elements share nodes. As a consequence, two different elements may contribute to the same entry of $\mathbf{b}$ or to the same row of $\mathbf{A}$. In parallel assembly (threads on CPUs or threads on GPUs), these shared updates can occur concurrently.

Correctness requires that concurrent updates are serialized or otherwise made conflict-free. In practice, this is handled by: (i) atomic updates (simple and robust, but potentially expensive under high contention), or (ii) scheduling strategies that reduce simultaneous updates to the

same DOF (e.g., coloring/subdomain grouping).
This is the numerical origin of "contention" discussed in the GPU performance chapters: even if the compute part is highly parallel, scatter can become partially serialized when many threads target the same DOFs.

### 4.2.6. Connection to the porting strategy in this thesis

The porting strategy developed in Chapter 5 targets precisely the structure summarized above. The assembly kernels are organized as packs of elements, so that the compute phase is performed over a small batch with a regular lane dimension, while gather and scatter remain driven by connectivity. The parameter `VECTOR_SIZE` controls the pack width (the number of elements per batch) and therefore the granularity at which the gather/compute/scatter pattern is executed on the target device.

## 4.3. The Advection-Diffusion-Reaction Problem on Alya

The focus of this thesis is the implicit Advection–Diffusion–Reaction (ADR) capability in Alya, implemented as a scalar transport equation that captures, in a compact form, the key ingredients present in many engineering multiphysics modules: convection-driven transport, diffusion-driven smoothing, and reactive/source terms. Beyond its intrinsic relevance (e.g., species transport, heat transport, passive scalars, simplified chemistry), ADR is an ideal vehicle to validate the implicit pipeline in a dimension-agnostic way: mesh handling, element integration, stabilized variational discretization, sparse assembly, boundary-condition enforcement, and iterative solution of the resulting linear system.

### 4.3.1. General dimension-agnostic PDE statement

Let $\Omega \subset \mathbb{R}^d$ be a bounded domain with $d \in \{1, 2, 3\}$ (the formulation and implementation are identical for any $d$ supported by the mesh). The unknown is a scalar field $u(\mathbf{x})$. A generic steady ADR model in conservative form can be written as

$$\nabla \cdot (\rho(\mathbf{x})\, \mathbf{a}(\mathbf{x})\, u) \;-\; \nabla \cdot (k(\mathbf{x})\, \nabla u) \;+\; r(\mathbf{x})\, u \;=\; s(\mathbf{x}) \qquad \text{in } \Omega. \tag{4.2}$$

Here $\rho$ is a density-like coefficient, $\mathbf{a}$ is an advective velocity field, $k$ is a diffusion coefficient, $r$ is a reaction coefficient, and $s$ is a source term. This formulation is kept dimension-agnostic by driving all geometry- and operator-related loops with the runtime mesh dimension `ndime`.

Boundary conditions are posed on $\partial\Omega = \Gamma_D \cup \Gamma_N$ with $\Gamma_D \cap \Gamma_N = \emptyset$. In the unit tests used in

this work, the problem is posed with Dirichlet conditions:

$$u = u_D \qquad \text{on } \Gamma_D. \tag{4.3}$$

### 4.3.2. Stabilization (SUPG/ASGS) and the Codina $\tau$ parameter

To ensure robustness in convection-dominated regimes, the ADR operator is discretized with a stabilized continuous Galerkin formulation. In Alya, the unit test configuration enables a SUPG-family method (`method='SUPG'`) with the Codina definition of the stabilization parameter [6] (`tau='CODINA'`). The core idea is to enrich the standard test function with a streamline-aligned contribution, replacing the pure Galerkin test $N_i$ with a stabilized test function evaluated at Gauss points,

$$\phi_i \;=\; N_i \;+\; \rho\,\tau\,(\mathbf{a}\cdot\nabla N_i), \tag{4.4}$$

where $\rho$ is a density-like coefficient, $\mathbf{a}$ is the advection field, and $\tau$ controls the strength of the stabilization. In the ASGS variant, an additional reaction correction is included,

$$\phi_i^{\mathrm{ASGS}} \;=\; N_i \;+\; \rho\,\tau\,(\mathbf{a}\cdot\nabla N_i) \;-\; \tau\,r\,N_i, \tag{4.5}$$

with $r$ the reaction coefficient. In the implementation, this stabilized test function is built explicitly (`gptes`) and used to weight the Gauss-point residual and operator contributions during assembly.

In the Codina choice used in this work, $\tau$ is computed pointwise (per Gauss point) from local diffusion, advection and reaction scalings and from element length scales. Defining the advection magnitude

$$|\mathbf{a}| \;=\; \sqrt{\sum_{d=1}^{n_d} a_d^2}, \tag{4.6}$$

the code forms scaled terms (through configurable constants $p_1, p_2, p_3$)

$$k = p_1\,k_{\mathrm{diff}}, \qquad a = p_2\,\rho\,|\mathbf{a}|, \qquad r = p_3\,r_{\mathrm{rea}}, \tag{4.7}$$

and sets

$$\tau \;=\; \left(\frac{4k}{h_2^2} \;+\; \frac{2a}{h_1} \;+\; |r| \;+\; \varepsilon\right)^{-1}, \tag{4.8}$$

where $h_1$ and $h_2$ are element length scales provided by the code (used to weight advection- and diffusion-like contributions), and $\varepsilon$ is a small positive value preventing division by zero. This

definition blends diffusion, advection and reaction effects into a single stabilizing timescale: stabilization increases as diffusion weakens, advection strengthens, or the mesh is refined, and decreases in diffusion-dominated regimes.

### 4.3.3. From PDE to algebra: stabilized variational discretization and implicit form

Alya targets realistic unstructured meshes and convection-dominated regimes; therefore, the ADR operator is expressed in a variational form and stabilized when required. At a high level, a continuous Galerkin finite element discretization introduces a finite-dimensional trial space and seeks $u_h \in V_h$ such that, for all test functions $v_h \in V_h$,

$$(\rho\,\mathbf{a}\cdot\nabla u_h, v_h) \;+\; (k\,\nabla u_h, \nabla v_h) \;+\; (r\,u_h, v_h) \;=\; (s, v_h) \;+\; \text{(stabilization terms)}. \tag{4.9}$$

In practice, Alya adopts SUPG/VMS-style stabilization for robustness in convection-dominated cases. Conceptually, this modifies the test function by adding a term proportional to $\tau\,\rho\,\mathbf{a}\cdot\nabla v_h$ (and, depending on the chosen variant, additional reaction-related components), where $\tau$ is an element- and physics-dependent stabilization parameter. This choice is reflected directly in the element kernel: the code constructs a perturbed test function at Gauss points and uses it to weight both the residual contributions and the operator terms.

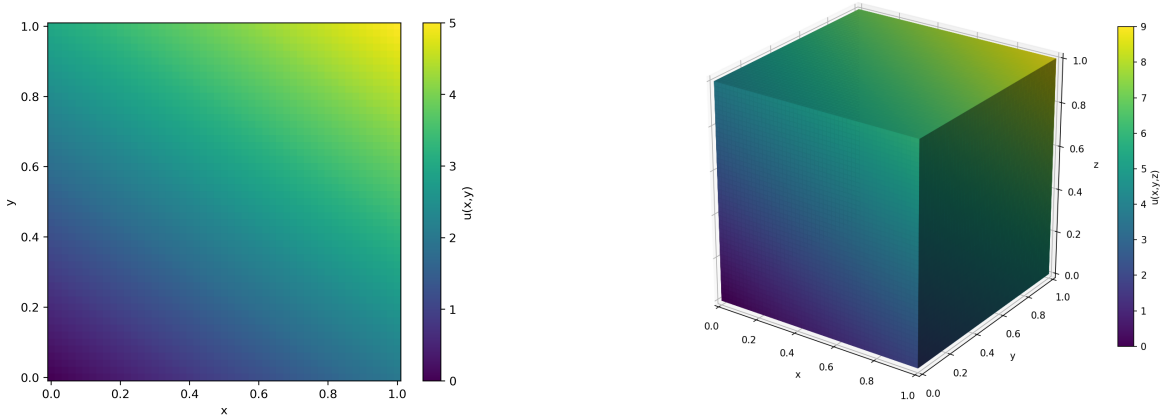


Figure 4.2 **Solution on Cartesian grids 2D and 3D.** Visualization of the manufactured scalar field used to validate the implicit ADR setup. Dirichlet boundary conditions are prescribed from the analytical solution (2D example: $u(x, y) = 2x + 3y$, and analogously in 3D as defined by the same manufactured-case setting in the unit test). The left panel shows $u(x, y)$ on $\Omega = [0, 1]^2$, while the right panel shows $u(x, y, z)$ on $\Omega = [0, 1]^3$ using a cube surface rendering. Resolutions match the unit-test configuration but are reduced for visualization; colorbars report the magnitude of $u$.

The stabilized discrete problem yields a sparse linear system of the form

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}, \tag{4.10}$$

where $\mathbf{x}$ is the vector of nodal unknowns, $\mathbf{A}$ contains the combined advection, diffusion, reaction, and stabilization contributions, and $\mathbf{b}$ aggregates the source term and boundary-condition contributions after constraint enforcement. Even when diffusion is present, the advection and stabilization contributions generally make $\mathbf{A}$ non-symmetric. From an algebraic standpoint, the inclusion of advection and SUPG/ASGS terms yields a sparse but generally non-symmetric global system, motivating the use of a Krylov method for non-symmetric operators, the GMRES solver [25].

### 4.3.4. Verification via manufactured solution and dimension consistency

To validate the full implicit ADR workflow (assembly + solve + boundary enforcement) in a controlled setting, a manufactured solution is employed. The idea is to prescribe an analytical $u(\mathbf{x})$, enforce it on the Dirichlet boundary, and construct the forcing so that the PDE is satisfied by design. This provides a strong end-to-end check: if the discretization, boundary conditions, and solver are consistent, the numerical solution converges to the manufactured field as the mesh is refined and the solver tolerance is tightened.

In the unit tests used in this work, the domain is a unit hypercube $\Omega = [0, 1]^d$ with a Cartesian grid, and the manufactured field is chosen as a simple linear function (2D example $u(x, y) = 2x + 3y$, and its consistent extension in 3D under the same manufactured-case setting). Dirichlet boundary conditions are applied by evaluating the analytical solution at boundary nodes. Figure 4.2 visualizes the resulting solution field in 2D and 3D for the unit-test configuration (with a reduced resolution for readability). The purpose of the figure is twofold: it confirms the expected smooth, linear behavior of the transported scalar, and it provides an immediate sanity check that the 2D and 3D pipelines are consistent (same logic, same parameters, different dimension).

From an implementation standpoint, this verification path is important because it stresses exactly the operations that are most error-prone in large codes: boundary tagging and extrapolation to nodal codes, consistent enforcement of fixed degrees of freedom, correct element geometry evaluation, and correct assembly into a global sparse matrix. A failure in any of these stages often manifests as boundary contamination, a globally shifted solution, or a solver that stagnates due to an inconsistent system.

### 4.3.5. What was "hard" in Alya: integrating ADR into the implicit pipeline

Although the ADR equation is mathematically compact, integrating it into Alya's implicit machinery involves several practical constraints imposed by a production-grade multiphysics code:

- **Consistent mesh-to-algebra mapping.** The solver stack operates on sparse matrices stored in CSR format and requires a stable graph (row pointers and column indices). The mesh connectivity must therefore be converted into a graph suitable for matrix assembly. In the unit tests, this is done by building the nodal adjacency graph from the mesh and allocating the CSR structure before assembly. The ADR kernel then only "fills" the matrix values by scattering element contributions into the precomputed sparsity pattern.
- **Boundary conditions through code-wide conventions.** Alya does not treat boundary conditions as a one-off feature in a single module; instead, it relies on common conventions: boundary tags $\rightarrow$ boundary flags $\rightarrow$ nodal codes $\rightarrow$ fixed-DOF arrays `fixn` and prescribed values `valn`. The ADR workflow must respect this pipeline so that the solver service can apply constraints consistently. In the unit tests, boundary tags are read from the mesh, mapped to boundary flags, extrapolated to nodal codes, and then used to mark fixed nodes and inject the manufactured values.
- **A stabilized formulation that remains implementation-friendly.** SUPG/VMS stabilization is expressed through Gauss-point quantities (shape functions, gradients, stabilization parameter $\tau$, and advective metrics). The implementation must compute these quantities efficiently and robustly for any element type and dimension, without hard-coding 2D/3D special cases. Alya's element integration service provides Jacobians, volumes, and Cartesian derivatives; the ADR kernel then builds the stabilized test functions and the final operator terms on top of those primitives.
- **A clean separation between "physics" and "algebra."** In Alya, the physics kernel should assemble local elemental matrices and right-hand sides, while the algebra service performs global insertion into the sparse structure. This avoids duplicating sparse-storage logic in each module and ensures solver compatibility. The ADR implementation therefore produces element-local arrays and relies on the matrix service for the actual assembly into CSR.

These constraints are exactly why a manufactured-case unit test is valuable: it validates that the ADR module is correctly plugged into Alya's shared implicit infrastructure, rather than merely producing plausible values in isolation.

### 4.3.6. Solver choice: why GMRES and why a diagonal preconditioner

The stabilized ADR discretization leads to a sparse linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ that is, in general, *non-symmetric* due to advection and SUPG stabilization contributions. This rules out Conjugate Gradient as a default (CG requires symmetry and positive definiteness). Alya provides several Krylov methods for non-symmetric operators (e.g., GMRES, BiCGStab); in this work the unit-test solver is configured as restarted GMRES with left preconditioning.

- *GMRES robustness.* GMRES builds a Krylov subspace and minimizes the residual over that subspace, making it a common default for non-symmetric advection–diffusion operators. In large problems, GMRES is typically used in restarted form GMRES($m$) to control memory usage and orthogonalization costs [25]. Restarting, however, can reduce robustness: unlike full GMRES (without restart), GMRES($m$) may stagnate or converge more slowly depending on the spectrum and on the restart length.

- *Left preconditioning.* With left preconditioning, GMRES is applied to

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}, \tag{4.11}$$

  where $\mathbf{M}$ is a preconditioning operator chosen so that applying $\mathbf{M}^{-1}$ is inexpensive and $\mathbf{M}^{-1}\mathbf{A}$ is better conditioned (informally, $\mathbf{M}$ should be easier to invert than $\mathbf{A}$ while capturing the main spectral features that slow down convergence) [25]. Typical choices in production codes include block-Jacobi, incomplete factorization variants (ILU), domain-decomposition preconditioners, or multilevel methods, depending on the problem and scalability constraints.

- *Diagonal (Jacobi) preconditioner as a baseline.* As a baseline, we use the diagonal (Jacobi) preconditioner, $\mathbf{M} = \mathrm{diag}(\mathbf{A})$. This choice has negligible setup cost and is universally available, but it often acts primarily as a row scaling and may provide limited improvement when poor conditioning is not addressable by diagonal rescaling. Despite its limitations, Jacobi is a practical reference in this thesis because it isolates the cost and behavior of assembly and the Krylov iteration without introducing a heavier preconditioner setup phase, and it is straightforward to support in an accelerator-oriented workflow.

In the unit-test configuration, GMRES uses a modest restart length (e.g., `nkryd=10`), a tolerance (e.g., `solco=1e-6`), and a large maximum iteration count to avoid premature termination during debugging and profiling. Dirichlet constraints are enabled so that fixed degrees of freedom are treated consistently by the solver service. This solver setup is not intended as an optimal choice for all ADR regimes; rather, it is a deliberately simple and reproducible baseline

that validates the discretization and supports performance attribution in the subsequent GPU porting analysis.

### 4.3.7. Element-level assembly: how the ADR operator becomes A and b

At the core of the ADR implementation lies the implicit element kernel, which follows the standard finite element pattern:

1. **Gather geometry and fields.** For each element, the kernel gathers nodal coordinates (and, when needed, nodal fields) into element-local arrays. This is the stage where the mapping from global mesh indexing to element-local indexing happens.
2. **Compute geometric operators at Gauss points.** Using Alya's integration service, the kernel computes Gauss-point volumes/Jacobians and Cartesian derivatives of shape functions. These quantities define gradients and Laplacian-related terms in a dimension-agnostic way.
3. **Evaluate coefficients and stabilization parameters.** The kernel evaluates material/-physics properties (density, diffusion, reaction, advection field, source term) at Gauss points. It then computes the stabilization parameter $\tau$ and the stabilized test function contributions.
4. **Build elemental matrix and RHS.** The element matrix includes (i) reaction-like and diffusion contributions, (ii) advective terms (proportional to $\rho\,\mathbf{a}\cdot\nabla u$ and the conservative contribution through $\nabla\cdot(\rho\mathbf{a}u)$), and (iii) stabilization corrections. The right-hand side aggregates the source term, again weighted by the stabilized test function when applicable.
5. **Scatter into global sparse structures.** Finally, the kernel hands the elemental matrix to the sparse-matrix service for insertion into the CSR structure and accumulates the elemental RHS into the global vector. This last step is where local physics meets global algebra, and it must be consistent with Alya's sparsity pattern and boundary-condition conventions.

A key point for this thesis is that the assembly is written in a dimension-agnostic manner: the same loops and algebra are driven by `ndime`, the number of Gauss points, and the element order/type descriptors. This mirrors the mathematical statement in Eq. (4.2): the PDE does not fundamentally change from 2D to 3D, only the underlying geometric operators do.

### 4.3.8. Putting it together: what the unit tests validate

The 2D and 3D unit tests implement a complete "mini-application" that mirrors Alya's production workflow in a controlled setting: (1) build a mesh, (2) initialize ADR data structures, (3) create and allocate the sparse matrix from the mesh graph, (4) define coefficients and stabilization, (5) enforce Dirichlet constraints from a manufactured solution, (6) assemble and solve with GMRES, and (7) verify that the numerical error is below a strict threshold. Figure 4.2 provides the qualitative check (expected field shape), while the error norms printed by the test provide the quantitative guarantee that the implicit ADR pipeline is correct in both 2D and 3D. This closes the conceptual loop needed for the rest of the thesis: the ADR problem is well-defined in a dimension-agnostic form, its discretization and stabilization are consistent with Alya's finite element machinery, the assembly produces the expected sparse algebraic system, and the chosen solver (GMRES with a simple diagonal preconditioner) provides a robust baseline for subsequent experimentation. In later sections, performance-oriented discussions can therefore focus on the computational kernels and data movement with confidence that the underlying mathematical and numerical pipeline is correct.

## 4.4. Overview on Profiling of CPU

Before introducing any GPU offload, a CPU-only baseline was established to (i) quantify where time is spent in an end-to-end implicit ADR run, (ii) identify the dominant algorithmic bottlenecks as the problem size increases, and (iii) explain why the relative weight of preprocessing, assembly, and solve differs substantially between the 2D and 3D configurations. The baseline measurements were collected using two compact Alya unit drivers, `unitt_adr_2d_openacc` and `unitt_adr_3d_openacc`, configured to solve a manufactured-solution problem on structured Cartesian meshes. Even though the source files contain OpenACC directives, the baseline results discussed in this section correspond to host execution, and they represent the reference point for the subsequent porting effort. Both drivers follow the same high-level workflow. A structured Cartesian mesh is generated with boundary tagging enabled, the graph of the discrete operator is built and stored in CSR format, the sparse matrix is allocated, boundary conditions are extrapolated to nodal codes, and finally the linear system is assembled and solved through a restarted GMRES method with diagonal preconditioning. The manufactured solution enforces Dirichlet values on the boundary (through `fixn` and `valn`), ensuring that the same physics and boundary treatment is applied in all runs. Assembly includes both the volume (element) contribution and the boundary contribution, while the solve phase measures the time spent inside the iterative solver once the matrix and right-hand side are ready. The experiments were designed to match the number of unknowns between 2D and 3D as closely as

2D

| Grid | $T_{\text{mesh}}$ [s] | $T_{\text{assembly}}$ [s] | $T_{\text{solve}}$ [s] | iters |
|---|---|---|---|---|
| 1M | 0.473 | 0.713 | 47.964 | 2053 |
| 2M | 0.920 | 1.057 | 146.712 | 3665 |
| 4M | 1.582 | 2.197 | 531.799 | 6676 |
| 8M | 2.840 | 4.054 | 1770.856 | 10000 |

3D

| Grid | $T_{\text{mesh}}$ [s] | $T_{\text{assembly}}$ [s] | $T_{\text{solve}}$ [s] | iters |
|---|---|---|---|---|
| 1M | 2.950 | 3.472 | 7.578 | 157 |
| 2M | 5.724 | 6.656 | 15.668 | 197 |
| 4M | 11.136 | 12.036 | 35.699 | 247 |
| 8M | 22.340 | 24.632 | 87.753 | 307 |

Table 4.1 **CPU baseline runtime summary (2D vs 3D).** CPU-only baseline for manufactured ADR runs with matched unknown counts. $T_{\text{mesh}}$ includes mesh generation plus CSR graph construction and matrix allocation; $T_{\text{assembly}}$ includes element and boundary assembly; $T_{\text{solve}}$ is GMRES time (tolerance $10^{-6}$). The iteration count (iters) reports the number of GMRES iterations to convergence.

possible, using the correspondence $1000^2 \approx 100^3$ as a reference. Four sizes were considered, approximately 1M, 2M, 4M, and 8M unknowns. This matching is important: it allows a direct comparison at similar memory footprint and similar algebraic system size, while still exposing the different geometric and spectral properties of the 2D and 3D discretizations. To guide the porting strategy, the total runtime was decomposed into three macro phases:

$$T_{\text{tot}} = T_{\text{mesh}} + T_{\text{assembly}} + T_{\text{solve}}. \tag{4.12}$$

Here, $T_{\text{mesh}}$ includes mesh generation plus CSR graph construction and matrix allocation; $T_{\text{assembly}}$ includes both element and boundary assembly of the sparse matrix and right-hand side; and $T_{\text{solve}}$ is the iterative solver time. The same decomposition is shown visually in Fig. 4.3 (left panel) as a normalized breakdown, together with the corresponding GMRES iteration counts (right panel). Absolute timings and iteration counts used to build Fig. 4.3 are reported in Table 4.1. The table makes explicit a key aspect that is not visible from normalized bars alone: in 2D, the solve time grows extremely quickly and dominates the end-to-end runtime already at 1M unknowns; in 3D, preprocessing and assembly remain a substantial fraction of the total time even at 8M unknowns. Figure 4.3 shows that the macro-phase balance is fundamentally different in 2D and 3D. In 2D, the solve phase dominates at all sizes, rising from about 97.6% of the runtime at 1M to about 99.6% at 8M. In contrast, in 3D the runtime remains much more balanced: at 1M unknowns the solve phase is about 54% of the total, and even at 8M it is about 65%, leaving the remaining 35% split between preprocessing and assembly. This divergence is not a constant-factor effect: it increases with problem size. At 1M unknowns the 2D solve time

is already about 6.3 times larger than the 3D solve time (47.96 s vs 7.58 s), and at 8M it becomes about 20 times larger (1770.86 s vs 87.75 s). At the same time, the opposite trend is observed for preprocessing and assembly: for a given number of unknowns, 3D spends significantly more time in mesh/graph construction and in assembly than 2D, reflecting higher per-unknown costs in these phases. The primary reason for the different macro-phase mix is the GMRES iteration count. The right panel of Fig. 4.3 show that the 2D iteration count is one to two orders of magnitude larger than the 3D iteration count at matched unknowns. At 1M unknowns, 2D requires 2053 iterations whereas 3D requires 157 iterations, a factor of about 13. At 8M unknowns, 2D reaches about $10^4$ iterations whereas 3D remains close to 300 iterations, a factor of more than 30. Because each GMRES iteration involves sparse matrix-vector products, vector updates, and inner products, the total cost of the solve phase scales approximately as

$$T_{\text{solve}} \approx N_{\text{it}} \cdot C_{\text{it}}(n), \tag{4.13}$$

where $N_{\text{it}}$ is the iteration count and $C_{\text{it}}(n)$ is the cost per iteration for an $n$-unknown sparse system. The baseline data indicates that both factors grow with problem size in 2D. The cost per iteration increases significantly as the system grows (due to larger sparse vectors and reduced cache effectiveness), and the iteration count also increases strongly with refinement. For 2D, the average time per iteration increases from roughly $2.34 \times 10^{-2}$ s at 1M to roughly $1.77 \times 10^{-1}$ s at 8M (about a 7.6x increase), while the iteration count increases from 2053 to 10000 (about a

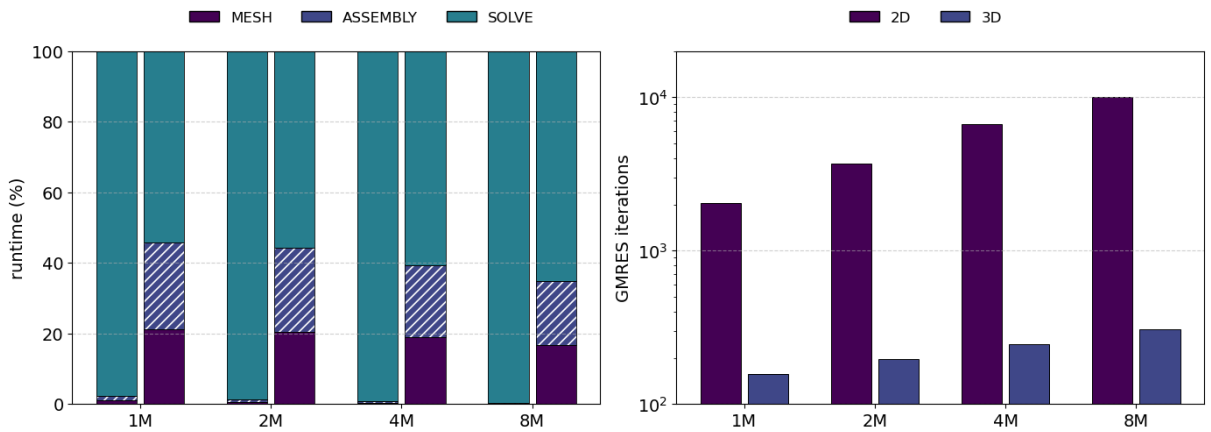


Figure 4.3 **Runtime breakdown and iteration count.** Comparison of 2D and 3D implicit ADR runs (manufactured solution) solved with GMRES at tolerance $10^{-6}$. **Left:** stacked runtime composition across three macro phases: MESH (mesh generation + CSR graph construction and matrix allocation), ASSEMBLY (element and boundary assembly of $A$ and RHS), and SOLVE (iterative solver time). **Right:** corresponding GMRES iteration counts (log scale). Problem sizes are matched to approximately 1M, 2M, 4M, and 8M unknowns (e.g., $1000^2 \approx 100^3$), highlighting that the larger solve-time share in 2D is primarily driven by a substantially higher iteration count.

4.9x increase). The product of these effects explains why the solve time grows by nearly two orders of magnitude and completely hides the preprocessing and assembly costs in the stacked breakdown. In 3D the situation is different. The cost per iteration also grows with size (the average time per iteration increases from roughly $4.83 \times 10^{-2}$ s at 1M to roughly $2.86 \times 10^{-1}$ s at 8M), but the iteration count remains comparatively small and grows moderately (157 to 307). As a result, $T_{\mathrm{solve}}$ increases by about 11.6x from 1M to 8M, which is closer to the expected growth driven by a larger sparse system, and it does not overwhelm the remaining phases. Even when the number of unknowns is matched, the 3D pipeline performs more work per unknown in preprocessing and assembly. This is visible in Table 4.3 at 1M unknowns, 3D spends 2.95 s in mesh/graph construction versus 0.47 s in 2D, and 3.47 s in assembly versus 0.71 s in 2D. The same trend persists at larger sizes. There are several cumulative reasons for this effect, all consistent with how the drivers build the system. First, 3D elements have more nodes and typically require more operations per element during both graph construction and local matrix evaluation. Second, the 3D sparsity pattern is denser: each row couples to more neighbors than in 2D, increasing both the amount of index work during CSR construction and the number of floating-point operations and memory references during assembly. Third, at matched unknowns the 3D grid uses a smaller linear resolution than the 2D grid (e.g., $100^3$ vs $1000^2$ for the 1M case). This increases the boundary-to-volume ratio in 3D at matched unknowns, and it makes boundary handling and associated indirect addressing comparatively more visible in the total time. These factors explain why, in the stacked breakdown of Fig. 4.3, the 3D mesh and assembly segments remain large enough to matter, while the 2D bars are almost entirely solve time. The macro-phase balance translates directly into a different instruction mix at the CPU level. In 2D, the runtime is dominated by the linear solver loop, which repeatedly executes bandwidth-bound sparse kernels (sparse matrix-vector products, vector updates, reductions) and spends a large fraction of cycles in memory accesses, address calculations, and reductions. In 3D, the smaller iteration count reduces the dominance of these sparse kernels, and a larger fraction of time is spent in preprocessing and in element and boundary assembly, which include more structured local computations, more geometry-related operations, and heavier index traversal during CSR construction. This difference is the key motivation for the porting strategy adopted in the next sections: the assembly stage is a natural GPU target in 3D because it remains a significant fraction of the time, while the 2D baseline highlights that end-to-end speedup can ultimately become solver-limited if the preconditioning strategy does not control iteration growth. The CPU baseline establishes two practical guidelines. First, in 3D the assembly cost is large in absolute terms and non-negligible in relative terms, so accelerating assembly can produce a visible reduction in end-to-end runtime, even without changing the solver. Second, the 2D baseline demonstrates that solver iterations can dominate at high resolutions, which limits the global speedup achievable by accelerating assembly alone. For this reason, the GPU port de-

scribed in the remainder of this chapter concentrates on maximizing assembly throughput and minimizing data motion, while keeping the solver configuration unchanged so that performance improvements can be attributed cleanly to the porting effort and its locality decisions.

# 5 | Porting and Performance Results

This chapter describes how the implicit ADR assembly workflow in Alya was ported to NVIDIA GPUs and optimized on the MareNostrum 5 ACC partition. The porting effort targets the dominant cost that remains highly visible in the 3D pipeline: finite-element element/boundary assembly, whose gather/compute/scatter structure stresses the GPU memory system and exposes correctness-sensitive write conflicts during scattering. Rather than rewriting kernels in a separate low-level implementation, the approach pursued in this thesis preserves a single source code for CPU and GPU execution using OpenACC, and concentrates optimization on (i) exposing the right granularity of parallelism through pack-based assembly, (ii) enforcing a GPU-friendly data layout to improve coalescing and locality, and (iii) making Alya's pointer-rich, object-oriented data structures valid on the device through an explicit deep-copy/attach strategy.
The chapter is organized around three complementary perspectives. First, it introduces the porting strategy at the algorithmic level, explaining how packs (controlled by `VECTOR_SIZE`/OpenACC `VECTOR_SIZE`) provide a unified portability bridge: a SIMD-friendly lane dimension on CPUs and a SIMT thread-parallel dimension on GPUs, while keeping correctness under scatter conflicts via atomics and subdomain grouping. Second, it addresses a practical but critical engineering aspect of GPU offload in large Fortran codes: object-oriented data ownership and pointer indirections, resolved here via a deterministic manual deep-copy procedure based on `enter data` and `attach`. Third, it quantifies the performance impact of these choices, focusing on assembly timings and speedups across 2D/3D manufactured cases and across a systematic `VECTOR_SIZE` sweep; qualitative profiler evidence is used to interpret why batching effects are dominant in 2D, while 3D assembly reaches a more bandwidth-oriented regime with smaller marginal gains.

## 5.1. Parallelization Strategy on GPU

This section describes the design adopted to offload the implicit ADR assembly to GPUs while preserving a single-source implementation. The strategy is centered on pack-based processing of homogeneous elements, where the same inner kernel is used for CPU and GPU execution and the mapping to the hardware is expressed through OpenACC directives and data layout choices.

### 5.1.1. Parallelization strategy for element and boundary assembly

In the implicit ADR workflow, a significant fraction of runtime is spent in assembly, where the global right-hand side vector and, in the implicit case, the sparse system matrix are constructed from element and boundary contributions. At the algorithmic level, both element and boundary assembly follow the same gather/compute/scatter structure. First, data is gathered from global mesh and field arrays into element-local storage. Second, element operators are evaluated at Gauss points and accumulated into element-level arrays (element matrix and element right-hand side). Third, the element arrays are scattered back to global data structures. This pattern is common in low-order finite element methods and is also the main source of performance difficulty: gathers and scatters use indirect addressing through connectivity, and scatters can introduce write conflicts because neighboring elements share nodes.

To address these issues while keeping a unified CPU/GPU implementation, Alya-ADR organizes assembly around packs (vectors/batches) of elements. Instead of assembling one element at a time, the code assembles a fixed number of elements simultaneously. The pack width is controlled by `VECTOR_SIZE`. Within a pack, the lane index `ivect=1..VECTOR_SIZE` identifies the element handled by that lane. The same lane loop is used as a portability dimension: it is SIMD-friendly on CPUs when the data is laid out with unit stride in `ivect`, and it maps naturally to GPU threads under OpenACC. The source remains single: when offload is enabled, OpenACC directives activate GPU execution, while the CPU path executes the same loops without requiring a separate kernel.

### 5.1.2. Element packing, categories, and padding

Packs are formed by grouping elements that are homogeneous with respect to the inner kernel requirements. In practice, this means that all elements in a pack share the same element type and the same quadrature rule, so loop bounds and data layouts are uniform across lanes. In the implementation, the element type `pelty`, the number of nodes `pnode`, and the number of Gauss points `pgaus` are taken from a representative element in the pack and passed as kernel parameters: `pelty = abs(mesh%ltype(ielem))`,
`pnode = element_type(pelty)%number_nodes`,
`pgaus = mesh%quad(pelty)%ngaus`.

Each pack stores a list of element identifiers `list_elements(:)` of length `VECTOR_SIZE`. If the number of elements in a category is not a multiple of `VECTOR_SIZE`, the final pack is padded using a sentinel identifier (typically `0`). Each lane then checks validity before executing work: `ielem = list_elements(ivect); if (ielem > 0) then ...`. This padding keeps the pack width fixed while preserving correctness and keeping the kernel control

flow uniform.

### 5.1.3. Two-level parallel structure: packs and lanes

Assembly exposes two complementary levels of parallelism.

At the outer level, the code iterates over packs. On CPUs, this loop is parallelized with OpenMP, distributing packs across CPU threads. Packs can also be organized in subdomains `isubd=1..pack%nsubd`. This additional grouping is used as a scheduling mechanism to reduce the probability of concurrent updates to the same global degrees of freedom during scatter. When conflict avoidance is effective, the amount of synchronization required for global updates is reduced; when conflicts remain, atomic updates are still required for correctness.
At the inner level, the lane loop `ivect=1..VECTOR_SIZE` provides fine-grained parallelism. On CPUs, this loop can be executed as a regular loop and may be auto-vectorized when data is arranged with `ivect` as the fastest-varying index. On GPUs, the same lane loop is mapped to threads using OpenACC, and `ivect` becomes the main thread-parallel dimension of the kernel.
The following schematic driver illustrates the pack-level control flow for the implicit case. The listing is intended to show the pack organization and parameter setup; the offloaded parallelism is expressed inside the pack kernel through OpenACC directives.

```
do isubd = 1, pack%nsubd
  do ipack = 1, pack%npack(isubd)

    list_elements => pack%list(isubd)%packs(ipack)%l

    ielem = list_elements(1)
    pelty = abs(mesh%ltype(ielem))
    pnode = element_type(pelty)%number_nodes
    pgaus = mesh%quad(pelty)%ngaus

    call element_assembly_implicit(ndime,pnode,pgaus,pelty, &
                                   list_elements, a, b, x)
  end do
end do
```

**Listing 5.1** Pack-level driver for implicit assembly (schematic). Packs are iterated in a host-side loop; the lane-parallel kernel uses the same source for CPU and GPU execution.

This organization preserves a single kernel implementation and concentrates architecture-specific

tuning into (i) the pack width VECTOR_SIZE, (ii) the lane-first data layout, and (iii) the OpenACC mapping of the lane loop.

### 5.1.4. Unified CPU/GPU kernel: lane-first data layout

The efficiency of pack-based assembly depends strongly on memory layout. Element-local arrays are allocated with an explicit lane dimension of length VECTOR_SIZE as the first (fastest) index. A simplified set of temporary arrays is:

```
real(rp) :: elrhs(VECTOR_SIZE, pnode)
real(rp) :: elmat(VECTOR_SIZE, pnode, pnode)
real(rp) :: gpvol(VECTOR_SIZE, pgaus)
real(rp) :: gpcar(VECTOR_SIZE, ndime, pnode, pgaus)
```

**Listing 5.2** Lane-first array layout for pack-based assembly. The first index is the lane, enabling unit-stride access across elements.

With this layout, the access pattern "same field, all lanes" corresponds to contiguous memory traversal. On CPUs, this improves cache-line utilization and supports SIMD vectorization over ivect. On GPUs, it improves coalescing because adjacent threads (lanes) tend to access adjacent addresses, reducing the number of memory transactions needed to serve a warp and increasing the achieved bandwidth in gather-dominated phases.

### 5.1.5. Gather, compute, scatter inside the pack kernel

The pack kernel performs gather/compute/scatter for VECTOR_SIZE elements. The compute phase is organized through Gauss-point and node loops, while the lane dimension is treated as the primary parallel dimension. When GPU offload is enabled, the lane loop is mapped to GPU threads with OpenACC. When offload is disabled, the directives are ignored and the same code runs as standard Fortran on the CPU.
A simplified element contribution illustrates the structure:

```
subroutine pack_kernel(pnode,pgaus,list_elements,elrhs,mesh,x)

  integer(ip), intent(in) :: pnode, pgaus
  integer(ip), intent(in) :: list_elements(VECTOR_SIZE)
  real(rp), intent(in) :: x(:)
  real(rp), intent(inout) :: elrhs(VECTOR_SIZE,pnode)

  integer(ip) :: ivect, ielem, igaus, inode
```

```
 !$acc parallel loop vector default(present) if(on_device)
 do ivect = 1, VECTOR_SIZE

  ielem = list_elements(ivect)
  if (ielem > 0) then

   ! gather (example)
   ! read coordinates, solution, and material data for this element

   ! compute (example)
   do igaus = 1, pgaus
    do inode = 1, pnode
     elrhs(ivect,inode) = elrhs(ivect,inode) + contribution(...)
    end do
   end do

  end if

 end do
 !$acc end parallel loop

end subroutine pack_kernel
```

**Listing 5.3** Unified CPU/GPU kernel skeleton for an element contribution. The lane loop is the thread-parallel dimension on GPUs and a vectorization-friendly dimension on CPUs.

In the full ADR operator, the gather stage prepares geometric terms and basis function data, the compute stage evaluates advection, diffusion, reaction, stabilization, and optional shock-capturing terms at Gauss points, and the scatter stage accumulates element contributions into the global right-hand side and sparse matrix.

### 5.1.6. Scatter and synchronization: atomics and subdomain grouping

Scatter is where correctness and performance interact most strongly. Since neighboring elements share nodes, different lanes and different packs can attempt to update the same global entry. When conflicts cannot be eliminated, global updates require atomic operations. Atomic overhead is low when conflicts are rare, but it can become dominant when many updates repeatedly target the same degrees of freedom.

For the right-hand side vector, the implementation scatters pack-local values by mapping the

lane loop to GPU threads and applying atomic increments:

```
!$acc parallel loop vector default(present) if(on_device)
do ivect = 1, VECTOR_SIZE
  ielem = list_elements(ivect)
  if (ielem > 0) then
    do inode = 1, pnode
      ipoin = mesh%lnods(inode, ielem)

      !$acc atomic update
      b(ipoin) = b(ipoin) + elrhs(ivect,inode)

    end do
  end if
end do
```

**Listing 5.4** Scatter from pack-local RHS to global vector. The lane loop is parallel; the update is atomic to preserve correctness under conflicts.

On CPUs, the same update can be performed either with atomics or with conflict-avoidance scheduling. The subdomain loop `isubd` provides a natural hook to reduce conflicts by grouping packs. Optional build-time strategies may further reduce conflicts, but when conflicts remain, atomic updates are required for correctness.
For the sparse matrix, scattering is delegated to a specialized assembly routine:
`call a%assembly(..., list_elements, ..., elmat)`. This routine encapsulates insertion of element matrices into the global sparse format. The same conflict considerations apply, while the internal handling depends on the matrix data structure and the chosen assembly strategy.

### 5.1.7. Choosing VECTOR_SIZE on CPUs and GPUs

The pack width `VECTOR_SIZE` is the central tuning parameter of the unified approach and acts as a granularity knob: it controls how much work is performed per pack invocation and how temporary storage scales inside the kernel.
On CPUs, smaller `VECTOR_SIZE` values tend to reduce the working-set size of lane-interleaved temporaries and increase the chance that intermediates remain in registers and caches. If `VECTOR_SIZE` is too large, temporary arrays grow and may increase cache pressure and register spilling.
On GPUs, larger `VECTOR_SIZE` values increase the amount of lane-parallel work per pack, providing more threads per kernel invocation and amortizing launch/runtime overheads. If

VECTOR_SIZE is too small, kernels may underutilize the GPU and expose latency; if too large, resource pressure (register use, occupancy reduction, and working-set growth) can offset the benefits. In this thesis, VECTOR_SIZE is treated as the primary tuning parameter and is swept in the performance study to quantify its impact on assembly throughput across 2D and 3D configurations.

### 5.1.8. Memory residency and pack processing

Pack-based assembly requires temporary element-local and Gauss-point arrays whose total size scales with VECTOR_SIZE. A practical approach is therefore to process packs sequentially so that only the temporaries for the current pack need to exist at a given time. This bounds the device memory footprint and keeps the method compatible with the physical limits of the GPU. At the same time, performance depends on controlling overheads associated with allocation and data movement. Two classes of data are relevant: (i) long-lived read-mostly data (mesh connectivity, coordinates, quadrature metadata, and invariant model parameters), and (ii) short-lived temporaries (element matrices/vectors and Gauss-point work arrays). The pack abstraction separates these concerns: long-lived data can be kept resident across multiple pack calls, while temporaries remain localized to the current pack. This makes it possible to tune memory residency independently of the physics operator implementation.
The final design exposes coarse-grain parallelism over packs and fine-grain parallelism over lanes. The lane dimension is the portability bridge: it is a unit-stride vectorization dimension on CPUs and a thread-parallel SIMT dimension on GPUs under OpenACC. Padding and category-based grouping keep pack processing uniform, while atomics and subdomain grouping provide correct handling of scatter conflicts. This organization preserves a single source code for CPU and GPU execution and concentrates performance tuning primarily into VECTOR_SIZE selection and device-residency policy.

## 5.2. Object-oriented data structures and OpenACC: manual deep copy with attach

Alya-ADR relies on derived-type hierarchies where high-level objects hold references to other objects, and where many components are stored through pointer members (e.g., coordinates, connectivities, element types, boundary data). This multi-level indirection is natural on CPUs, but it is not directly compatible with accelerator execution unless the complete pointer graph is made valid on the device. A common failure mode is an illegal-address error when device code dereferences a pointer member that still contains a host address.

The adopted strategy keeps a single source for CPU and GPU and explicitly manages the device-visible object graph in two phases. First, device allocations and mappings are established for leaf buffers (the actual data arrays) using OpenACC data directives. Second, pointer members in the parent objects are fixed on the device using `attach`, so that device-resident objects point to device-resident children. The `attach` directive does not move data; it updates the device-side value of a pointer member and therefore must be applied only after the pointed-to target is already present on the device. For nested objects, the same rule applies recursively: attachment proceeds from leaves to root [17][18].
Listing 5.5 shows the minimal pattern used to make a pointer-rich hierarchy valid on the GPU. Device copies are created for the relevant leaf arrays, then pointer members are attached in dependency order.

```
subroutine acc_attach_mesh(mesh)
  type(mesh_t), intent(inout) :: mesh
  if (.not. mesh%on_device) return

  !$acc enter data copyin(mesh)

  if (associated(mesh%coord)) then
    !$acc enter data copyin(mesh%coord)
    !$acc enter data attach(mesh%coord)
  end if

  if (associated(mesh%lnods)) then
    !$acc enter data copyin(mesh%lnods)
    !$acc enter data attach(mesh%lnods)
  end if

  if (associated(mesh%ltype)) then
    !$acc enter data copyin(mesh%ltype)
    !$acc enter data attach(mesh%ltype)
  end if

  if (associated(mesh%boundary)) then
    !$acc enter data copyin(mesh%boundary)
    !$acc enter data attach(mesh%boundary)
    call acc_attach_boundary(mesh%boundary, mesh%on_device)
  end if
end subroutine acc_attach_mesh
```

```
subroutine set_mesh(self, mesh)
  class(adr_t), intent(inout) :: self
  type(mesh_t), target, intent(inout) :: mesh

  self%mesh => mesh
  if (self%on_device) then
    mesh%on_device = .true.
    call acc_attach_mesh(mesh)
    !$acc enter data attach(self%mesh)
  end if
end subroutine set_mesh
```

**Listing 5.5** Minimal pattern for making a pointer-rich hierarchy valid on the GPU. Leaf arrays are made present on the device, then pointer members are attached from leaves to root.

This pattern enforces three practical rules:

1. Leaf arrays must be created or copied to the device before attachment. A pointer member can only be attached if a valid device mapping for its target already exists.
2. Attachment must follow the pointer dependency order.
   For a chain such as `self%mesh%boundary%lnods`, the ordering is: make `boundary%lnods` present and attach it; make `boundary` present and attach `mesh%boundary`; make `mesh` present and attach `self%mesh`.
3. Attachment is a correctness operation, not a data transfer. Data movement is performed by `copyin`, `update`, or explicit device allocation, while `attach` only fixes the device-side pointer value.

Alya-ADR applies this approach systematically to its production objects. During initialization, the solver stores pointers to the mesh and related structures and then executes a staged deep-copy/attach procedure for the mesh and boundary hierarchies. This explicit management provides a deterministic way to ensure that device kernels always dereference valid device addresses rather than stale host pointers.

## 5.3. Impact of `VECTOR_SIZE` on GPU assembly performance

This section quantifies how the OpenACC pack width (`VECTOR_SIZE`) affects GPU-accelerated finite-element assembly and explains why the sensitivity to `VECTOR_SIZE` differs between the 2D and 3D configurations shown in Fig. 5.1 and Tab. 5.1. The analysis combines two sources

2D (elements $= N_x^2$)

| Elems | $N_x$ | CPU asm [ms] | CPU solve [ms/it] | GPU asm [ms] for VECTOR_SIZE | | | | | GPU solve [ms/it] | Asm spd (best) | Solve spd |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 64k | 128k | 256k | 512k | 1024k | | CPU/GPU | CPU/GPU |
| 1M | 1000 | 593 | 23.5 | 24.3 | 17.3 | 13.8 | 12.9 | 11.5 | 0.727 | 51.6 | 32.3 |
| 2M | 1414 | 1170 | 53.5 | 33.0 | 28.9 | 24.6 | 20.4 | 18.2 | 1.08 | 64.3 | 49.5 |
| 4M | 2000 | 2410 | 113.0 | 60.6 | 52.7 | 42.0 | 35.7 | 31.5 | 1.93 | 76.5 | 58.5 |
| 8M | 2828 | 4660 | 231.0 | 114.0 | 89.7 | 69.0 | 64.3 | 57.3 | 3.47 | 81.3 | 66.6 |
| 16M | 4000 | 9450 | 471.0 | 221.0 | 184.0 | 141.0 | 119.0 | 107.0 | 5.98 | 88.3 | 78.8 |
| 32M | 5657 | 18300 | 946.0 | 431.0 | 347.0 | 250.0 | 220.0 | 208.0 | 10.9 | 88.0 | 86.8 |
| 64M | 8000 | 36900 | 1890.0 | 997.0 | 633.0 | 539.0 | 447.0 | 396.0 | 20.7 | 93.2 | 91.3 |

3D (elements $= N_x^3$)

| Elems | $N_x$ | CPU asm [ms] | CPU solve [ms/it] | GPU asm [ms] for VECTOR_SIZE | | | | | GPU solve [ms/it] | Asm spd (best) | Solve spd |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 64k | 128k | 256k | 512k | 1024k | | CPU/GPU | CPU/GPU |
| 1M | 100 | 3960 | 48.5 | 52.6 | 47.0 | 47.3 | 46.3 | 48.9 | 1.02 | 85.5 | 47.5 |
| 2M | 126 | 7960 | 102.0 | 97.6 | 88.4 | 87.9 | 86.5 | 89.9 | 1.64 | 92.0 | 62.2 |
| 4M | 159 | 16200 | 215.0 | 188.0 | 176.0 | 173.0 | 169.0 | 172.0 | 3.02 | 95.9 | 71.2 |
| 8M | 200 | 31300 | 425.0 | 368.0 | 349.0 | 340.0 | 337.0 | 335.0 | 5.64 | 93.4 | 75.4 |
| 16M | 252 | 65000 | 856.0 | 739.0 | 753.0 | 698.0 | 729.0 | 744.0 | 10.3 | 93.1 | 83.1 |
| 32M | 317 | 125000 | 1730.0 | 1390.0 | 1400.0 | 1490.0 | 1400.0 | 1450.0 | 19.8 | 89.9 | 87.4 |

Table 5.1 **CPU baseline vs GPU offload under a `VECTOR_SIZE` sweep (2D and 3D).** Assembly and solve timings for the manufactured ADR cases. CPU results correspond to `acc_off` (host-only baseline), while GPU results correspond to `acc_on`. The `VECTOR_SIZE` sweep (64k–1024k) affects assembly only; GPU solve is reported at `VECTOR_SIZE`=128k since it is essentially insensitive to the sweep. Solver timings are not time-to-convergence: both CPU and GPU solves are capped at a fixed maximum of 100 iterations, therefore solve cost is reported as time per iteration. "Asm spd (best)" is computed as CPU assembly time divided by the best (minimum) GPU assembly time across the sweep; "Solve spd" is computed as CPU time/iteration divided by GPU time/iteration. The "Elems" labels match the campaign mapping used in the plots (2D: $N_x^2$, 3D: $N_x^3$), with $4000^2 = 252^3 \approx 16.0$M elements.

of evidence: (i) end-to-end assembly timings reported by the internal performance counters (`assemble_ms`) and (ii) kernel-level measurements from Nsight Compute (NCU) collected within the assembly NVTX range [12, 13]. The NCU results are used to support statements about launch structure, dominant-kernel behavior, and bandwidth utilization; conclusions that depend on non-kernel overheads are stated conservatively, since NCU measures kernel execution only.

### 5.3.1. Pack-based assembly and performance model

Alya-ADR assembles element and boundary contributions using *packs* (vectors/batches): instead of processing one element at a time, it processes `VECTOR_SIZE` elements per pack. Therefore, `VECTOR_SIZE` primarily controls the *granularity* of the assembly offload: larger packs imply fewer pack iterations and fewer kernel invocations inside the assembly phase, while each kernel launch processes more work.

A convenient view of the end-to-end assembly timer is the qualitative decomposition

$$T_{\text{assemble}} \approx T_{\text{kernels}} + T_{\text{overhead}}, \tag{5.1}$$

where $T_{\text{kernels}}$ is the cumulative execution time of GPU kernels in the assembly phase and $T_{\text{overhead}}$ groups all non-kernel costs that can vary with granularity (kernel launch latency, OpenACC runtime bookkeeping, implicit synchronizations, and any host-side orchestration performed inside the timed region). NCU provides direct evidence for $T_{\text{kernels}}$ and kernel efficiency indicators (e.g., Speed-of-Light throughput metrics), while the internal `assemble_ms` includes both terms in (5.1). Consequently, changes in `assemble_ms` can be larger than changes in $T_{\text{kernels}}$ when $T_{\text{overhead}}$ is a non-negligible fraction of the assembly phase.
In the following, the shorthand notation 64k–1024k denotes $64 \cdot 1024$ to $1024 \cdot 1024$ elements per pack.

### 5.3.2. Observed trends and profiler-supported explanation

Figure 5.1 highlights two distinct regimes. In 2D, assembly speedup increases markedly with `VECTOR_SIZE` and the separation between curves grows with problem size. In 3D, speedup is high already at small pack sizes and varies weakly with `VECTOR_SIZE`, with mild non-monotonicity at the largest configurations. The main mechanism behind these trends is that `VECTOR_SIZE` changes the batching/launch structure, while the dominant kernels tend to preserve approximately constant total work.

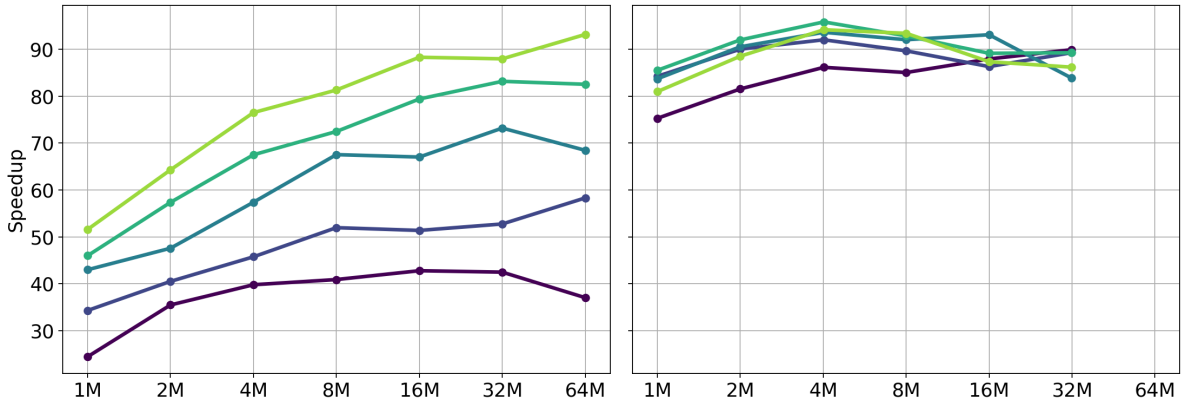


Figure 5.1 **Assembly speedup vs problem size under `VECTOR_SIZE` sweep.** Assembly speedup versus problem size for the 2D (left) and 3D (right) configurations under a `VECTOR_SIZE` sweep (64k–1024k). The x-axis reports the total number of elements (from 1M to 64M). Speedup is computed as the ratio between the CPU baseline assembly time and the corresponding GPU assembly time. In 2D, speedup increases strongly with `VECTOR_SIZE` and problem size. In 3D, speedup is already high at small `VECTOR_SIZE` and exhibits a weaker dependence on the sweep.

This behavior is directly visible in the NCU exports for the 2D case at $N = 4000$ (approximately 16M elements), profiled within the assembly NVTX range for `VECTOR_SIZE`$\in \{128\text{k}, 256\text{k}, 512\text{k}, 1024\text{k}\}$. The total number of kernel invocations decreases almost proportionally with pack size:

$$N_{\text{invoc}} = 3754 \;\rightarrow\; 1894 \;\rightarrow\; 964 \;\rightarrow\; 484,$$

which is the expected signature of `VECTOR_SIZE` acting as a granularity knob. However, the cumulative profiled kernel time decreases only modestly:

$$T_{\text{kernels}} = 96.85\ \text{ms} \;\rightarrow\; 92.58\ \text{ms} \;\rightarrow\; 90.65\ \text{ms} \;\rightarrow\; 89.14\ \text{ms},$$

corresponding to a maximum kernel-time improvement of about $1.09\times$ over this sweep. This indicates that, at large problem size, the dominant assembly work is already executed in a sustained regime where reducing launch count alone does not change the total amount of work performed by the hot kernels; rather, it repartitions that work into fewer, longer launches.

The kernel-level data also clarifies why end-to-end gains can still be substantial in 2D for the full sweep in Fig. 5.1 and Tab. 5.1. First, several secondary kernels benefit non-trivially from increased batching, showing speedups up to about $1.6\times$ over the same sweep (as observed in the per-kernel comparisons). Second, bandwidth utilization becomes more stable at larger packs: time-weighted NCU Speed-of-Light indicators increase significantly across the sweep (e.g., DRAM throughput rises from $54.3\%$ at 128k to $72.5\%$ at 1024k, and overall memory throughput from $60.4\%$ to $74.1\%$). These changes support the interpretation that larger packs reduce short-kernel effects and push more of the assembly work into a steady streaming regime. Since `assemble_ms` includes non-kernel overheads (runtime bookkeeping and launch effects) in addition to kernel execution, the end-to-end assembly time can improve more than $T_{\text{kernels}}$ alone when the assembly phase contains many relatively short kernel launches, as is typical in 2D at smaller pack sizes.

A representative dominant kernel illustrates the "granularity changes, total work stays" behavior. For example, `def_adr_element_assembly_implicit_2516` (dominant in the profiled range) exhibits nearly constant total time across vector sizes (about 20.5–20.9 ms), while its invocation count decreases from 125 (128k) to 16 (1024k) and its per-launch duration increases almost proportionally (from 0.166 ms to 1.282 ms). This is the expected scaling pattern when `VECTOR_SIZE` changes batching but the kernel remains limited by steady-state data movement and update costs: the device executes essentially the same total work, distributed across fewer launches.

The weaker sensitivity observed in 3D follows naturally. In 3D, the per-element assembly workload is heavier (larger local structures and more integration work), so kernels are longer already at moderate `VECTOR_SIZE`. In that regime, granularity-dependent overheads are more effec-

tively amortized, and changing VECTOR_SIZE mainly reshapes the launch structure without materially reducing the dominant-kernel work. Mild non-monotonicity at very large pack sizes is consistent with second-order effects (changes in register pressure, occupancy, cache working set, and scatter/update dynamics) that can offset small batching gains once the dominant kernels operate in a sustained regime.
Finally, the linear-solver time is nearly insensitive to VECTOR_SIZE in both 2D and 3D, which is expected: the pack granularity affects how the discrete operator is assembled, but once the sparse system and right-hand side are formed, solver cost is governed primarily by the solver/preconditioner configuration and the resulting matrix properties rather than by the assembly batching.

### 5.3.3. Practical implications and limitations

The combined evidence supports the following tuning guidance. In 2D-like regimes, increasing VECTOR_SIZE is a high-impact lever because it reduces the number of pack iterations and stabilizes bandwidth utilization across many short kernel launches, leading to large end-to-end assembly improvements and strongly increasing speedups with problem size. In 3D-like regimes, the assembly reaches a sustained-kernel regime at smaller pack sizes, so VECTOR_SIZE tuning yields diminishing returns and can be mildly non-monotonic at the largest values.
Two limitations apply to the profiler-based interpretation. First, NCU measures kernel execution and cannot directly attribute changes in assemble_ms to non-kernel overheads; therefore, conclusions about runtime/launch contributions are stated as consistent with, rather than proven by, kernel-only metrics. Second, while throughput indicators (e.g., DRAM and memory Speed-of-Light) provide robust qualitative evidence of streaming vs short-kernel behavior, they do not by themselves identify which specific mechanism dominates (coalescing, cache reuse, atomics, or runtime scheduling). For that reason, the section uses NCU primarily to substantiate the central, defensible claim: VECTOR_SIZE acts mainly as a batching knob that strongly reduces kernel invocations, while global speedup ultimately saturates when the dominant assembly kernels already execute in a steady-state regime and their total work is approximately invariant to further batching.

## 5.4. Overall GPU gain for the manufactured ADR workflow

This final section summarizes the end-to-end benefit of GPU offload for the manufactured ADR workflow, using the two most representative artifacts of the performance campaign: (i) the global speedup of the compute-dominant phases (assembly + solve) as a function of problem size and VECTOR_SIZE (Fig. 5.1), and (ii) the corresponding timing table that separates as-

sembly and solve costs (Table 5.1).
It is important to clarify what is (and is not) included in this analysis. The reported speedups refer to the phases that dominate the implicit ADR workload once the data structures are available, namely element/boundary assembly and the Krylov solve. Mesh/graph construction is excluded: in the current unit-test setup those stages are host-oriented and, in preliminary GPU experiments, the mesh-related path exhibits poor scaling due to irregular updates (atomic-heavy patterns), making it an unrepresentative component for the main thesis message. For this reason, the comparison is deliberately restricted to assembly + solve, where the porting work and the tuning parameter `VECTOR_SIZE` apply directly.

### 5.4.1. Speedup trends: 2D versus 3D

Figure 5.1 shows two distinct regimes. In 2D, the global speedup increases strongly with both problem size and `VECTOR_SIZE`. For the smallest case (about 1M elements), the overall speedup ranges from roughly $25\times$ at 64k up to about $50\times$ at 1024k; for the largest case (about 64M elements) it grows to the $40\times$–$90\times$ range, with the best configuration near $93\times$. This behavior mirrors the assembly timing reductions observed in Table 5.1: as the 2D problem grows, assembly remains sufficiently "short-kernel" and overhead-sensitive that larger packs reduce pack-iteration overheads and push the GPU execution toward a steadier regime. In 3D, the global speedup is already high at small sizes and varies only weakly across `VECTOR_SIZE`. Across the sweep, speedups remain mostly in the $80\times$–$95\times$ band, with mild non-monotonic changes at the largest sizes. This is consistent with the interpretation developed earlier: 3D element integration is heavier, so assembly kernels reach a bandwidth-oriented, steady-state regime at moderate pack sizes; further increasing the pack width mostly reshapes granularity (fewer launches, more work per launch) without reducing the total work, and second-order effects (register pressure, cache working set, scatter contention) can cancel or slightly offset batching benefits.

### 5.4.2. Assembly vs solve: time-per-iteration interpretation

Table 5.1 clarifies how assembly and solve contribute to the observed speedups. The assembly speedup ("Asm spd (best)") is computed as CPU assembly time divided by the best GPU assembly time across the `VECTOR_SIZE` sweep. In 2D, this best-case assembly speedup grows from about $52\times$ (1M elements) to about $93\times$ (64M elements). In 3D, it is already about $85\times$–$96\times$ at small and mid sizes, then slightly decreases toward $\sim 90\times$ at the largest size, again reflecting saturation and mild sensitivity to resource effects.
The solve columns require an explicit methodological remark: the solve is not reported as time-

to-convergence. Both CPU and GPU runs are capped at a fixed maximum of 100 iterations in these campaigns, so the meaningful quantity is solve time per iteration. With that interpretation, the solve speedup (CPU ms/it divided by GPU ms/it) increases with problem size and becomes comparable to assembly speedup at the largest cases (e.g., in 2D it reaches $\sim 91\times$ at 64M elements; in 3D it reaches $\sim 87\times$ at 32M elements). This confirms two practical points for the thesis narrative: (i) the Krylov iteration cost is significant and must be accounted for when discussing end-to-end benefit, and (ii) in this setup the solve phase is essentially independent of `VECTOR_SIZE`, since the pack width only controls the assembly batching.

### 5.4.3. Role of `VECTOR_SIZE`: why it changes only assembly

Across both 2D and 3D, the curves in Fig. 5.1 separate because `VECTOR_SIZE` changes the granularity of pack-based assembly. Larger packs reduce the number of packs required to cover a fixed number of elements, which reduces launch/runtime overheads and can improve steady-state bandwidth utilization when kernels are short (as in 2D). Once the linear system is assembled, the solver operates on global sparse structures and its kernels do not depend on how the assembly work was batched; consequently, solver time remains essentially constant across the sweep (Table 5.1 reports GPU solve at 128k as representative). This separation is useful operationally: `VECTOR_SIZE` can be tuned as an assembly parameter without retuning the solver configuration.

### 5.4.4. Why the 3D campaign stops at $\sim$ 32M elements

The 3D sweep was intentionally limited to the largest case that could be executed reliably on a single MareNostrum 5 ACC node without exceeding device-memory constraints. While the 2D campaign reaches 64M elements, the corresponding 3D case would require $N_x = 400$ ($400^3 = 64$M), which increases the working set substantially. In 3D, the element kernel and the solver pipeline must keep in memory not only the primary unknowns but also mesh/connectivity, CSR sparse-matrix structures, temporary assembly buffers, and auxiliary vectors used by the Krylov solver. The overall footprint grows faster in 3D because (i) each element involves more geometric data and more quadrature work, and (ii) sparse algebra structures and temporaries scale with the number of nonzeros and with additional per-DOF bookkeeping.

In practice, the 64M-element 3D configuration exceeded the memory available to the node in the tested setup (in particular, device-resident allocations required by the offloaded assembly and associated data regions), leading to allocation failures and/or instability. For this reason, the largest 3D case reported in Table 5.1 is $N_x = 317$, corresponding to $\sim$ 32M elements, which fits within the memory limits of the MN5 ACC node while still providing a clear large-problem

regime for performance analysis.

### 5.4.5. End-of-chapter conclusions

The combined evidence from Fig. 5.1 and Table 5.1 supports four conclusions for the porting-and-results chapter. First, GPU offload yields very large reductions of time in the compute-dominant phases (assembly + solve), reaching order $10^2$ speedups at the largest manufactured cases under the best pack configuration. Second, the sensitivity to `VECTOR_SIZE` is strongly dimension-dependent: 2D benefits substantially from larger packs because overhead and short-kernel effects remain visible, while 3D reaches a throughput-limited regime earlier and shows diminishing returns with mild non-monotonicity at extreme pack sizes. Third, `VECTOR_SIZE` is an assembly-only knob in this workflow: it reshapes pack granularity and kernel launch structure, but it does not affect solver cost per iteration. Finally, mesh/graph preparation is not yet a meaningful target in the present analysis due to irregular, atomic-heavy behavior in the current porting path; the thesis therefore focuses on the phases where the OpenACC pack-based strategy is directly applicable and where the measured benefit is both robust and interpretable.

# 6 | Conclusions and Future Works

This thesis addressed the acceleration of the implicit Advection–Diffusion–Reaction (ADR) workflow in Alya by targeting the assembly stage on modern GPU-accelerated HPC systems. The work combined a system-aware characterization of the target platform, a CPU baseline analysis that motivated where acceleration is impactful, and a single-source porting strategy based on OpenACC and pack-based element processing. The resulting implementation preserves Alya's production constraints (modularity, object-oriented data structures, and shared services) while exposing the parallelism and memory access structure required to obtain meaningful speedups on GPUs.

The chapter summarizes the main contributions and findings, discusses the practical implications for Alya users and developers, and outlines the most relevant directions for future work, particularly those required to move from a validated unit-test workflow toward sustained production performance on unstructured meshes and at scale.

## 6.1. Summary of contributions

This work delivered a coherent porting and performance-engineering pathway for implicit ADR assembly in Alya, centered on four main contributions.

### 6.1.1. Architecture-aware experimental context

A detailed characterization of the target MareNostrum 5 accelerated node topology was provided to motivate locality-aware choices for binding and affinity (CPU sockets, NUMA domains, GPU attachment, and network endpoints). This context is essential in heterogeneous nodes where cross-socket traffic and suboptimal device affinity can degrade both preprocessing and offload efficiency. The discussion in Chapter **??** and Fig. 4.1 established the hardware locality model used as a reference for subsequent optimization decisions.

### 6.1.2. CPU baseline and motivation for GPU targeting

A CPU-only baseline was constructed for both 2D and 3D manufactured ADR configurations to quantify how runtime decomposes across preprocessing, assembly, and solve. The results showed a clear divergence between 2D and 3D at matched unknown counts: 2D becomes solver-dominated due to rapidly growing Krylov iteration counts, whereas 3D retains a substantial assembly share even at large problem sizes. This baseline justified focusing the GPU port on assembly, where acceleration yields visible end-to-end benefits in 3D while also exposing the solver-limited nature of 2D. The key observations were summarized in Section 4.4 and Fig. 4.3.

### 6.1.3. Single-source GPU porting through pack-based assembly

The assembly implementation was organized around packs (vectors/batches) of homogeneous elements, exposing two levels of parallelism: coarse-grain parallelism over packs and fine-grain parallelism over lanes. The lane dimension acts as the portability bridge: it is SIMD-friendly on CPUs and maps naturally to SIMT threads on GPUs under OpenACC. A lane-first memory layout was adopted to improve locality and coalescing, aligning the data organization with the GPU execution model while preserving a unified source code. This design and its implications were developed in Section 5.1.1.

### 6.1.4. Correctness enabler for object-oriented data: manual deep copy with attach

Alya's pointer-rich, object-oriented data structures require explicit management of the device-visible pointer graph. A robust pattern based on staged device allocation/copyin of leaf arrays followed by pointer attachment from leaves to root was described and applied to the ADR workflow. This contribution is primarily about correctness and reproducibility: it provides deterministic rules for making nested derived types safe for device dereferencing, avoiding illegal-address failures and making data residency a controllable performance knob. The approach was detailed in Section 5.2.

## 6.2. Main findings

### 6.2.1. Assembly is fundamentally memory-oriented, with contention as the dominant risk

Across the gather/compute/scatter structure of finite-element assembly, performance is governed primarily by effective memory behavior rather than peak floating-point throughput. Indirect gathers weaken coalescing and reduce cache reuse, while scatters introduce correctness-driven synchronization through atomic updates when elements share nodes or degrees of freedom. As discussed in Section 2.3, improving coalescing and locality is necessary but not sufficient when contention dominates the scatter phase; in that regime, reducing conflicting updates becomes the main lever.

### 6.2.2. Pack size is a granularity knob: large impact in 2D, diminishing returns in 3D

The pack-size sweep demonstrated that the OpenACC `VECTOR_SIZE` parameter primarily controls batching granularity and therefore the number of kernel invocations and runtime-managed overheads. In 2D, where per-element work is lighter and kernels are shorter, larger packs substantially reduce launch-frequency effects and move execution toward a steadier streaming regime. This produces strong and largely monotonic speedup gains as `VECTOR_SIZE` increases.

In 3D, per-element work is heavier and dominant kernels already operate in a sustained bandwidth-oriented regime at moderate pack sizes. Consequently, further increases in `VECTOR_SIZE` yield smaller marginal gains and can introduce second-order resource effects (register pressure, occupancy reduction, cache working-set growth, and altered scatter contention), leading to weak dependence or mild non-monotonic behavior. These trends were summarized in Section 5.3 and Fig. 5.1.

### 6.2.3. End-to-end speedup depends on where the global bottleneck sits

The CPU baseline and the GPU assembly study jointly imply that end-to-end improvement is constrained by different limits in 2D and 3D. In 3D, assembly remains a meaningful fraction of the workflow; accelerating it translates into visible reductions in total runtime without changing the solver configuration. In 2D, solver time dominates at large resolutions due to iteration growth, so assembly acceleration alone cannot yield proportional end-to-end gains. This is not a weakness of the port; it is a structural property of the algorithmic pipeline under the chosen solver and preconditioner baseline. The implication is that future gains in 2D require addressing solver/preconditioner effectiveness in addition to assembly throughput.

## 6.3. Practical implications for Alya GPU execution

### 6.3.1. Guidelines for choosing pack size

The results support different tuning guidance depending on dimension and workload intensity. For 2D-like regimes, larger `VECTOR_SIZE` values are generally beneficial because they amortize launch and runtime overhead and reduce short-kernel effects. For 3D-like regimes, moderate pack sizes often achieve near-optimal performance because kernels are already long enough to be throughput-limited; pushing to maximal pack sizes can trigger resource-pressure effects without reducing total work.
In the final thesis version, this section should report the two headline choices observed in your experiments: (i) the best-performing `VECTOR_SIZE` range for 2D at large problems, and (ii) the best-performing `VECTOR_SIZE` range for 3D together with any observed slowdown region.

### 6.3.2. Device data residency remains a first-order requirement

Independent of kernel tuning, sus tained performance requires controlling host–device data motion. The explicit deep-copy/attach pattern is not only a correctness mechanism but also the foundation for persistent device-resident data regions across repeated kernel calls. This aligns with the observed sensitivity of end-to-end time to kernel granularity in 2D: when kernels are short, unnecessary synchronization and implicit transfers become disproportionately expensive. Keeping read-mostly mesh/connectivity data resident and minimizing device updates to the essential evolving fields is therefore a necessary condition for predictable performance.

### 6.3.3. Scatter contention is the main algorithmic obstacle to higher throughput

When atomic contention dominates, further improvements from packing and coalescing diminish. In production unstructured meshes, this risk increases because connectivity irregularity tends to increase both gather inefficiency and update conflicts. Practical optimization should therefore be framed around conflict mitigation strategies (reducing the number of contended updates, changing update order, or introducing staged accumulation) rather than purely around increasing parallelism.

## 6.4. Future work

The following directions are the most direct extensions of this thesis, ordered by their expected impact on sustained production performance.

### 6.4.1. Reducing scatter contention

A primary next step is to reduce the cost of atomic updates in scatter. Candidate strategies include graph coloring or subdomain scheduling tuned for GPUs, staged accumulation (e.g., block-local reductions into shared memory followed by fewer global atomics), and alternative assembly formulations that restructure updates to reduce hot-spot addresses. The pack abstraction provides a natural unit of work for experimenting with these strategies because it localizes temporary storage and separates gather/compute from global updates.

### 6.4.2. Extending to unstructured and heterogeneous production cases

A systematic evaluation on unstructured meshes with mixed element types and realistic boundary sets is required to confirm that the lane-first, pack-based approach maintains efficiency under stronger irregularity. This includes validating pack categorization policies, padding overhead, and the sensitivity of contention to mesh ordering and partition boundaries.

### 6.4.3. Solver and preconditioner acceleration

To overcome solver-limited regimes (especially in 2D) and to enable scalable time-to-solution improvements, future work should target GPU-efficient preconditioning and solver kernels. Options range from improving diagonal/Jacobi variants to introducing stronger algebraic preconditioners (e.g., block-Jacobi, domain decomposition, or multilevel methods) with GPU-friendly building blocks. Because Krylov solvers introduce global reductions and halo exchanges, the performance model must consider overlap and synchronization costs explicitly.

### 6.4.4. Multi-GPU and multi-node performance with distributed execution

A natural extension of this work is to move from single-GPU node studies to distributed executions that exploit multiple GPUs per node and multiple nodes. This requires validating that the pack-based assembly remains efficient when multiple MPI ranks target distinct GPUs and when halo exchanges and global reductions appear in the overall time-to-solution. At node level, topology-aware rank placement becomes essential so that each MPI rank communicates

through the closest GPU and network endpoint, avoiding cross-socket traffic and maximizing effective bandwidth. At multi-node level, strong- and weak-scaling studies can quantify how assembly acceleration interacts with distributed sparse linear algebra, clarifying whether the workflow remains assembly-limited or transitions to communication-limited behavior as concurrency grows.

In this direction, an important practical target is a multi-GPU configuration where each GPU is driven by one or more MPI ranks, leveraging production mechanisms such as GPU multi-process scheduling when the rank-to-GPU ratio exceeds one. The goal is to preserve the existing MPI programming model while exploiting all available accelerators and measuring the resulting impact on time-to-solution and scalability.

### 6.4.5. Energy-to-solution and comparison against CPU-only nodes

Beyond wall-clock time, future work should evaluate the energy efficiency of the accelerated workflow by comparing GPU-accelerated runs against CPU-only executions on contemporary supercomputing processors. This comparison should be formulated in terms of energy-to-solution and performance-per-watt, in addition to speedup. A rigorous study would collect node power traces (or energy counters where available) during the same workload phases considered in this thesis (assembly and solve), and would report both absolute energy and normalized metrics such as Joules per degree of freedom or Joules per element assembled.

This perspective is particularly relevant for throughput-oriented kernels such as assembly, where GPUs often provide higher effective memory bandwidth per watt than CPU-only nodes. Conversely, solver-dominated regimes may exhibit smaller energy advantages unless solver kernels are also accelerated or communication overheads are reduced. Establishing this comparison would provide a stronger basis for assessing the global benefit of the port in realistic procurement- and scheduling-driven HPC contexts.

# Bibliography

[1] Application Performance on Accelerators. Technical Report D7.1.2, PRACE-1IP, 2012.

[2] F. Banchelli, M. Garcia-Gasulla, F. Mantovani, J. Vinyals, J. Pocurull, D. Vicente, B. Eguzkitza, F. C. C. Galeazzo, M. C. Acosta, and S. Girona. Introducing marenostrum5: A european pre-exascale energy-efficient system designed to serve a broad spectrum of scientific workloads, 2025.

[3] Barcelona Supercomputing Center (BSC). Marenostrum 5, 2026. URL `https://www.bsc.es/marenostrum/marenostrum-5`.

[4] R. Borrell, J. C. Cajas, D. Mira, A. Taha, S. Koric, M. Vázquez, and G. Houzeaux. Parallel mesh partitioning based on space filling curves. *Computers & Fluids*, 173:264–272, 2018.

[5] R. Borrell, A. Taha, D. Dosimont, M. Garcia-Gasulla, G. Houzeaux, M. Vázquez, E. López, J. Jorba, and P. Quiles. Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture. *Future Generation Computer Systems*, 107:31–48, 2020.

[6] R. Codina. Stabilization of incompressibility and convection through orthogonal subscales in finite element methods. *Computer Methods in Applied Mechanics and Engineering*, 190, 2000.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6 edition, 2019.

[8] G. Houzeaux, M. Vázquez, R. Aubry, and J. M. Cela. A massively parallel fractional step solver for incompressible flows. *Journal of Computational Physics*, 228(17), 2009.

[9] G. Houzeaux, M. L. de la Cruz, and M. Vázquez. Parallel uniform mesh subdivision in Alya. Technical Report WP197, PRACE, 2011. PRACE White Paper.

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

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

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

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

[14] *CUDA C++ Best Practices Guide*. NVIDIA, 2025. URL https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[15] *CUDA C++ Programming Guide*. NVIDIA, 2025. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[16] *NVIDIA HPC Compilers User's Guide*. NVIDIA, 2025. URL https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html.

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

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

[19] OpenACC Organization. Openacc best practices guide (porting and portability guide). GitHub repository, 2025. URL https://github.com/OpenACC/openacc-best-practices-guide.

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

[21] OpenMP Organization. Openmp for gpu, introduction to gpu architecture, 2025. URL https://enccs.github.io/openmp-gpu/gpu-architecture/.

[22] H. Owen, D. Ernst, T. Gruber, O. Lehmkuhl, G. Houzeaux, G. Gasparino, and G. 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.

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

[24] RES (Red Española de Supercomputación). MareNostrum 5 ACC (Accelerated Partition) — System Overview, 2026.

[25] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2 edition, 2003.

[26] TOP500. Marenostrum 5 acc — system entry, 2026. URL https://top500.org/system/180238/.

[27] M. Vázquez, A. Rubio, G. Houzeaux, M. González, J. Giménez, and V. Beltrán. Xeon phi

performance for HPC-based computational mechanics codes. PRACE technical report / dataset record, 2014. URL https://zenodo.org/record/827121.

[28] M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Arís, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, E. Dering Burness, J. M. Cela, and M. Valero. Alya: Multiphysics engineering simulation towards exascale. *Journal of Computational Science*, 14, 2016.

# A | Guide on OpenACC directives

Table A.1 summarizes the OpenACC directives and clauses actually used in this work. The selection reflects the needs of the Alya porting effort: explicit parallel regions and loop mapping for GPU kernels, explicit data lifetime management to keep large arrays resident on the device, atomic updates for correctness in scatter phases, and pointer attachment to make derived-type hierarchies valid on the accelerator.

Table A.1 **OpenACC directives and clauses used in this work.** Summary of the OpenACC constructs referenced throughout this thesis. The table is organized by purpose—compute and loop control, data-region lifetime, synchronization, and data mapping—and lists the specific directives and clauses employed to express GPU parallelism, manage host–device data movement, and preserve correctness in Alya's ADR offload. It is not intended as a complete OpenACC reference, but as a compact "vocabulary" of the features actually used in the implementation and discussed in the porting and performance chapters.

| Name | Description |
|---|---|
| **Compute and loop constructs** | |
| `parallel` | Starts a parallel region on the accelerator where gangs, workers and vectors execute concurrently. |
| `serial` | Starts a region on the accelerator executed by a single gang/thread, without implicit gang/worker/vector parallelism. |
| `kernels` | Marks a region in which the compiler analyzes the code and generates one or more accelerator kernels for detected parallel loops. |
| `loop` | Marks a loop for execution on the accelerator, allowing its iterations to be mapped to gangs, workers or vectors. |
| `routine` | Declares a function or subroutine that is compiled for device execution and can be called from within accelerator regions. |
| **Data constructs and directives** | |
| `enter data` | Creates device allocations and/or copies data from host to device, establishing or extending the lifetime of data on the accelerator. |
| `exit data` | Destroys device allocations and/or copies data from device back to host, ending the associated data lifetime on the accelerator. |

| Name | Description |
|---|---|
| `declare` | Associates variables at declaration with device-resident storage and data clauses, controlling their lifetime independently of explicit regions. |
| **Synchronization and atomics** | |
| `atomic` | Ensures that a specific memory operation (read, write, update or capture) on a variable is performed atomically across accelerator threads. |
| `update` | Explicitly transfers already-mapped data between host and device during an active data region (host→device or device→host). |
| **Data-sharing and control clauses** | |
| `private` | Gives each parallel context (e.g. gang, worker, vector lane or thread) its own private copy of the listed variables. |
| `reduction` | Creates private copies of the listed variables and combines them at the end of the region or loop using the specified reduction operator. |
| `if(condition)` | Controls whether the associated construct executes on the accelerator or falls back to host execution depending on the boolean condition. |
| `default(none)` | Disables implicit data mapping; all non-scalar variables used in the region must appear explicitly in data or data-sharing clauses. |
| `default(present)` | Assumes referenced variables are already present on the device and uses those allocations; typically errors if the data is not present. |
| **Data mapping clauses** | |
| `copy(var-list)` | Allocates storage on the device, copies data from host to device on region entry and copies it back from device to host on region exit. |
| `copyin(var-list)` | Allocates storage on the device and copies data from host to device on entry; no implicit copy back on exit. The optional `readonly:` qualifier documents that the data is not modified on the device. |
| `copyout(var-list)` | Allocates storage on the device and copies data from device to host on exit; no implicit copy in on entry. The optional `zero:` qualifier requests zero-initialization on the device. |
| `create(var-list)` | Allocates storage on the device for the listed variables without any implicit data transfer. The optional `zero:` qualifier requests zero-initialization. |
| `attach(var-list)` | Attaches host pointer variables to existing device allocations so that pointer dereferences on the device refer to the correct targets. |

| Name | Description |
|---|---|
| `detach(var-list)` | Detaches host pointer variables from their associated device allocations, removing the pointer–target association on the device. |
| `delete(var-list)` | Deallocates device-resident storage for the listed variables without performing additional data transfers |

# List of Figures

# List of Tables

# Acknowledgements

—