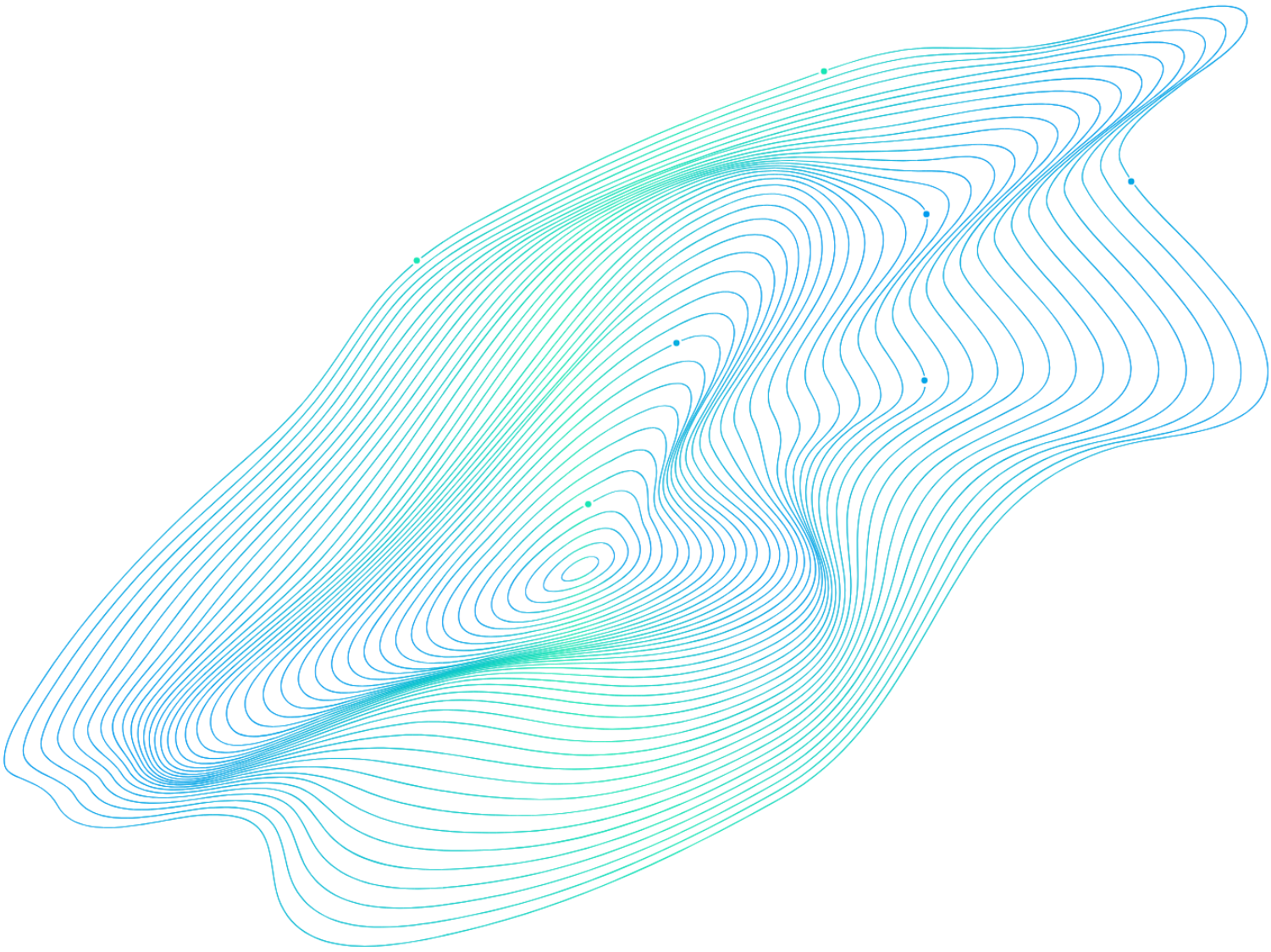
EPICURE

Unlocking European-level HPC Support

HIGH-LEVEL SPECIALISED APPLICATION SUPPORT SERVICE IN HIGH-PERFORMANCE COMPUTING (HPC)



Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the European High Performance Computing Joint Undertaking under grant agreement No.101139786. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or EuroHPC Joint Undertaking. Neither the European Union nor the granting authority can be held responsible for them.

EPICURE

DELIVERABLE 3.1

Report on code optimization for Year 1



Project Title	High-level specialised application support service in High-Performance Computing (HPC)
Project Acronym	EPICURE
Project Number	101139786
Type of Action	DIGITAL JU Simple Grants
Topic	DIGITAL-EUROHPC-JU-2022-APPSUPPORT-01-01
Starting Date of Project	2024-02-01
Ending Date of Project	2028-01-31
Duration of the Project	48 months
Website	epicure-hpc.eu

Work Package	3
Task	3.1 -3.3
Lead Authors	Andrew Emerson (CINECA),
Contributors	Laura Bellentani (CINECA), Emmanuele Vitali (CSC), Ziyad Albonoby (BSC), Trevor Hall (BSC), Francesco Bongiovanni (LuxProvide)
Peer Reviewers	David Vicente (BSC) Joao Barbosa (IT4I)
Version	2.0
Due Date	31/01/2025
Submission Date	29/01/2026 (2nd submission, after the review comments)

Dissemination Level

X	PU: Public
	SEN: Sensitive – limited under the conditions of the Grant Agreement
	EU-RES. Classified Information: RESTREINT UE (Commission Decision 2005/444/EC)
	EU-CON. Classified Information: CONFIDENTIEL UE (Commission Decision 2005/444/EC)
	EU-SEC. Classified Information: SECRET UE (Commission Decision 2005/444/EC)

Version History

Revision	Date	Editors	Comments
0.1	04/12/2024	Andrew Emerson (CINECA)	First outline
0.2	23/12/2024	Andrew Emerson (CINECA)	First draft for review by Application Support Teams
0.3	10/01/2024	Andrew Emerson (CINECA)	Ready for internal review
0.4	17/01/2024	Andrew Emerson (CINECA)	Updated according to reviewer's comments. Ready for PMB view
1.0	31/01/2025	Eeva Harjula (CSC)	Final quality check performed by the PMO. File ready to be submitted.
1.1	13/01/2026	Andrew Emerson (CINECA)	The document has been modified considering comments of EU-appointed reviewers: - Corrected incorrect statements, typos and duplications - Unified style for each project report - Added sections and summary table highlighting achievements of each optimisation activity - Added a summary of the collaborators for each project (in Table 1)
1.6	15/01/2026	Alberto Lanzanova (CSC) Joao Barbosa (IT4I)	Review of the document content. Correction of few typos, formatting and minor changes to stylistic choices.
1.8	19/01/2026	Andrew Emerson (CINECA)	Consolidated version including the comments from the last round of review.
2.0	29/01/2026	Alberto Lanzanova (CSC) Virvatuli Uusimäki (CSC)	Final quality check performed by the PMO. File ready to be submitted.

Glossary of Terms

Item	Description
stride	Difference in the indices between successive accesses of an array in a loop. A unit stride is the most efficient because it allows memory to be accessed contiguously, maximizing cache usage and other optimisations.
Loop unrolling	Explicitly writing the loop iterations. For short loops, this often brings efficiency advantages since it avoids the overheads in managing loops.
Loop fusion	An optimisation and loop transformation which replaces multiple loops with single one.
DFT	Density Functional Theory, a quantum-mechanical method used in chemistry and physics to calculate the electronic structure of atoms, molecules and solids.
AST	Application Support Team, the team of HPC specialists assigned by EPICURE to provide support to a EUROHPC project.

Executive Summary

This document represents the first optimisation report of EPICURE and covers the first year of the project. The activities described are level 3 optimisations, i.e. those that go beyond simple porting or performance benchmarks, and very often involving deep profiling and changes to the source code.

Given that only a subset of projects in EPICURE required support at this level, we have focused only on these, and in particular those activities which have either finished or have made significant progress in improving the performance of the application codes at the time of writing. We have also included one project which, despite considerable efforts from the support team, were not able to obtain performance gains. The reason for reporting on this action was to highlight the difficulties in optimizing application programs which are used in some communities but not regularly maintained by HPC specialists. Of the remaining projects it is significant that Python is commonly used, either as a wrapper to a C/C++ or FORTRAN code, or as part of a workflow.

In fact, some optimisations concentrated on the Python codes which, because of the high-level nature of the language, often performed operations in a very opaque and inefficient manner (i.e., by making multiple copies of large objects). Other applications have employed OpenACC acceleration to run on GPUs or used machine learning workflows, which can sometimes represent a challenge when, for example, containers need to be scaled across multiple HPC nodes. Finally, some optimisations have focused on accelerating loops, the traditional targets for code optimisation.

Table of Contents

1. Introduction 8
2. Optimisation Reports 9
2.1 Summary of projects for code optimisation 9
2.2 Project 2: Design Principles for Non-Aromatic Fluorescence in Biology (DpNAFB) 10
2.3 Project 5: From methane and iron nanoparticles - catalytic process inside a reactor 14
2.4 Project 8: HiPsum - Translating openQCD-fastsum to AMD 16
2.5 Project 9: AtmoRep 18
2.6 Project 11: Large Eddy Simulation of Supersonic Air Intakes 23
2.7 Project 13: Enhancing AI of EU institutional LLMs for the public sector 28
2.8 Project 16: DEAREL: Large-scale Deep rEinforcement leARning for activE fLow control in wings 33
3. Conclusions 38

List of Figures

Figure 1 Nsight Systems profile trace of the wbset_init kernel (project 2) 11
Figure 2 Runtimes for 48 atoms with the baseline and optimised versions of the kernels (project 2) 12
Figure 3 Runtimes for 384 atoms with the baseline and optimised versions of the kernels (project 2) 12
Figure 4 Improvements in performance in wbse_init after optimisation (project 2) 13
Figure 5 Measured training speed after optimisation (project 9). 21
Figure 6 MAQOA profiling show 47% of vectorized instructions 24
Figure 7 Speed up for a problem size of 512x512x512 on Meluxina 24
Figure 8 Vampir trace visualizer 25
Figure 9 Illustration of an MPI efficiency problem called “Late Sender Wrong Order”. This occurs when pairs of MPI SEND – RECEIVE calls are incorrectly ordered. 25
Figure 10 Map profiling of the hybrid OpenMPI+OpenMP code 26
Figure 11 Intrinsics behaviour for non-continuous loads 27
Figure 12 CPU floating-points operations between original (top) and improved (bottom) codes 27
Figure 13 GPU events in the forward-backward pass as described in the text (project 13) 31
Figure 14 Zoom of the Figure 6 trace showing the sendrecv call blocking the synchronisation (project 13) 32
Figure 15 Times for the relevant events with (red) and without CPU offload (blue) (project 13) 32
Figure 16 Location of the probes in the simulation (project 16) 34

1. Introduction

This document represents the first optimisation report of WP3, which provides level 3 application support to the projects managed by EPICURE. This level of support goes beyond porting or simple performance analysis, instead focusing on optimisations which may require a deep profiling or tracing and potentially code modifications, to obtain the best performance possible on the chosen HPC system. This can be contrasted with the level 2 application support provided by WP2 and the overall management of these activities provided by WP4, which are described in separate documents. In the sections below we describe the optimisation activities for those projects which requested level 3 support and for which significant activity had been reported by the application support teams (ASTs) before the time of writing of the document. It is worth noting that most projects in the EPICURE environment requested only level 2 optimisations and these are described in the associated deliverable D2.1. For the allocation of the ASTs to the projects the reader is instead directed to the WP4 deliverables.

The structure of this document is as follows: first we give a table of the projects and then for each one we detail its background, a short summary of the WP2 activity, the optimisation objectives, the status before the work done and the results obtained. We also where appropriate highlight the lessons learned, useful information that can be passed onto other ASTs (or researchers) for future projects. In all these activities we stress that there is always a close collaboration between the AST and researchers, sometimes to the extent where optimisations have been pushed into the repository maintaining the application source code.

We remark that this document represents the second version of the deliverable D3.1, modified to emphasise the improvements in performance, resource usage or code usability, resulting from EPICURE optimisation.

2. Optimisation Reports

2.1 Summary of projects for code optimisation

In Table 1 we give a summary of the projects for which optimisation activities in WP3 have been carried out. As can be seen from the table, some of these projects are still on-going at the time of writing, but since the optimisation so far achieved has been significant, we decided to include them anyway in this report. In later reports further details from these projects may be given.

Lead Partner	Collaborators	Project ID	EuroHPC project type	Project Title	Application codes	Host Partition	Optimisation Status
BSC	-	2	Extreme	Design Principles for Non-Aromatic Fluorescence in Biology (DpNAFB)	Python wrapper with FORTRAN, MPI and OpenACC	MN5	Completed
CINECA	-	5	Extreme	From methane and iron nanoparticles - catalytic process inside a reactor	Python	Leonardo	Completed
CSC	-	8	Extreme	HiPsum - Translating openQCD-fastsum to AMD	C	LUMI-G	Abandoned
BSC	-	9	Regular	AtmoRep, classical weather prediction	Python, NumPy, Zarr and PyTorch	MN5	Completed
LUXPROVIDE	-	11	Regular	Large Eddy Simulation of Supersonic Air Intakes	Code based on C++ and MPI	MeluXina CPU	Completed
CINECA	-	13	Development + AI	Enhancing AI of EU institutional LLMs for the public sector	Python frameworks for ML (e.g. DeepSpeed)	Leonardo	Completed
BSC	-	16	Regular	DEAREL: Large-scale Deep rEinforcement leARning for activE fLow control in wings	FORTRAN(CFD), Python (ML-TensorFlow), MPI+OpenACC	MN5 ACC	Completed

Table 1 List of projects for the code optimisation activity in year 1 of the project

2.2 Project 2: Design Principles for Non-Aromatic Fluorescence in Biology (DpNAFB)

2.2.1 Project Overview

The DpNAFB project was part of an ERC-funded effort to investigate the unusual optical properties of hydrogen-bond networks. To support this research, various computational tools were used, including the WEST code (https://west-code.org/), which performs periodic calculations on hydrogen-bonded systems such as amino acids and liquid water. Optimizing WEST on the cluster is essential for maximizing performance and accelerating these investigations.

2.2.2 Code description

WEST's code performs large-scale many-body perturbation theory calculations, providing electronic and optical spectroscopic data for complex systems. It implements Time-Dependent Density Functional Theory (TD-DFT) with semi-local and hybrid functionals and uses GPU acceleration to speed up calculations and enable parallel thermal averaging across different configurations. To calculate absorption and emission spectra, users first run SCF and NSCF steps in Quantum Espresso, then pass these results to WEST. Both codes are GPU-compatible. Notably, emission spectra require running WEST and WESTpy.

The main application is written in Fortran, parallelized with MPI and OpenACC, and uses libraries such as cuFFT, cuBLAS, and ELPA for optimized performance.

2.2.3 Optimisation objectives

The aim was to reduce the runtime of the WEST code by optimizing the most time-consuming kernels detected during the profiling process.

2.2.4 Status before optimisation

Since profiling with Nsight Systems identified three kernels that consumed about 53% of the total execution time, these parts of the code were targeted for WP3 optimisation. Prior to our optimisation efforts, the preprocessing program **`wbse_init`** took an average of 179.41 s with 16 MPI processes for the 384-atom case; after optimisation, this was reduced to 88.56 s, an improvement of 2.02x. In contrast, the solver program **`wbse`** previously took about 179.03s per iteration for the same processed input with 16 MPI processes, which we were able to reduce to 162.02s, an improvement of approximately 9.5%.

.

2.2.5 Code optimisation and results

The first step in code optimisation was in fact to resolve a bug in the integration of NumPy within the Fortran library *Forpy* which caused floating point errors, crashing the program. To prevent this happening floating point errors were disabled during the initialization of forpy:

```
CALL IEEE_SET_HALTING_MODE(IEEE_ALL,.FALSE.)
```

Further fixes were made to the Python implementation of WEST which were merged into the master branch following a pull request to the authors.

Optimisation of wbse_init.x

The next step in the activity concentrated on the `wbse_init.x` kernel where a thread pool was created to perform background IO writes. This allowed us to overlap the computation and IO as can be seen in the NVIDIA Nsight trace in Figure 1.

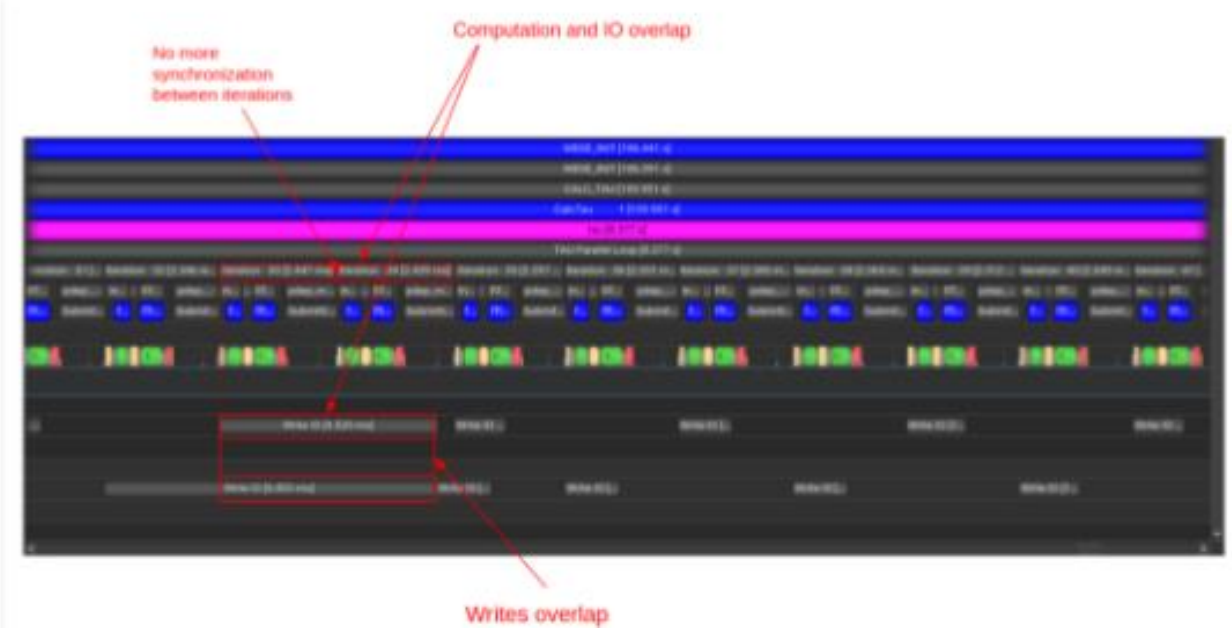


Figure 1 Nsight Systems profile trace of the wbset_init kernel (project 2)

In conjunction to this, three OpenACC kernels were rewritten to Cuda achieving speedups of **7.15x**, **6.69x**, and **3.65x** respectively. The results of our optimisation efforts are summarised in Figure 2 where an input lattice of 48 atoms was used and in Figure 3 where a larger system of 384 atoms was run.

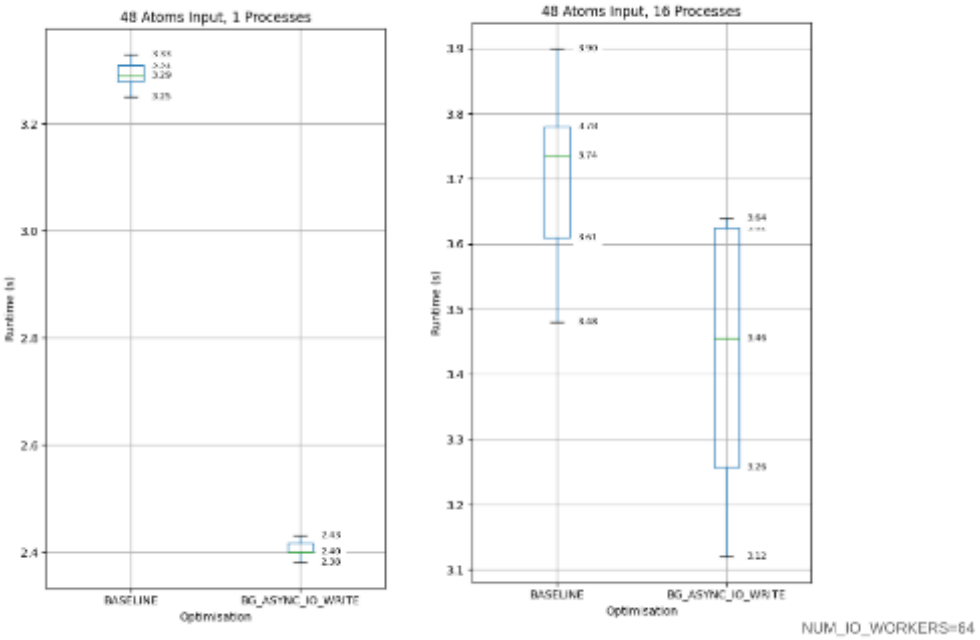


Figure 2 Runtimes for 48 atoms with the baseline and optimised versions of the kernels (project 2)

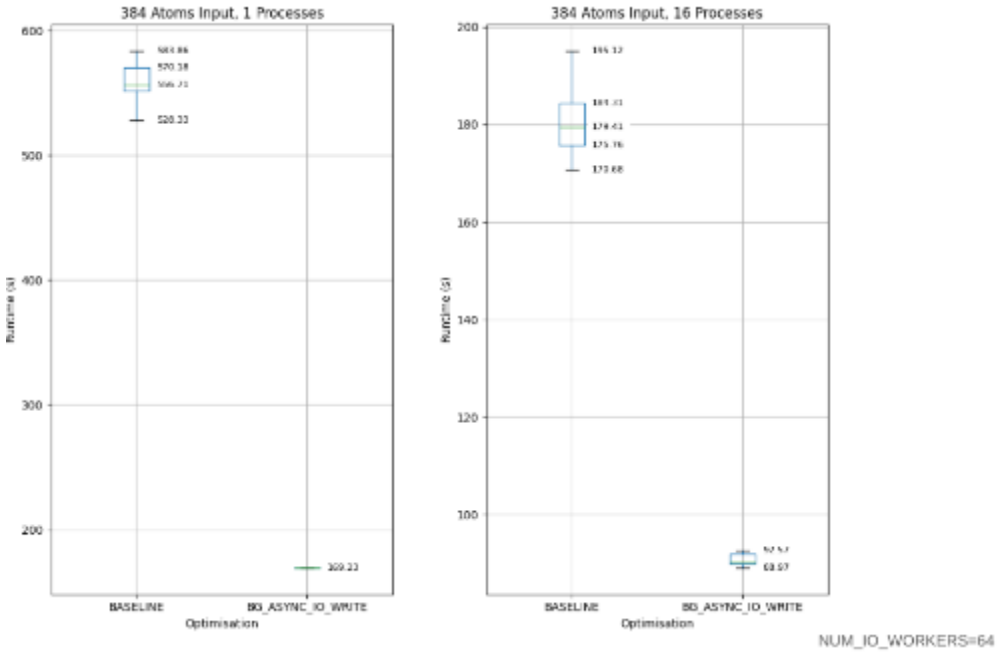


Figure 3 Runtimes for 384 atoms with the baseline and optimised versions of the kernels (project 2)

From the figures above, we see that the optimized version consistently outperforms the unoptimized baseline, achieving speedups from **1.12x** to **3.13x**, with larger gains for bigger inputs. However, when the same input (e.g., 384 atoms) is distributed across multiple nodes, each node handles fewer atoms, allowing both the baseline and the optimized version to benefit from parallelization. As a result, the difference in performance between the two methods becomes smaller. For example, when running on four nodes, the speedup for the medium input drops to **1.76×**. This does not imply the optimisation is less effective on multiple nodes; rather, the baseline also improves with parallel processing, reducing the performance gap between them.

Optimisation of wbse.x

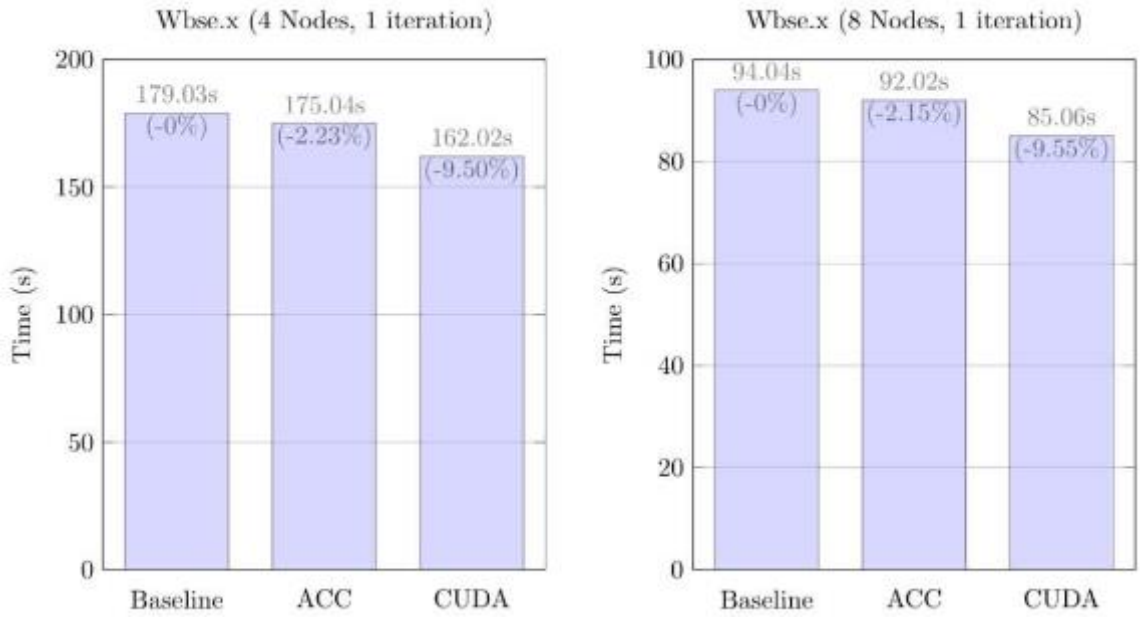


Figure 4 Improvements in performance in wbse_init after optimisation (project 2)

We focused our optimisations on improving memory access patterns and parallelization in wbse.x. Through loop reordering, CUDA kernel implementations, and vectorization with double2, we achieved significant speedups. Starting with OpenACC and later moving to CUDA when we hit memory limits, we saw improvements of up to **17.53x** in key sections. These optimisations scaled well across configurations, giving us **2x** speedup on single nodes while maintaining good performances on multi-node setups (see **Error! Reference source not found.** for the multi-node results).

2.2.6 Improvements achieved

The optimized version consistently outperforms the baseline, with 1.12x–3.13x speedups. The speedup slightly degrades on more nodes, but the optimisation remains beneficial. In wbse.x, memory and parallelization improvements (CUDA, loop reordering, double2) deliver up to 17.53x kernel gains and about 9.5% improvement per iteration on 4 nodes, with a similar behaviour on 8 nodes.

2.2.7 Summary and lessons learned

Based on the experience of this project we can outline some of the activities which worked and those that did not as follows:

What worked:

- Creating C bindings to pthreads to avoid the segmentation fault caused by nvfortran, allowing us to create a thread pool for parallel async io writes.
- Loop reordering and unrolling loop in OpenACC which were then mapped to thrust C kernels using a Fortran-C interface.
- OpenACC reduction is not as fast as CUDA based reduction.

What didn't work:

- Fortran native async IO using did not exhibit the intended behaviour.
- IO writes from Fortran native threads did not work as expected instead the writes remained serialized even though they were executed by different threads.
- Fortran-C bindings to pthreads did not work with nvfortran compiled code, as it always caused thecompiled program to segfault.
- Batched calls while provide a good speed up can be overshadowed by data movement especially for small inputs. (This was tried with wannier JADE where a rotation matrix was applied to A along its 3rd dim. RAR^T)
- Batching FFTs lead to out of memory exceptions when a larger input was used and hence it was avoided.

2.3 Project 5: From methane and iron nanoparticles - catalytic process inside a reactor

2.3.1 Project Overview

In this project atomistic simulations were carried out to understand the turquoise hydrogen production process, which allows hydrogen to be obtained from methane dissociation and avoids the release of greenhouse gasses by capturing carbon and converting it into solid carbon nanotubes. Simulations were managed via Tribchem, a workflow generation and management tool, that allows automatic high-throughput DFT calculations to be performed involving solid surfaces and interfaces.

2.3.2 Code description

TribChem is an open-source Python program built on top of Atomate, Pymatgen, FireWorks and MongoDB and uses external applications (e.g. VASP or QuantumEspresso) to run DFT calculations on the GPUs on the Leonardo Booster. In this project we only considered optimisations of the Python programs running on the CPUs of the Leonardo Booster.

2.3.3 Optimisation objectives

Typically, DFT calculations (carried out with GPU-optimized codes) represented the most resource-consuming part of the workflows. It was realized though that, when increasing the system size, Python-based functions, used for pre- and post-processing analysis, become a true bottleneck.

EPICURE support was requested to optimize these functions, in particular two parts of the project workflow were identified as critical:

- Task 1. A pre-processing task devoted to the search of the so-called "high symmetry points" and of the unique configurations for the interface under examination. Reported (serial) timings on Leonardo Booster for three different interfaces (i.e., input cases) are given in Table 2.
- Task 2. A post-processing analysis where an RBF (Radial Basis Function) interpolation of the potential energy surface is performed. Reported (serial) timings on Leonardo Booster are in Table 3.

2.3.4 Status before optimisation

The following actions were carried as part of the WP2 activity:

- Update of Python environment to later versions.
- Profiling the unit test for Issue 2 (RBF interpolation) with cProfile (a common Python profiler).
- Implementation/extension of unit tests reproducing the two bottlenecks.

Based on the profiling results it was found that over 99.9% of the time was spent in the Python Rbf.__init__ function.

2.3.5 Code Optimisation and Results

One immediate effect of the upgrade of the python implementation was a 20x increase in performance which was tracked down to the different linear algebra libraries used by the Python scipy module – netlib in the preceding version of scipy (version 1.9.1) and the more efficient libblas in the later version of the library (1.14.1). In addition, an update of the `scipy.interpolate.Rbf` to `scipy.interpolate.RBFInterpolator` provides an additional 30% speed-up on the unit test for the system sizes considered in the proposal.

The timings of the test cases resulting from these two optimisations can be summarised in Table 2.

Interface	Old Rbf/old environment	old Rbf/new environment	new Rbf/new environment
Ag(111)/Cu(111)	54s	12s	11s
Al(111)/Cu(111)	1h18m	3m38s	2m12s
Ti(001)/Mg(001)	> 17h	43m49s	28m27s

Table 2 Timings for the three interface inputs before and after the changes in Rbf and environment.

As can be seen from the Table 2 the optimisations have resulted in very significant performance improvements. A further optimisation was applied by parallelizing a double loop in the Python code which, based on the profiling with cProfile, was estimated to take 89% of the runtime. By parallelizing this loop over cores (9 cores on Leonardo) with the multiprocessing package it was possible to obtain further speedups as show in Table 3.

Interface	Serial	Parallel
Ti(001)/V(110)	2m12s	23s
Ti(001)/Fe(110)	38m45s	5m33s
Ti(001)/Cr(110)	1h0m1s	9m4s

Table 3 Timings for the three inputs after parallelisation. Note that these are different than the ones in the objectives section since the interface sizes are not the same.

2.3.6 Improvements achieved

Thanks to the library and code changes, it was possible to obtain a 6x speedup in task 1 (Python multiprocessing) and a 20x speedup for task 2 (Python environment upgrade).

2.3.7 Summary and lessons learned

The results in this project demonstrate the importance of understanding the libraries used in the workflow and updating to the latest versions.

Other lessons learned included the following:

- cProfile is very useful for detecting bottlenecks in Python code
- using the openBLAS via LD_PRELOAD did not work, possibly due to different symbol names in the scipy linking to BLAS
- there are many profiling or tracing tools for Python, and it takes some experimentation to find the most useful.

2.4 Project 8: HiPsum - Translating openQCD-fastsum to AMD

2.4.1 Project overview

The FASTSUM project invested significant effort into modifications of the OpenQCD code so that it can support anisotropic lattices and stout smearing. This modification of the CPU code resulted in orders of magnitude improvement in performance at the more challenging lighter quark masses. The aim was to port this modified code to AMD GPUs.

The openQCD simulation program is based upon the Hybrid Monte Carlo (HMC) algorithm. The openQCD code contains several advanced techniques that can be selected at run time:

- Nested hierarchical integrators for the molecular-dynamics equations, based on any combination of the leapfrog, 2nd order Omelyan-Mryglod-Folk (OMF) and 4th order OMF elementary integrators, are supported.
- Twisted-mass Hasenbusch frequency splitting, with any number of factors and twisted masses. Optionally with even-odd preconditioning.
- Twisted-mass determinant reweighting.

- Deflation acceleration and chronological solver along the molecular-dynamics trajectories.
- A choice of solvers (CGNE, MSCG, SAP+GCR, deflated SAP+GCR) for the Dirac equation, separately configurable for each force component and pseudo- fermion action.

The CGNE solver has been ported to Nvidia CUDA, so the aim was to port to HIP such that it could run on LUMI-G

The target for EPICURE with this code was to port the current implementation to the target machine (LUMI), converting the existing CUDA CGNE solver in order to enable the possibility of running the accelerated version of OpenQCD on LUMI-G. No further optimisation was requested in the project form.

2.4.2 Code description

The code is a C application, which makes large use of macros and a standard Makefile for compilation. The CUDA solver unfortunately is not integrated in the main repository but instead it is present on a code fork and is no longer maintained. It is not complete, as the code was tried on a NVIDIA architecture and was not able to run. The PI was not aware of this at the beginning of the collaboration.

2.4.3 Optimisation objectives

As described in the previous section, the objective of the EPICURE collaboration was to port the CGNE CUDA solver to HIP. In this way the code would be able to run on LUMI-G.

2.4.4 Status before optimisation

The application could run on NVIDIA GPUs, but not on AMD devices. In addition, the build system was complex and not standard, so it was not easy to deploy on different machines.

2.4.5 Code optimisation and results

The team spent a lot of time reverse engineering the code – not an easy or quick task as the program makes large use of macros to do what in C++ is done with templates. We managed to create a HIP version of the code that successfully compiled (but initially did not run). hipify worked quite well here, since we only had to manually fix a couple of functions, that have slightly different APIs and the script was not able to translate everything correctly (e.g: `cudaLaunchCooperativeKernel` and hipLaunchCooperativeKernel have different number of parameters, with HIP requiring an extra mandatory parameter for the stream, that is optional in CUDA).

By reverse engineering the code it was possible to run the produced code, however:

1. The code wouldn't use the GPU path with the single example provided. It was possible to resolve this, but only by making a number of assumptions about input parameters.
2. When running with the example supplied the code goes in a deadlock. This was found to be due to some hardcoded parameters in the source, such as some parts of the code being run only by rank 0 containing collective operations. We believe that this does not depend on CUDA/hip porting but is a previous issue of the application, which did not use the GPU back-end, but only the testcases which would not compile. As mentioned above, the code is not maintained and therefore the hardcoded options were never removed.
3. We ran with a single process to avoid the deadlock: we obtained the same error with CUDA and HIP. We believe the issue with the code is somewhere else and the porting works. However, this will never be known as the PI came back to us saying that he is no more interested in this aspect and wished to change the scope of the project.

2.4.6 Improvements achieved

The project's objectives were not realised (see below) but the EPICURE team was able to gain some valuable experience in porting CUDA codes to HIP on LUMI (especially with hipify).

2.4.7 Summary and lessons learned

The optimisation project was ultimately unsuccessful - the lack of support from the PI normally would result in EPICURE suspending the WP3 activity but we decided to continue working and include this report in the deliverable for the following reasons:

- In the first few months of EPICURE there was a limited number of EuroHPC projects requesting WP3 optimisation for LUMI sites, therefore we had more resources to allocate to this project.
- It is also useful to report on activities which did not succeed for whatever reason (in this case lack of interest from the PI and poorly written and maintained code).
- Regardless of the project outcome, the experience with porting accelerated codes from one architecture to another has been useful, particularly the use of hipify for porting CUDA source to AMD.

2.5 Project 9: AtmoRep

2.5.1 Project overview

Classical weather prediction models are computationally heavy and exhibit biases. AtmoRep was the first prototype of a stochastic model for atmospheric dynamics. It is a probabilistic, task-agnostic model that can be used for weather forecasting, spatio-temporal interpolation, downscaling and scenario generation.

2.5.2 Code description

The AtmoRep application makes use of the Python packages: NumPy, Zarr and Pytorch (especially the Distributed Data Parallel package) and can run on GPUs (the accelerated partition of MareNostrum5).

2.5.3 Optimisation objectives

These included the following activities:

- reducing run-time/increasing efficiency
- scaling out
- maintaining efficiency at scale
- fine-tuning to run on Marenostrum 5 ACC (Accelerated) partition i.e. the partition containing GPUs.

2.5.4 Status before optimisation

The WP2 activity consisted of:

1. Application Analysis
2. Performance measurement
3. Benchmarking.

After some analysis it was decided to use `cProfile` for profiling, `SnakeViz` and `GraphViz` for visualising the execution paths and `DebugPy` with Visual Studio Code for debugging. Tests were performed on the accelerated partition of MN5 using 1 node, 1 task/node and the SLURM directive `cpus-per-task=80` with the 4 available GPUs. After profiling the code with cProfile and visualising the call path with GraphViz, we saw that most of the time was being consumed in the `__iter__()` function of MultifieldDataSampler (89.62% of the total). The time-consuming part of the `__iter__()` function was `separated` into a new function called process_data(). Further analysis revealed that the "`take()`" function of NumPy inside `process_data()` consumed about **40.15%** of the total time and the Zarr function __get_item__ consumed about **46%** of the total time. From these results it was decided to concentrate on the `take()` function.

We note that a useful metric for assessing performance in this project is the number of samples per second which before EPICURE optimisation was equal to 2.66.

2.5.5 Code optimisation and results

Optimisation of the take function

The `take()` function in NumPy extracts a set of elements pointed to by a list of indices along a given axis in a NumPy array. The statements containing the `take()` function in `process_data()` are shown below:

```
cdata=np.take(np.take(data_t,lat_ran, -2),lon_ran,-1)
normalizer=np.take(np.take(normalizer, lon_ran, -1), lat_ran, -2)
```

In each case the inner take call first extracts the elements as pointed to by the list of indices contained in `lat_ran` and returns a copy of the temporary/intermediate array. Creating a copy of the array, especially a higher dimensional array is an expensive operation. The outer `take()` then acts on this temporary result (array) and again returns a copy of the new result. Thus, two copies are being generated. Our aim was to improve efficiency by reducing the number of copies being generated. To do this we eliminated the `take()` function by using NumPy indexing as shown below:

```
cdata=data_t[…,lat_ran[:,np.newaxis],lon_ran]
normalizer=normalizer[…,lat_ran[:,np.newaxis],lon_ran]
```

The experiments were rerun using no workers and 4 workers as shown in Table 4.

Program Version	Number of Data Loaders	Samples/sec	Speed-up for (Samples/sec)	Total time/ min	(Time) Speed-up / Gain %
Original	0	0.35	1	80	1 / -
Optimised	0	1.36	3.88	36	2.22X / 55%
Original	4	2.66	1	24	1
Optimised	4	3.88	1.45	13	1.84X / 46%

Table 4 Speedup in terms of samples/second and runtime for the original and optimised version with np.take() replaced. Results are shown for no workers (0) and 4 workers.

As can be seen we get good speedups by eliminating np.take(). It is worth noting also that with the optimised version the number of samples/second is fairly stable whereas with the unoptimized code this can be very variable.

Optimisation of Zarr reading

A second optimisation involves the reading of the Zarr file, a format widely used for multidimensional data. The line_profiler of Python showed that reading the data array from the compressed Zarr file on the GPFS (General Parallel File System) is a bottleneck. It contributes to 87.6% of the total time, i.e. 1868 seconds of a total of 2133 seconds (with no num_workers / data_loaders) in the process data function. Our next aim was to optimize the oindex (orthogonal indexing) function of Zarr which was used for reading from the GPFS. This was done by using dask, which allows Python objects to be distributed over parallel resources, to replace the function calls for the reads. The relevant code sections implementing the dask calls are given below:

```
import dask.array as da
# in __init__
self.dask_array_data=da.from_zarr(self.ds['data'])
```

```
self.dask_array_data_sfc=da.from_zarr(self.ds['data'_sfc])
# in __iter__
data_tt_sfc = self.dask_array_sfc[idxs_t].compute()
data_tt = self.dask_array_data[idxs_t].compute()
```

The first lines convert Zarr files to distributed dask objects in parallel. The second lines coalesce the distributed objects onto the master node. Using Dask arrays, the statement taking 1868 seconds now takes only 495 seconds i.e. a **73%** decrease in the run-time of the statement. With the Dask arrays optimisation, the original total run time of **21** minutes reduces to **14.5** minutes i.e. a total time reduction of approx. **30%** (NumPy + Dask).

Fixing and optimizing for torch.compile

The optimisation efforts focused on three key areas:

- PyTorch compilation
- Data pipelines

and keeping tensors on GPU by removing redundant cpu().detach() calls.

PyTorch compilation was enhanced by enabling the fused optimizer (AdamW(..., fused=True)) and configuring the inductor compiler with:

```
torch._inductor.config.cpp_wrapper = True
torch._inductor.config.triton.cudagraphs = True
```

Data pipeline improvements included increasing worker counts from 8 to 32 and preventing RAM usage spikes during worker initialization through staggered starts time.sleep(worker_info.id). The model was compiled using `torch.compile(model_ddp, mode="max-autotune", dynamic=True)`. These changes resulted in significant performance improvements, achieving a steady-state throughput of 58.48 samples/second with high consistency as seen in Figure 5. This represents an approximately 15x improvement over the previous implementation's best performance of 3.88 samples/second and a ~2.5x improvement over the mixed precision baseline of 23.7 samples/second.

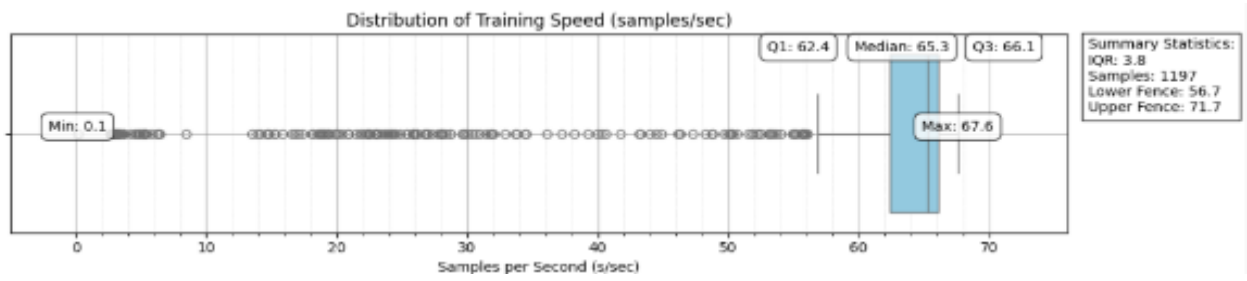


Figure 5 Measured training speed after optimisation (project 9).

However, the optimisations led to some resource imbalance, with GPU 3 showing higher utilization (>90% memory, 25% computational load) compared to GPUs 0-2 (~60% memory, 12% computational load) due to hosting two encoder-decoder transformers on the same GPU – GPU 3.

2.5.6 Improvements achieved

The optimisations resulted in a 15x performance of the previous model' baseline and ca. 2.5x speedup for the mixed precision version. We also obtained an increase in the number of samples per second from 2.66 to 66, representing a speed up of about 30x.

2.5.7 Summary and Lessons learned

In this project we observed that simple Python changes can make large improvements in performance, while the introduction of an optimised package such as dask can accelerate the reading of large data files.

Lessons we learnt from the optimisation activity include:

1. Linaro DDT does not work with the multiprocessing package of Python. When running with either srun or Python, the debugger remains stuck after a few steps. `cProfile` only produces the profile for the main thread and not the workers created for Data Loading by the main thread. `GraphViz` does not display the time taken by the various parts of the application but only the percentage. The call path visualisation in `SnakeViz` is difficult to follow and is unintuitive.
2. AtmoRep runs one process per 4 GPUs and not a single process per GPU. PyTorch recommends that there should be a single instance of `DistributedDataParallel` (**DDP**) per process per GPU and informs that the older method of using multiple GPUs with `DistributedParallel` (**DP**) yields sub-optimal performance. Our efforts to run one process per GPU in AtmoRep did not work as possibly the various fields are destined (hard-coded) to go to certain GPUs and that there is no `DistributedSampler` for an `IterableDataSet` object.
3. The Python line_`profiler` records the timings only when the number of data loaders was set to 0.
4. An attempt was made to split the data among multiple data-workers for a multi-node run and a new function was written for the same. This does not work as the internal logic of the program does not support data-splitting among workers from different nodes. This is justified as the amount of data is so large that workers from different nodes are simply given some different data from the dataset. This is different from traditional HPC where the aim is to divide the total data among several processes.
5. Following the `PyTorch` tuning guide, we tried setting gradients that did not change to none by setting the parameter set_to_none to True when zeroing out the gradients `Optimizer.zero_grad(set_to_none=True)` but this did not lead to any time reductions.

2.6 Project 11: Large Eddy Simulation of Supersonic Air Intakes

2.6.1 Project overview

In this project a High-order Compact Compressible Code (IC3) solver was used to solve the compressible Navier-Stokes equations on unstructured meshes with a finite-volume approach. Time integration was performed using explicit Runge-Kutta schemes and featured a solution-adaptive methodology to handle discontinuities. The spatial numerical scheme combined a non-dissipative centred part and an essentially non-oscillatory second-order shock-capturing scheme that was applied in regions of shock waves, identified by a shock sensor. Variables are located at cell centroids and fluxes computed at the interfaces of control volumes.

2.6.2 Code description

The IC3 code is proprietary and based on C++ and MPI. The code uses ParMETIS for domain decomposition and consists of 329000 lines of source which are mostly written in C++. We note that unfortunately, because of license restrictions, the source and any subsequent modifications to it cannot be included in this report.

2.6.3 Optimisation objectives

One performance bottleneck was based on the observation that on increasing the number of cores allocated in a single node, the performance worsens by a factor of 2 or 3 depending on the architecture (Intel, AMD). The aim of the enabling was to identify the reasons of this behaviour and then propose some solutions to be implemented in the solver. Given the code size, the work concentrated on the data structures and the computationally intensive routines. In particular, the PI asked us to focus on vectorization and correct memory usage by exploiting data locality.

2.6.4 Status before optimisation

The IC3 solver before optimisation demonstrated very poor intranode scaling on CPU cores. Before profiling the IC3 solver, we investigated the influence of compiling with various combinations of optimisation flags on Meluxina, but no significant differences in elapsed time were observed. The code was then profiled with Linaro-map, Extrae, Score-P and Maqao. The profiling revealed that the main bottleneck was due to indirect memory access leading to only **47%** of vectorized instructions (see Figure 6). In fact, the Cells and Faces traversal routines create numerous cache misses.

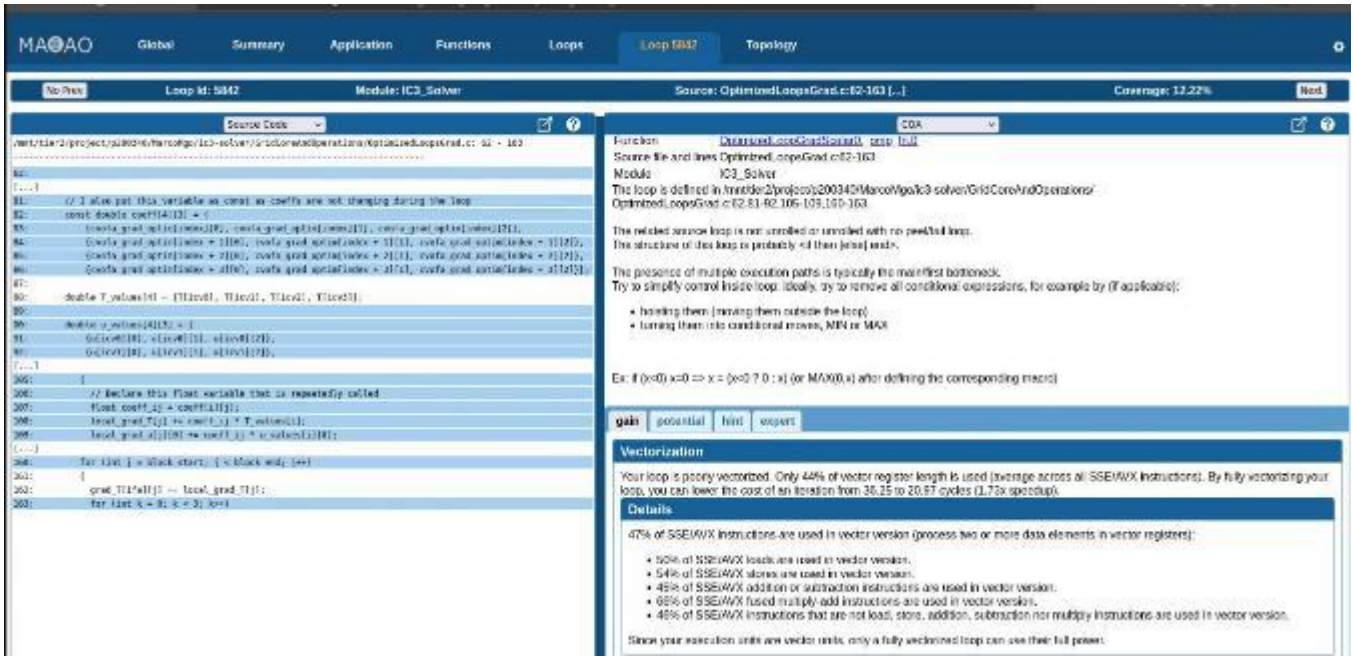


Figure 6 MAQOA profiling show 47% of vectorized instructions

We then did a performance benchmark to test internode scaling. As can be seen in Figure 7, the parallel scaling is very good, with around 90% parallel efficiency at around 8196 cores.

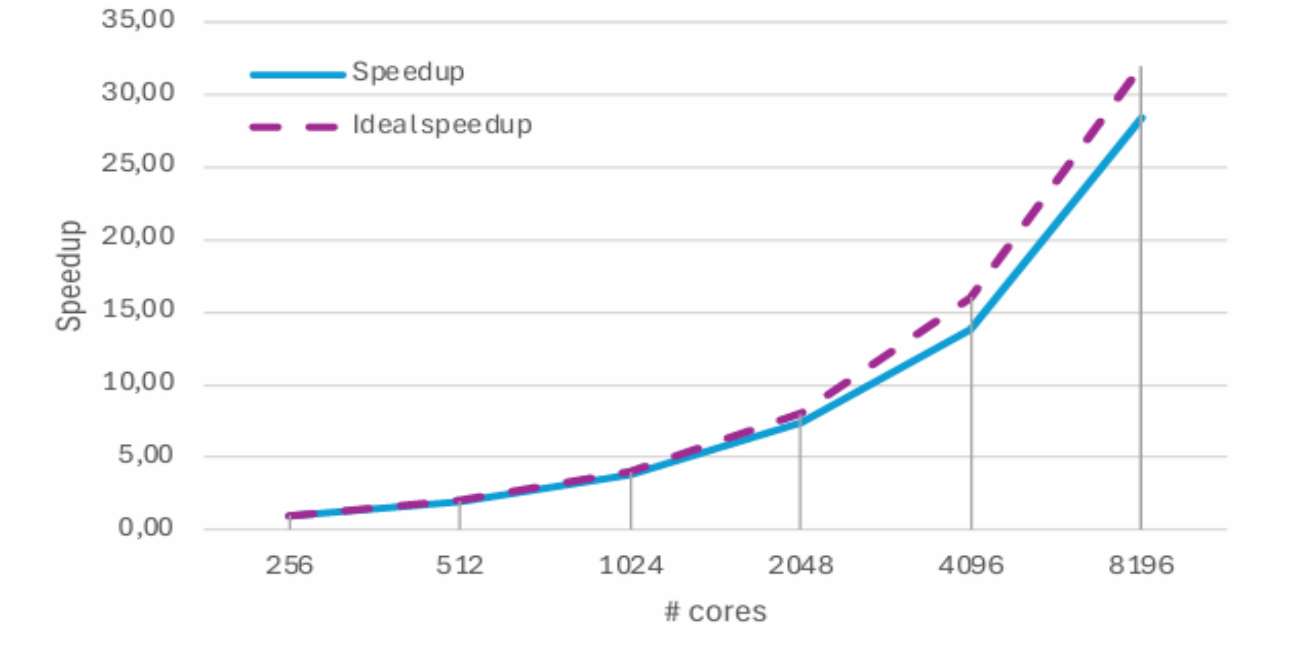


Figure 7 Speed up for a problem size of 512x512x512 on Meluxina

2.6.5 Code optimisation and results

MPI profiling was performed to ensure that no communication/synchronisation issue was present. Measurements were collected using Score-P and Vampir was used as the trace visualizer as can be seen in Figure 8.

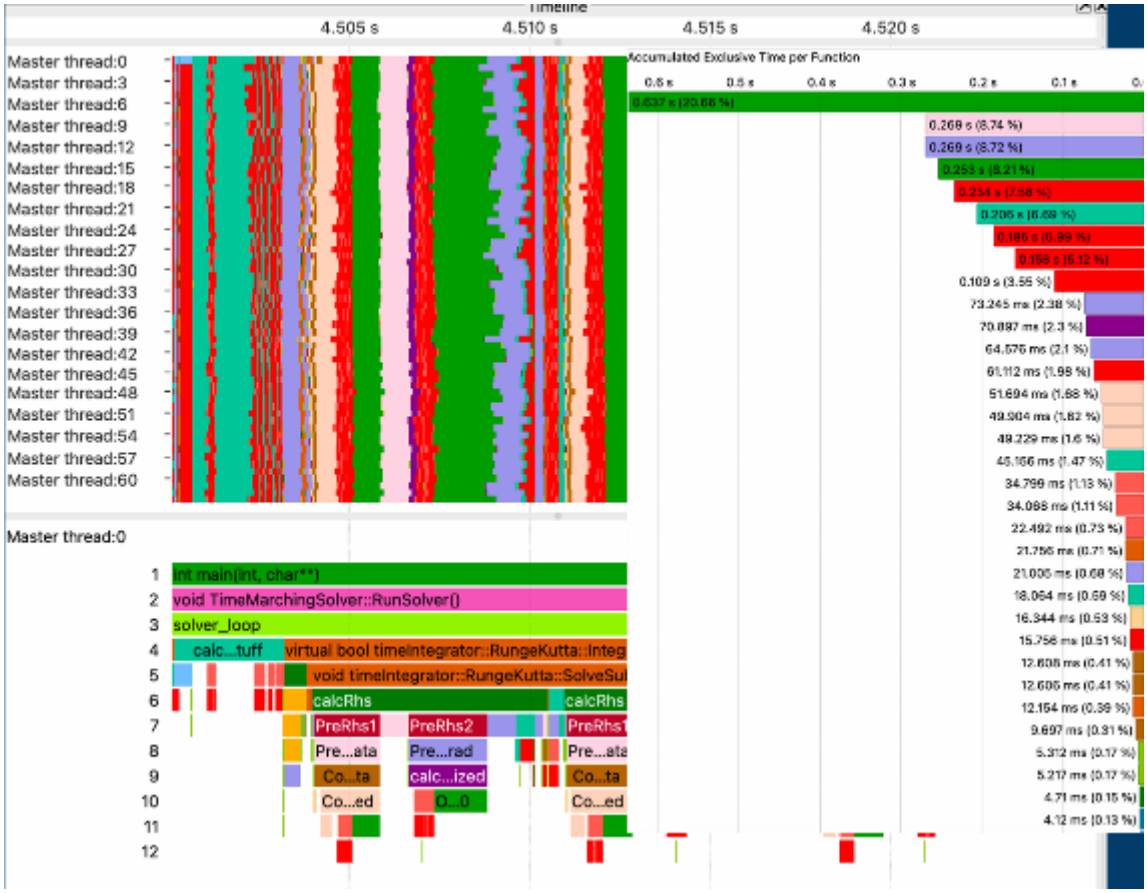


Figure 8 Vampir trace visualizer

A “Late Sender Wrong Order” communication pattern (see Figure 9) was indicated by the profiler with the recommendation to replace the source parameters in MPI_Irecv by MPI_ANY_SOURCE – this information was reported to the PI and the developers of the IC3 code.

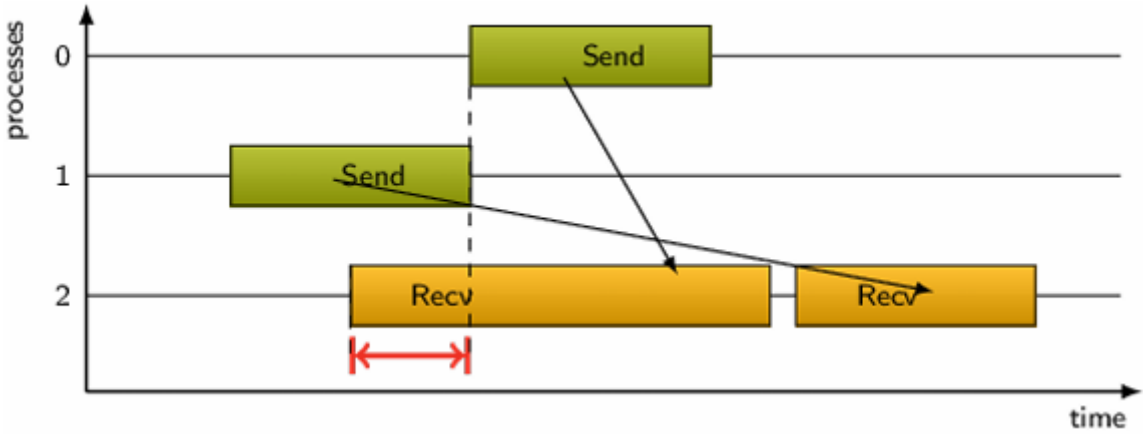


Figure 9 Illustration of an MPI efficiency problem called “Late Sender Wrong Order”. This occurs when pairs of MPI SEND – RECEIVE calls are incorrectly ordered.

Following the profiling process, which revealed significant cache misses, we investigated the use of process affinity within nodes to optimize performance. Process affinity allows binding specific processes (or threads) to particular cores or sets of cores, enhancing cache utilization. Processes running on the same core(s) and within the same NUMA domain can benefit from shared cached data, reducing latency and improving efficiency. To implement this, we explored binding options using SLURM, OpenMP (details on OpenMP implementation are provided below), and the numactl tool.

The following options have been tested for SLURM:

a. Threads binding: srun –ntasks-per-node=8 –cpu-bind=threads …
b. Cores binding: srun –ntasks-per-node=8 –cpu-bind=cores …
c. Sockets binding: srun –ntasks-per-node=8 –cpu-bind=sockets …

Binding tasks to the 2 logical threads of the CPU cores gives a slightly better performance than other options.

The following options have been tested for numactl tool:

a. Default option for always having the allocation on the current node: namctl -l …
b. Option for allocation interleaved on all CPUs: numactl –interleave=all
c. Option for explicitly choosing node or memory: numactl –membind=0,1 –cpubind=0

The first option produced a better result, but it was already the default behaviour.

By leveraging these binding strategies, we aimed to minimize cache misses and improve overall code performance. The findings suggest that careful configuration of process affinity can enhance resource utilization but overall, this did not give an expected significant leap in performance.

The following optimisations were attempted:

1.OpenMP Implementation: We have investigated hybrid OpenMPI+OpenMP parallelization for the solver, but we have not seen any substantial improvements. After a second profiling, we have noticed that the indirect array indexing causes false sharing and reduces the percentage of vectorized code (see Figure 10).

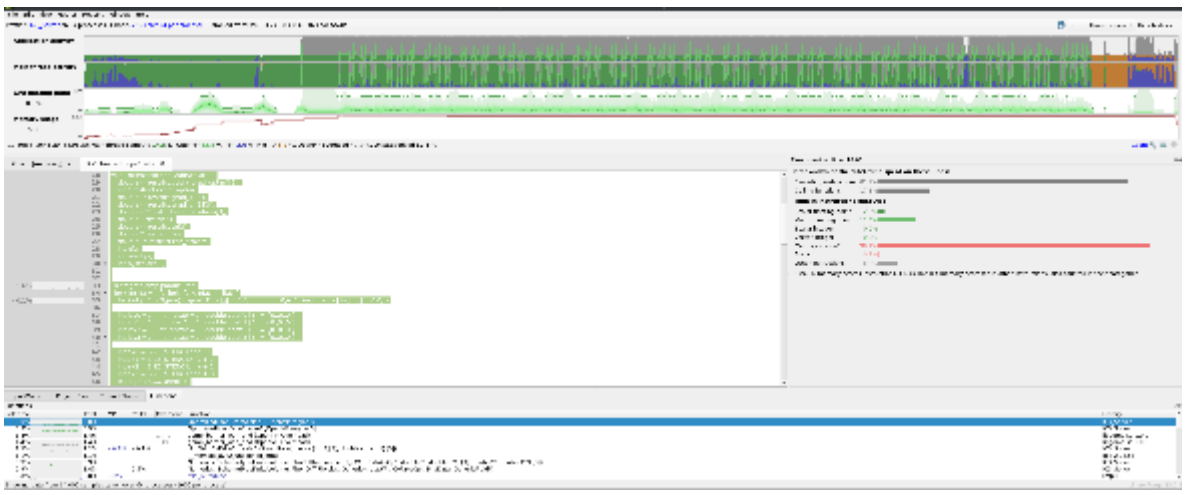


Figure 10 Map profiling of the hybrid OpenMPI+OpenMP code

2.Vectorization: Automatic vectorization has not been conclusive due to indirect memory access patterns preventing the compiler to vectorize the time-consuming function. We therefore manually vectorized the bottleneck functions using specific intrinsics such as the non-contiguous load instructions **mm256_i32gather_pd** to mitigate the problem (see Figure 11).

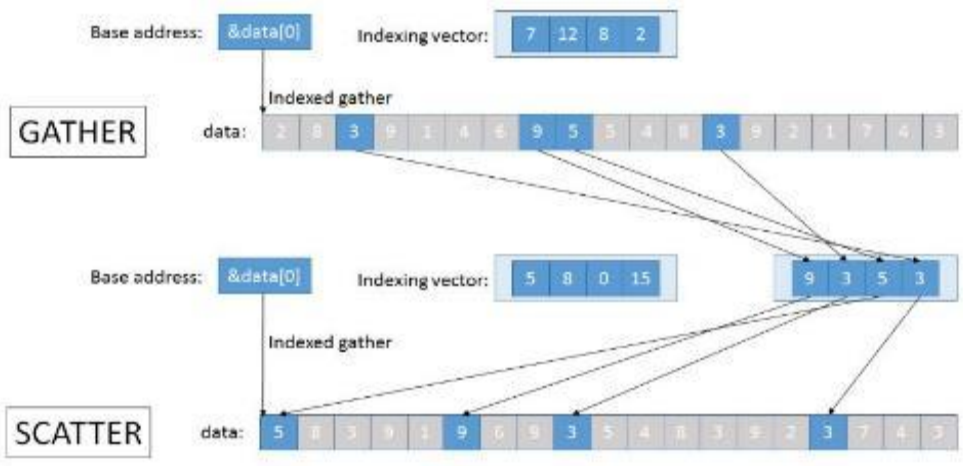


Figure 11 Intrinsics behaviour for non-continuous loads

This approach helps us to increase CPU floating-point and thus the arithmetic intensity. In fact, we find that CPU floating-point usage more than doubled going from 22.3% to 47.7% as shown in Figure 12.

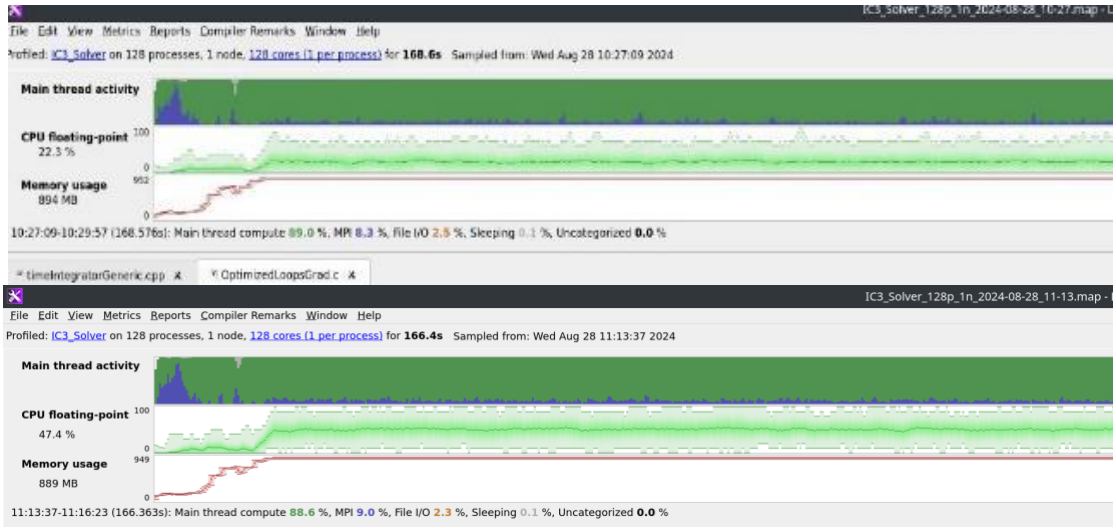


Figure 12 CPU floating-points operations between original (top) and improved (bottom) codes

2.6.6 Improvements achieved

Unfortunately, in this project it was not possible to make significant performance improvements despite the efforts of the support team.

The main reason is related to the original algorithmic design and the data locality. When iterating from one cell to the next, the index of the latter was in certain cases drastically different of the previous one and the associated degrees of freedom are so “distant” in memory that cache misses are almost unavoidable. Only very significant code changes which were outside of the support scope could lead to more visible improvements.

2.6.7 Summary and lessons learned

Unfortunately, in this project it was not possible to make significant performance improvements, despite the efforts of the support team. The main reason for this is of algorithmic nature and comes from how cells are numbered: when iterating from one cell to the next, the index of the latter is in certain cases drastically different of the previous one and the associated degrees of freedom are so “distant” in memory that cache misses are unavoidable. In this project, we faced difficulties when optimizing large codes, where some specific algorithms are difficult to improve due to their dependencies on objects coming from libraries on which the software depends on.

The lessons learned include:

- Optimizing code can be quite difficult on such large software especially when the code has seen a lot of developers working on it.
- The data locality issue is intrinsic to this kind of application:
  - Indexing cells or faces is crucial
  - Can we change it without breaking other parts of the code?
- There is a real need to adopt a data-oriented approach for such an application

2.7 Project 13: Enhancing AI of EU institutional LLMs for the public sector

2.7.1 Project overview

The European Commission's DG Translation (DGT) has proposed to create large language models (LLMs) for generative artificial intelligence (AI) that are more adapted and controllable for use cases at the European Institutions and more broadly in the European public sector, safer and more respectful of European values, and with better coverage of all EU official languages. The project aimed to leverage the European Institutions' own high-quality and multilingual internal data. The training technique proposed was the continued pre-training of existing opensource large language models, with the potential outlook for training a sovereign European LLM from scratch.

2.7.2 Code description

The codes used include machine learning frameworks such as Megatron-LM, Colossal-AI, FairScale, DeepSpeed, Hugging Face Accelerate, and PyTorch. The software was packaged into a container for use with singularity on the Leonardo Booster. We note that during the support the PI moved from the Llama model to the Mixtral one, so we shall report the results only for the latter.

Features of the latest training, implemented with **Colossal AI** and **Booster** to integrate training with parallel setup, include:

- The use of **Mixtral-8x7B-v0.1**, a sparse mixture-of-experts network, where at every layer, for every token, a router network chooses two among 8 groups of parameters (the “experts”) to process the token and combine their output additively. This way the model only uses a fraction of the total set of parameters per token: in detail, only 12.9B parameters per token are used despite Mixtral has 46.7B total parameters. Input processing and output generation is thus expected to have the same speed and cost as a 12.9B model.
- **MoeHybridParallelPlugin**, which combines data parallel (ddp/zero), pipeline (model sharding) and expert parallelism.
- **Adam optimisation** via HybridAdamOptimizer
- **Zero Redundancy Optimizer** (only stage 1 available), an optimizer which partitions optimizer states across processes.
- **Mixed precision**
- **Lazy initialization** plugin, to save memory with large models at initialization stage

2.7.3 Optimisation objectives

The main aim of the optimisation was to scale the LLM mode efficiently over multiple nodes of the Leonardo Booster. In addition, refining memory utilization and profiling was performed to locate any other optimisations to improve efficiency.

2.7.4 Status before optimisation

The PI intended to execute the training on Leonardo in a singularity container using the Pytorch Distributed communication package to interact with the SLURM job scheduler. Strong scaling benchmarks on Leonardo revealed that this initial configuration could not use more than one GPU per node. By adding the correct number of processes per node in the pytorch launcher, slurm parameters in the jobscript and the definition of the global rank in the training script, we enabled multi-GPU and multi-node distribution. In this setup, srun was used to launch one singularity instance per node, and each container implements distributed training on 4 pytorch processes mapped to Leonardo's GPUs. The code has further been integrated with the JUBE benchmark environment in order to facilitate benchmarking.

2.7.5 Code optimisation and results

Initial tests were focused on assessing the minimum memory requirement, to overcome the “CUDA out of memory” error, initially encountered regardless of the number of GPUs used and the parallelization implemented. The model uses Adam (Adaptive moment Estimation) optimizer, to dynamically adjust the learning rate for each individual parameter, rather than using a single global

learning rate. It accelerates convergence by reusing the past gradient information and is more robust in case of sparsity (datasets where features occur less frequently). However, as reported (https://huggingface.co/mistralai/Mixtral-8x7B-v0.1/discussions/43), while the model is expected to require about 90 GB in half precision, with Adam optimizer the minimum memory needed is about 350 GB, that is at least two Leonardo's nodes. To enable the model against the high memory footprint, two modifications have been crucial:

- Pipeline set to 2 minimum, just to load the model on GPUs (this already occupied about 50 GB of the GPU memory)
- Enabling CPU offload, as not the being the default; this modification leverages the memory footprint on the GPU, with a limited cost in terms of FLOPs during the optimisation stage.
- Reducing the "max length" parameter to 4096.

Without CPU offload, one epoch has been successfully executed on not less than 4 nodes with 16 pp processes, with a peak memory of about 55 GB, as reported by torchrun utilities. Further analysis to understand the high memory costs is ongoing.

Considering that the model by design requires multi-node runs, we initially focused our performance analysis on the use of Leonardo's network.

AI training using distributed pytorch are known to implement NCCL as a backend for communication, which is considered the most efficient library for collectives on NVIDIA GPU cards. As being implemented to support inter-GPU communications on different platforms, such as Clouds and HPC clusters, this library automatically selects the network, either Infiniband or socket, according to its availability.

The PI reported initially random failures of NCCL, with messages like the one below:

```
NCCL error in: ../torch/csrc/distributed/c10d/ProcessGroupNCCL.cpprsion
2.20.5 ncclInternalError: Internal check failed
```

Activating runtime debug information with NCCL_DEBUG=INFO showed that socket network was used instead of Infiniband, and enforcing IB over socket with the following environment variables:

```
export NCCL_NET=IB

export NCCL_SOCKET_IFNAME=ib0,ib1,ib2,ib3

export NCCL_IB_HCA=mlx5_0,mlx5_1,mlx5_2,mlx5_3
```

was not successful, because the Infiniband plugin was not found by the library. To this, we further had to build the singularity container on the cluster's /dev/ and /etc/ directories, by adding the options

```
-B /dev/infiniband,/etc/libibverbs.d.
```

With these modifications the PI reported a speedup of about 10%, which is a reasonable gain considering that in model parallelism the main bottleneck is the presence of dependencies, which prevent hiding communication costs between devices with computation on GPUs.

After analysing the use of network, we focused on the memory footprint in order to identify a setup without CPU offload and improve the average GPU utilization. The minimum memory requirement is computed with the formula weights*pr + gradients*pr + optimizer_states*2*pr = 16 * model_weights = 752 GB, considering that optimisation is done at 32-bit precision. This corresponds to 3 Leonardo nodes and does not take into account additional memory required for temporary buffers, memory fragmentations and activations. By considering a batchsize of 32, zero stage 1 and activating pipeline parallelism, which is used to distribute layers in shards across devices, we have been able to run on 4 nodes without CPU offload only with a minimum number of 16 pipeline processes, thus without data parallel. This corresponds to about 17 GB of memory per GPU for the booster initialization only, and a peak memory of 54 GB. By scaling the application over nodes, the number of processes for pipeline can be decreased in favour of data parallelism (dp_size = world_size / pp_size) up to 16 nodes, where the minimum number of pipeline process can be set to 4, thus ensuring that the more intense communication triggered by pipeline parallelism rely on NVLink rather than using Infiniband. With this node number, we achieve a gain of about 20% in terms of second per iteration, and consequently in terms of FLOPs, at the cost of an increase in the peak memory on the GPU of 10 GB (due to optimisation stage on the device), which is affordable in this setup.

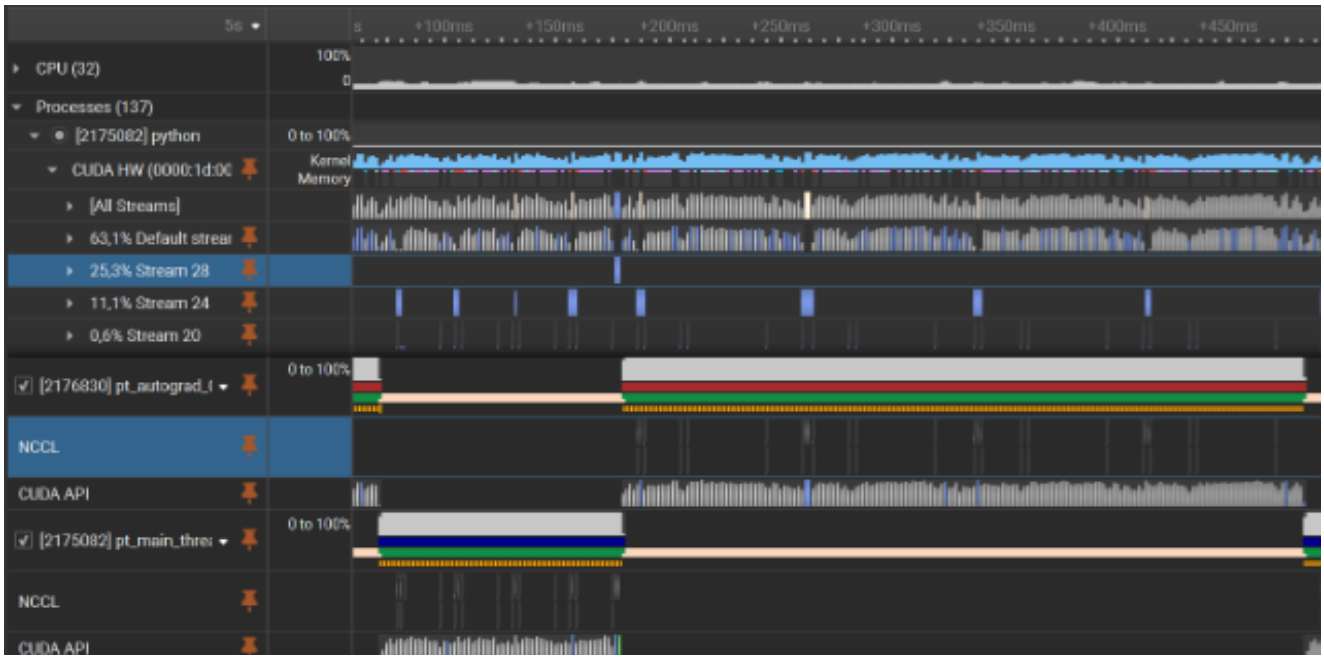


Figure 13 GPU events in the forward-backward pass as described in the text (project 13)

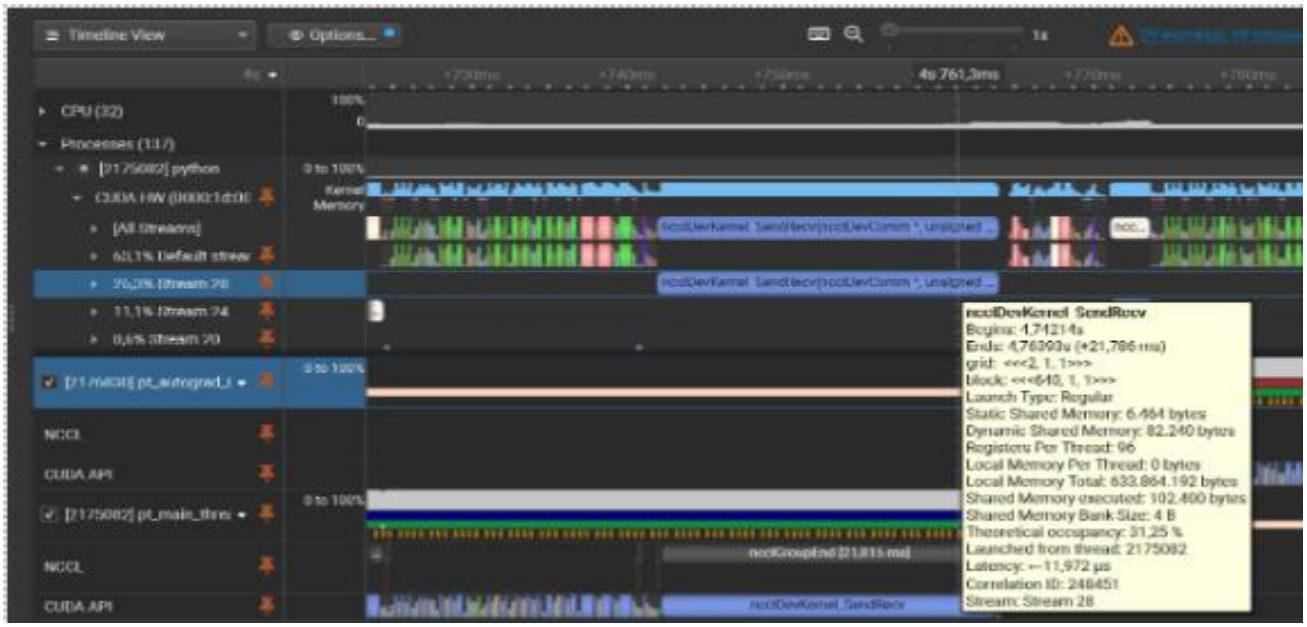


Figure 14 Zoom of the Figure 13 trace showing the sendrecv call blocking the synchronisation (project 13)

The figures above (Figure 13 and Figure 14) show GPU events in a range of the forward/backward pass, where the default CUDA streams execute computational kernels, and other streams NCCL perform send/recv and Allreduce calls. Despite being on different streams, NCCL and communications are synchronous with this parallelism. For example, the second picture shows in particular a blocking SendRecv operation triggered by pipeline.

By using the nsys stats utility to trace only NVTX ranges for a full run with pipeline 4 and expert 2 over 16 nodes, we can retrieve the average time of the optimisation passes (forward, learning rate compute, loss compute, optimisation), left plot, and overall duration of the NCCL APIs, right plot. In Figure 15 we compare the most relevant events for the configuration with CPU offload (red) and without (blue). Clearly, optimisation on GPU if feasible reduces the time to solution for the optimizer step. The high total time in ncclGroupEnd indicates a high number of synchronizations (the ones triggered by pipeline parallelism to exchange the microbatch, as the one in the trace above).

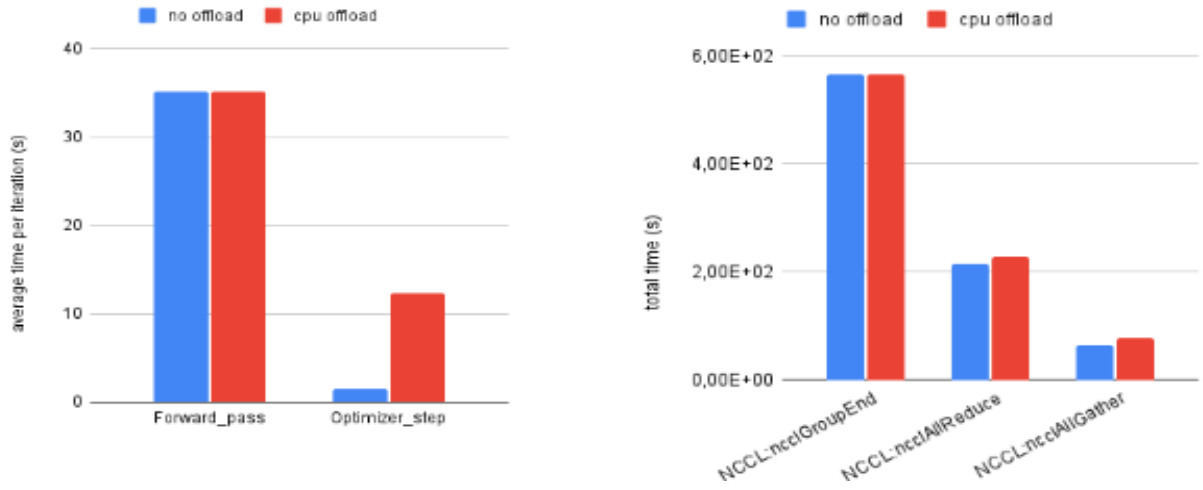


Figure 15 Times for the relevant events with (red) and without CPU offload (blue) (project 13)

How much improvement can be achieved by tailoring the number of pipeline processes with respect to the world_size has not been addressed yet.

2.7.6 Improvements achieved

The training was **enabled to use all the 4 GPUs** on Leonardo nodes when integrating the singularity container with the SLURM job scheduler and identified the parallel configuration to load the model and avoid CPU offload for the production dataset. We set NCCL to use Infiniband with a **speedup of 10% with** respect to default TCP/Socket usage, and we scaled this configuration up to 16 nodes, where the minimum number of pipeline processes could be lowered to 4, achieving an additional **speedup of 20%.**

2.7.7 Summary and lessons learned

In this project we have demonstrated the pitfalls associated in using containers in a multi-node environment on HPC systems. In particular, the fact that by default the most efficient network on the system is not detected. The work is ongoing, and more results will be described in future reports and deliverables.

2.8 Project 16: DEAREL: Large-scale Deep rEinforcement leARning for activE fLow control in wings

2.8.1 Project overview

The DEAREL project combined high-performance computational fluid dynamics (CFD) with deep reinforcement learning to optimize airflow control around wings. Running on Barcelona's MareNostrum5 supercomputer, the project integrated a Fortran-based CFD solver running on GPUs with a Python-based machine learning model. Through the SmartSim library and Redis database system, multiple parallel CFD simulations communicate with the reinforcement learning model to optimize flow control parameters. The project aimed to improve both the communication efficiency between the CFD and machine learning components, as well as enhance the scalability of the incompressible flow solver to handle thousands of GPUs simultaneously.

2.8.2 Code description

DEAREL represents an integration of computational fluid dynamics (**CFD**) and machine learning, combining a Fortran-based CFD solver (**SOD2D**) with a Python-based reinforcement learning (**RL**) model implemented in TensorFlow. The system leverages both GPU acceleration and MPI parallelization for the CFD component, while communication between components is managed through SmartSim's Redis in-memory database system, enabled by the **SmartSOD2D** wrapper. The RL model periodically updates boundary conditions based on flow states, and training efficiency is achieved through parallel execution of multiple CFD simulations until convergence is reached.

2.8.3 Optimisation objectives

The main objectives of the optimisation activity included:

- Reduce communication overhead to allow for better internode scaling
- Scale the solver from 100 to 1000 GPUs

2.8.4 Status before optimisation

NVIDIA Nsight Systems was used to generate a detailed trace of the application. This trace proved instrumental in identifying performance bottlenecks, pinpointing synchronization issues, and repeated operations in the application.

2.8.5 Code optimisation and results

Kernel optimisation

In this section we show how the most time-consuming kernels in the application were optimized.

Preparation for optimisation

In order to ensure the correctness of our optimisations we collaborated with the developers to add a set of probes in the simulation, using which we can compare our runs against baseline runs. In Figure 16 we show the locations of the probes in the simulation.

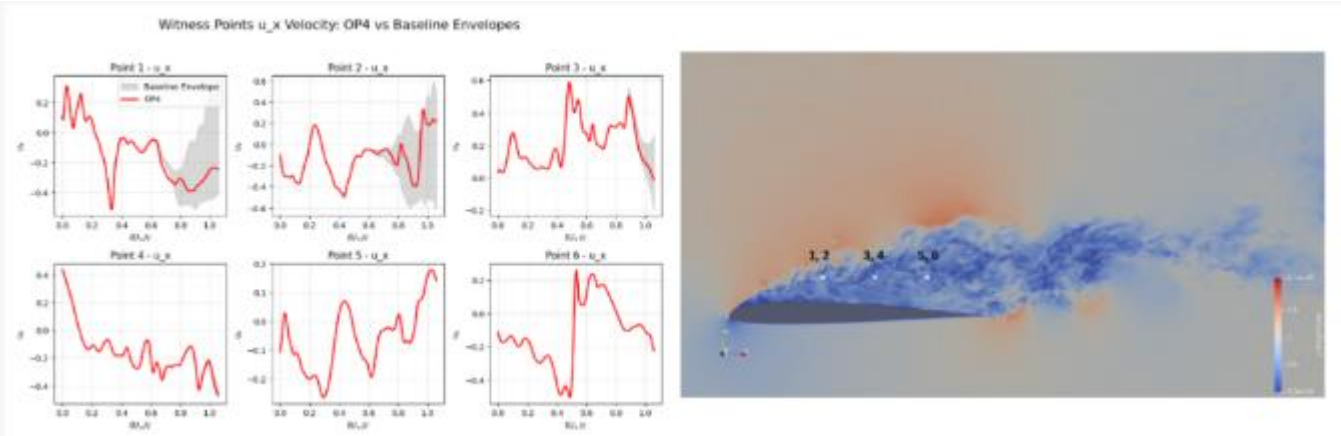


Figure 16 Location of the probes in the simulation (project 16)

Optimisation of visc_dissipationRate

Analysis of this kernel revealed several inefficiencies in the OpenACC FORTRAN code implementation:

- Stride memory access patterns
- Indirect array accesses
- Suboptimal usage of the LeviCivi tensor

The indirect memory access pattern was particularly problematic given its impact on cache utilization. Through analysis of the application runtime behaviour, we determined that the LeviCivi tensor was quite sparse, making its array-based implementation inefficient. The optimisation strategy involved:

- Introduction of an auxiliary array to eliminate indirect addressing
- Restructuring array index ordering to remove strided access patterns
- Replacing LeviCivi array references with their direct {-1, 0, 1} based on index permutation

These optimisations reduced the kernel runtime from *140.08* ms to *72.75* ms, achieving a **1.93** x speedup.

Optimisation of save_hdf5_restartfile

This kernel exhibited significant performance improvement through careful restructuring of its parallel execution pattern. The original implementation used the OpenACC kernels directive around nested loops. Through performance analysis, we identified this as a major bottleneck and replaced it with the parallel construct, resulting in dramatically improved performance from *542.27 ms* to *370.53 µs* - a **1463.22x** speedup.

eval_laplacian_diag

Multiple optimisation opportunities were identified in this kernel through performance analysis:

- Removal of unnecessary loop collapse directives (particularly ineffective given ndim=3)
- Optimisation of memory stride patterns
- Register-level optimisation through loop merging
- Cache utilization improvements through data reuse
- Reduction of atomic operations to increase parallelism

Implementation of these optimisations reduced runtime from *30.51* ms to *3.41* ms, achieving an **8.95x** speedup.

full_diffusion_ijk_incomp

Analysis of this kernel suggested loop fusion as the primary optimisation opportunity. However, implementation showed only modest gains, with runtime improving from *1.21* ms to *1.18* ms (**1.03x** speedup).

smart_visc_spectral_incomp

Performance analysis revealed several optimisation opportunities:

- Replacement of sequential loop execution with vectorized parallel implementation
- Global memory access reduction through data locality optimisation
- Strategic use of shared memory for frequently accessed data
- Optimisation of mue_l parameter handling

These changes reduced kernel runtime from *2.63* ms to *785.41* µs, providing a **3.35x** speedup.

eval_gradient

Loop structure optimisations focused on:

- Branch reduction through loop reordering
- Improved register utilization

- Removal of collapse directives to enable loop unrolling
- Increasing data locality

These modifications improved performance from *701.12 µs* to *486.08 µs*, achieving a **1.44x** speedup.

full_convec_ijk_incomp

Optimisation strategies included:

- Loop fusion for improved efficiency
- Elimination of redundant private variables
- Removal of suboptimal collapse directives
- Utilization of shared memory for ql and ul variables

These changes reduced runtime from *2.53* ms to *1.40* ms, providing a **1.81x** speedup.

The final results of the optimisations of the code can be summarized in **Error! Reference source not found.**.

Kernel Name	Time Before	Time After	Speedup
visc_dissipationRate	140.08 ms	72.75 ms	× 1.93
save_hdf5_restartfile	542.27 ms	370.53µs	× 1463
eval_laplacian_diag	30.51 ms	3.41 ms	× 8.95
full_diffuision_ijk_incomp	1.21 ms	1.18 ms	× 1.03
smart_visc_spectral_incomp	2.63 ms	785.41 µs	× 3.35
eval_gradient	701.12 µs	486.08 µs	× 1.44
full_convec_ijk_incomp	2.53 ms	1.40 ms	× 1.81

Table 5 Optimisations for each kernel in the application

2.8.6 Improvements achieved

We validated the correctness of our changes by adding simulation probes and comparing runs of the optimized code against the baseline, maintaining numerical correctness. We focused on the most time-consuming kernels, removing bottlenecks in memory access and parallelization, which allowed us to achieve speedups ranging from 1.03x up to 1463x (for instance, 1.93x for `visc_dissipationRate`, 8.95x for `eval_laplacian_diag`, and 1463x for `save_hdf5_restartfile`). We do not report an overall speedup for the whole application because these measurements were taken mid-optimisation, but the kernel-level gains clearly indicate substantial performance improvements.

2.8.7 Summary and lessons learned

Lesson learned include:

- **Eliminate Indirect Memory Access**: Replace indirect addressing with auxiliary arrays, when possible, reorder array indices to remove stride access patterns, and look for sparse data

structures that can be replaced with direct value assignments if the nonzero patterns are predictable.

- **Optimize Parallel Execution**: Identify and replace basic directives around nested loops with more advanced parallel constructs if analysis shows significant bottlenecks; this can lead to dramatic speedups.
- **Refine Loop Structures**: Remove unnecessary collapse directives, merge loops for register-level optimisation, reduce atomic operations to increase parallelism, and ensure data is reused effectively to leverage cache locality.
- **Evaluate Loop Fusion Carefully**: While it may be an obvious optimisation strategy, loop fusion does not always guarantee large performance gains, so weigh the complexity of implementation against real-world improvements.
- **Vectorize and Enhance Data Locality**: Convert sequential loops to vectorized parallel implementations whenever feasible, minimize global memory traffic through data locality optimisation, and consider using shared memory for frequently accessed or updated data.
- **Streamline Branching and Unrolling**: Reorder loops to reduce branches, remove directives that prevent compiler optimisations like loop unrolling, and focus on register utilization to boost performance.
- **Use Shared Memory and Remove Redundancies**: In scenarios with repeated data accesses, leverage shared memory to cut down on global memory usage. Eliminate needless private variables, remove ineffective directives, and combine loops where beneficial to streamline calculations.

3. Conclusions

In this first optimisation report we have selected 7 projects where there has been significant application enabling for level 3 optimisations. Given that this report has been compiled for the first year of the EPICURE project, which required a few months to put operational procedures in place, we anticipate that future documents will describe a higher number of optimisation projects.

As requested in a review of the first version of this document we have emphasised the improvements achieved by each project both in the project descriptions and the summary in Table 6**Error! Reference source not found.**. We can see that for some projects significant performance optimisations were obtained for many application codes and code kernels. For other projects, it was not possible to gain performance improvements, but the activity was able to resolve bugs or refactor the code to facilitate future optimisations. In all the projects, the EPICURE ASTs learned valuable experience on working with the latest tools and technologies in HPC.

EPICURE Project no.	Maximum performance gain	Improved parallel scaling	Code improvements (incl. fixing errors and code refactoring)
2	Overall 2x (but higher on some sections)		X
5	20x		
8			X
9	15x		
13	1.2x	X	
16	Up to 9x for some kernels, >1400x for I/O		

Table 6 Summary of improvements resulting from EPICURE optimisation activities in 2024

As a final comment we emphasise the importance of the close collaboration of the ASTs with the researchers from the projects which has on occasion resulted in the optimised code being pushed directly into the git repository of the code developers. On other occasions lack of PI cooperation has

resulted in a less successful optimisation activity. Fortunately, such cases are rare and in general the AST-project team interactions have been successful, resulting in a more efficient use of EuroHPC resources.