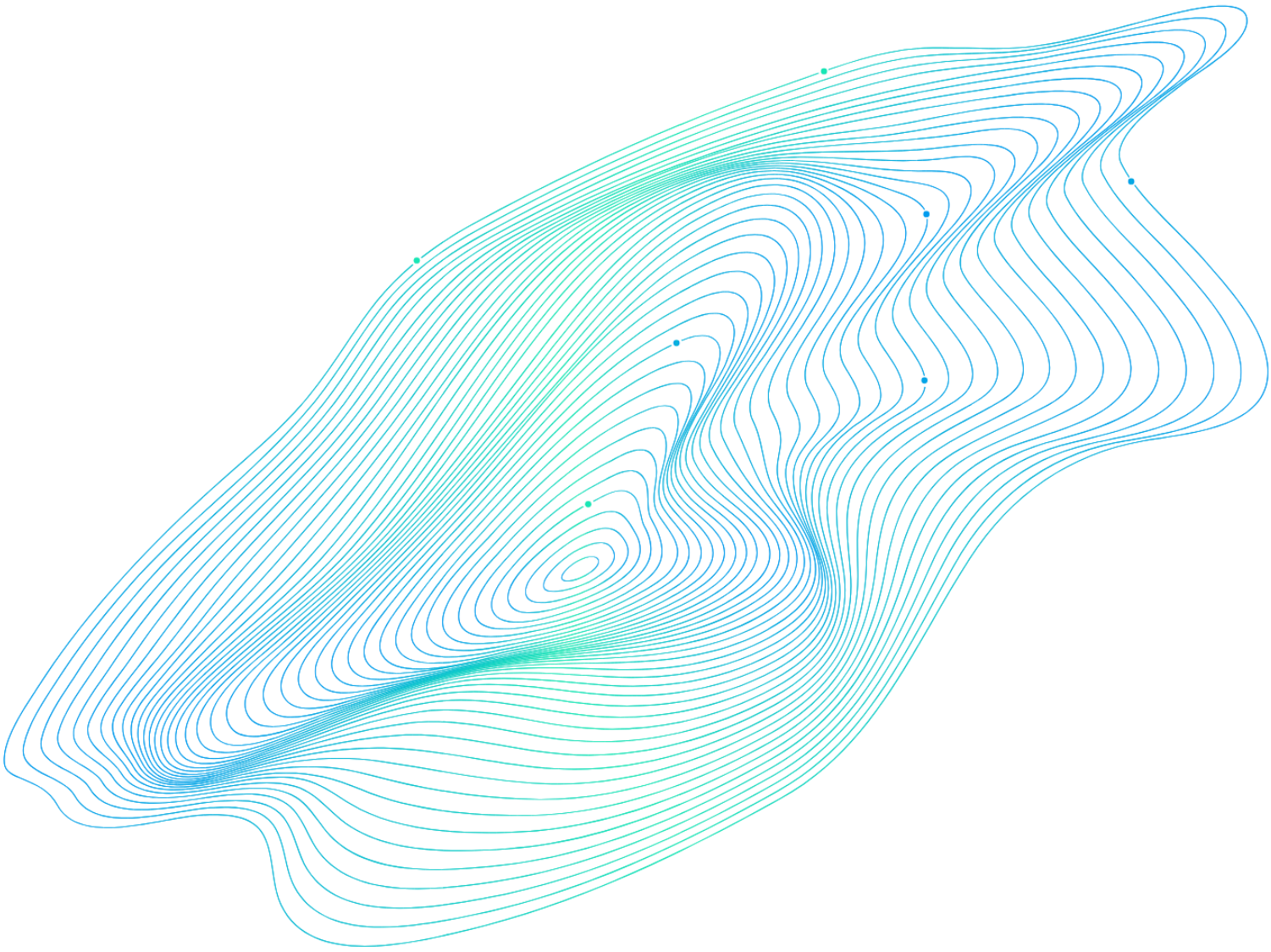
EPICURE

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 3.2

Report on code optimisation for Year 2



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	WP3
Task	3.1,3.2,3.3
Lead Authors	Andrew Emerson (CINECA)
Contributors	G. Cortiada (BSC), G. C. Rovira (BSC), G. Saxena (BSC), A. Paliouras (BSC), S. Bna' (CINECA), M. Celoria (CINECA), G. Frumenzio (CINECA), T. Gorni (CINECA), D. Amblard, L. Déjoua (CINES), G. Gil (CINES) E. Salaun (CINES), E. Vitali (CSC), A. Dorobisz (CYFRONET), B. Sobol (CYFRONET), A. Oliveira (INESC), A. Sousa (INESC), P.Larsson (KTH), F. Bongiovanni (LUXPROVIDE), E. Kieffer (LUXPROVIDE), F. Pinel (LUXPROVIDE)
Peer Reviewers	Eric Boyer (GENCI), Mikael Djurfeldt (KTH)
Version	1.0
Due Date	31/01/2026
Submission Date	29/01/2026

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)

Co-funded by the European Union
EuroHPC Joint Undertaking

Version History

Revision	Date	Editors	Comments
V0.1	10/12/2025	Andrew Emerson (CINECA)	Table of Contents
V0.2	31/12/2025	Andrew Emerson (CINECA)	First complete draft of the deliverable
V0.3	12/01/2026	Andrew Emerson (CINECA)	Review conducted by the appointed reviewers
V0.4	20/01/2026	Andrew Emerson (CINECA)	Reviewers input consolidated and second draft sent to the Project Management Board (PMB)
V0.5	22/01/2026	Ville Savolainen (CSC) Joao Barbosa (IT4I)	PMB Feedback on the deliverable received
V0.6	22/01/2026	Andrew Emerson (CINECA)	Project Management Board feedback consolidated and final version of the deliverable made available to the PMO
V1.0	29/01/2026	Virvatuli Uusimäki (CSC) Alberto Lanzanova (CSC)	Final quality check performed by the Project Management Office and submission of the deliverable

Co-funded by the European Union
EuroHPC Joint Undertaking

Glossary of Terms

Item	Description
AST	Application Support Team, EPICURE staff assigned to provide optimisation for the EuroHPC project
CADD	Computer Aided Drug Design, the use of computers to find and optimize new drugs
DDP	Distributed Data Parallel, a way of parallelising ML models by replicating the data over multiple nodes.
FSDP	Fully Sharded Data Parallel, another way of parallelising ML models but unlike DDP the model parameters, gradients, and optimizer states are divided amongst the available GPUs (“sharding”).
Dask	Python framework for distributed computing
ESM	EPICURE Support Manager, database created in WP4 and used by EPICURE staff to manage projects requesting support. Any benchmark or optimisation activities are also recorded by the tool.
JIT	Just In Time compiler, used in interpreted languages such as Python to compile code portions at runtime.
Lead	The EPICURE partner in charge of coordinating the optimisation support team (i.e. the AST) for the EuroHPC project.
PI	In EPICURE the PI (i.e. Principal Investigator) is the person leading the EuroHPC project team and represents the main point of contact.

Co-funded by the European Union
EuroHPC Joint Undertaking

Executive Summary

This is the second code optimisation report of EPICURE and describes the advanced application enabling provided to EuroHPC JU projects who requested this support in 2025. In this document optimisation activities are described for 13 projects, 5 more than in the preceding report for 2024, with both academic and industrial users, and covering a range of scientific, engineering and computer science fields. In all the projects detailed in this document the improvements performed by the EPICURE support teams have been emphasised. These improvements often involved increases in performance but also included porting to new architectures, better parallel scaling and reduction in memory requirements, which allowed new test cases to be studied. In even those cases where such enhancements were not possible, the application code was usually modernized and refactored to facilitate future possible optimisations. The overall result is that EPICURE optimisation support has had a significant impact for many researchers working in EuroHPC JU-funded projects.

Co-funded by the European Union
EuroHPC Joint Undertaking

Table of Contents

1. Introduction 9
1.1 How the optimisation reports were prepared 9
2. Optimisation Reports 10
2.1 Summary of projects for code optimisation 10
2.2 Project: 32 CAPHARD: Codon-Assisted Phage-Host Automatic inteRaction Discovery (Lead: CINECA) 11
2.3 Project 34: Unsupervised artificial intelligence models for Down syndrome neuroimaging research: from automatic brain alteration detection to the generation of synthetic 3D magnetic resonance images (Lead: BSC) 15
2.4 Project 50: Machine Learning of Exchange Energy Density and Integration with Python-based Electronic Structure Codes (Lead: LuxProvide, Collaborators: UAntwerpen) 20
2.5 Project 57: Quantum Monte Carlo generation of machine learning potentials for hydrogen-based high-temperature superconductors (Lead: CINECA) 24
2.6 Project 89: Predictive Exploration of Correlation in Specific Interactions and Optimisation Networks (PRECISION) (Lead: CINECA) 26
2.7 Project 99: Monte Carlo simulations of spin glasses for quantum-simulation experiments (Lead: CSC) 30
2.8 Project 107: BCPNNSim (Lead: CINES) 34
2.9 Project 108: GPU-enabling of explicit compressible OpenFOAM solvers for flow and acoustic prediction (Lead: CINECA) 38
2.10 Project 109: Exploiting GPU computational power to simulate a turbulent flow over a grid spaced 127 pin bundle using SPUMA (Lead: CINECA) 45
2.11 Project 113: Compilation and porting of the code THATMPI using the Cray compiler in LUMI (Lead: CYFRONET, Collaborators: KTH) 49
2.12 Project 128: Past Stall Operation of Hydrogen Multistage Radial Compressors (Lead: CINECA, Collaborators: IT4I@VSB) 55
2.13 Project 133: Scaling Deep Neural Networks for Public Safety: Fire, Flood etc. with HPC (Lead: BSC) 58
2.14 Project 137: Distributed LLMs on ARM (Lead: INESC TEC) 62
3. Conclusions 66

Co-funded by the European Union
EuroHPC Joint Undertaking

List of Figures

Figure 1 Nsight sys traces highlighting the ncclDevKernal_AllReduce_Sum functions used by the Allgather collective call, before optimisation (top) and after optimisation (bottom). (project 32) 14
Figure 2 Linaro-Map Automatic report (project 50) 21
Figure 3 Detailed profiling (project 50) 22
Figure 4 Avoiding data duplication among workers (project 50) 23
Figure 5 Function prototypes for the (old) cublas and the (new) cublas_v2 APIs. The ZGEMM operation (matrix-matrix multiplication of double precision complex numbers) was taken as example reference. Notice that in cublas_v2 the handle referring to the library state needs to be explicitly passed as a parameter, and that other changes in the function signatures occurred (alpha and beta passed by reference instead of by value, custom data types are used to define the matrix-matrix operation type). (project 57) 25
Figure 6 Scaling curve for the parallel Python program (project 89) 30
Figure 7 cProfile output of the original code (project 99) 32
Figure 8 cProfile output of the improved code (project 99) 33
Figure 9 Time spent in different HIP operations (project 107) 36
Figure 10 Kernels trace profiled with rocprofv3 (project 107) 36
Figure 11 GPU Strong Scaling Performance - Total Simulation Time (80M element mesh) (project 108) 41
Figure 12 GPU Strong Scaling Performance - Total Simulation Time (480M element mesh) (project 108) 42
Figure 13 Computational Performance - GPU Scaling vs CPU Baseline (project 108) 43
Figure 14 Speedup Comparison vs CPU Baseline (project 108) 44
Figure 15 pin bundle representation (project 109) 46
Figure 16 flow representation in the pin bundle (project 109) 47
Figure 17 Flow representation (with visible mesh) around a pin (project 109) 48
Figure 18 Performance from a test simulation on LUMI with32 CPUs and 1000 timesteps. The data includes the following results: total for the entire simulation, averaged for a single domain (one core), and averaged relative to the domain location (division of the computational grid into columns). (project 113) 52
Figure 19 Results of the compiler optimisation flags benchmark on LUMI involving a single full CPU-node and 100 timesteps, the best rates from 3 runs ("sim" stands for total simulation rate, "mover" and "depos" are the two most compute intensive parts of the simulation). (project 113) 52
Figure 20 Performance metrics output from a test simulation on LUMI involving 32 CPUs and 1000 timesteps, executed as 8 processes, 4 OpenMP-threads each. Now cores assigned to all domains spent the same amount of time in computing the mover phase. This results in a lower idle time in less occupied domains, which can be observed in time spent on "other" computations. 54
Figure 21 Incorrect passing of RANK argument instead of LOCAL_RANK argument in YOLOv8 codebase (project 133) 60
Figure 22 Scalability speedup when using multi-node multi-GPU setups (project 133) 61

Co-funded by the European Union
EuroHPC Joint Undertaking

1. Introduction

This document represents the second optimisation report of WP3 and covers year 2 of the EPICURE project (i.e. 2025). We recall that WP3 manages level 3 support which goes beyond porting or simple performance analysis (level 2), instead focusing on optimisations or application porting which require code modifications, often guided by a deep profiling analysis or tracing, to obtain the best performance possible on the target EuroHPC architecture. The structure of this document is as follows: we first explain how this report has been prepared using information from the EPICURE project management (ESM) tool. In the second Section we then give the reports for each project, compiled using a common template and highlighting improvements resulting from the optimisation effort. Finally, we make some observations from the projects studied and summarise the benefits to HPC users from the EPICURE WP3 activity in year 2.

Information on WP2 activities (Code Porting, Enabling and Scaling) for year 2 can be found in D2.2.

1.1 How the optimisation reports were prepared

To compile the reports, we used the EPICURE Support Manager (ESM) tool to list all those projects indicating WP3 activity in year 2. This draft selection was then made available to EPICURE partners who could suggest removing or adding projects to the list, according to the latest updates on optimisation progress. In fact, for some projects starting in 2025, it was decided to delay reporting until the year 3 deliverable to allow a more complete record of the activity. Other projects were added thanks to very recent successful optimisation results reported by the ASTs.

Once the list had been finalized, the leader of each AST was asked to complete a report based on a template, which was assembled into this final document. The fields used to describe the optimisation activities are like those used in D3.1 but more attention has been devoted to ensuring the same format for each project description as requested in the last review. Since another recommendation of the review was to highlight better the achievements of each optimisation, we have included two extra subsections, **Status before optimisation** and **Improvements achieved**. In addition, we include the Lead partner involved in the optimisation, and any other partners involved not belonging to the Lead organisation (the Team).

Co-funded by the European Union
EuroHPC Joint Undertaking

2. Optimisation Reports

2.1 Summary of projects for code optimisation

In Table 1 we give a summary table of the projects for which optimisation activities in WP3 have been carried out.

*Table 1 Summary table of the projects optimised in year 2 of the project. The * indicates the EPICURE project ID.*

ID* (Lead)	Collaborators	EuroHPC ID	Project Title	PI category (academic, industry, SME or Startup)	Code type	Host Partition	Optimisation status
32 (CINECA)	-	EHPC-AI-2024A02-037	CAPHARD: Codon-Assisted Phage-Host Automatic inteRaction Discovery	Startup	AI (LLM)	Leonardo Booster	Completed
34 (BSC)	-	EHPC-AI-2024A02-043	Unsupervised artificial intelligence models for Down syndrome neuroimaging research: from automatic brain alteration detection to the generation of synthetic 3D magnetic resonance images	Academic	AI (Gen)	MareNostrum 5	In progress
50 (LuxProvide)	UAntwerpen	EHPC-DEV-2024D07-076	Machine Learning of Exchange Energy Density and Integration with Python-based Electronic Structure Codes	Academic	AI	Meluxina	Completed
57 (CINECA)	-	EHPC-EXT-2024E01-064	Quantum Monte Carlo generation of machine learning potentials for hydrogen-based high-temperature superconductors	Academic	C and Python with MPI	Leonardo Booster	Completed
89 (CINECA)	-	EHPC-DEV-2024D11-037	Predictive Exploration of Correlation in Specific Interactions and Optimisation Networks (PRECISION)	Industry	Python with MPI	Leonardo DCGP	Completed
99 (CSC)	-	EHPC-REG-	Monte Carlo simulations of spin	Academic	Python with MPI	LUMI	In Progress

Co-funded by the European Union

EuroHPC Joint Undertaking

		2024R02-140	glasses for quantum-simulation experiments				
107 (CINES)	-	EHPC-DEV-2025D04-021	BCPNNSim	Academic	C++ with HIP	LUMI	Completed
108 (CINECA)	-	EHPC-DEV-2025D04-035	GPU-enabling of explicit compressible OpenFOAM solvers for flow and acoustic prediction	Academic	CFD (C++)	Leonardo	In Progress
109 (CINECA)	-	EHPC-DEV-2025D04-113	Exploiting GPU computational power to simulate a turbulent flow over a grid spaced 127 pin bundle using SPUMA	Government	CFD	Leonardo	In progress
113 (CYFRONET)	KTH	EHPC-DEV-2025D02-094	Compilation and porting of the code THATMPI using the Cray compiler in LUMI	Academic	Plasma Physics (PIC)	LUMI-C	Completed
128 (CINECA)	IT4I	EHPC-DEV-2025D05-006	Past Stall Operation of Hydrogen Multistage Radial Compressors	Industry + Academic	CFD	Leonardo	In progress
133 (BSC)	-	EHPC-BEN-2025B07-009	Exploring additional computational resources for Latvian genome reference project	Academic	AI	MareNostrum5	Completed
137 (INESC TEC)	-	EHPC-DEV-2025D07-067	Distributed LLMs on ARM	Academic	AI (LLM)	Deucalion	In Progress

2.2 Project: 32 CAPHARD: Codon-Assisted Phage-Host Automatic inteRaction Discovery (Lead: CINECA)

2.2.1 Project overview

The goal of the project was to deliver AI-driven, effective personalized bacteriophage therapy and combat antibiotic resistance. More specifically, CAPHARD (Codon-Assisted Phage-Host Automatic inteRaction Discovery) uses attention-based deep neural networks to predict, given the genome of a bacteria and the genome a of phage as inputs, the probability that the phage is likely to kill the bacteria.

Co-funded by the European Union
EuroHPC Joint Undertaking

2.2.2 HPC systems used in the project

The project targeted the Leonardo Booster (GPU) partition.

2.2.3 Code description

The algorithm learns the genome representations using a transformer architecture that includes both self-attention as well as cross-attention layers and performs a binary classification task. The code performs the training from scratch as fine-tuning existing models is not a viable option. It is parallelized using a Data Parallelism pattern and relies on PyTorch and Hugging Face Accelerate libraries.

2.2.4 Status before optimisation

The code is already ported on multi-GPUs using the PyTorch distributed library and the Hugging Face Accelerate library, using mixed precision. CAPHARD can scale well up to 64 nodes, using Infiniband network and RDMA.

2.2.5 Optimisation objectives

The optimisation objectives included the:

- assessment of the current state of the optimisation of the software.
- optimisation of the software both at the level of single node and at the level of multi-node (with a focus on GPUs).

2.2.6 Code optimisation

We started by profiling and measuring the GPU memory footprint as a function of the batch size, analysing how changing the batch size could boost performances by increasing the GPU memory utilization. Results are summarized in Table 2, where we have considered 1% of the total training dataset on 2 Leonardo Booster nodes.

Table 2 Time taken and memory required as a function of batch size

1% of the training dataset on 2 Leonardo Booster nodes		
Batch size	**Mem. reserved per GPU [MB]**	**Time taken for epoch [s]**
4	6738	80.25
8	8478	64.52
16	12389	55.97
32	20044	51.96

Co-funded by the European Union
EuroHPC Joint Undertaking

64	36811	50.17
96	54026	49.02

We also tried to increase the number of workers in the Dataloader, but the best result was achieved with the default value:

```
loader=DataLoader(data, batch_size=batch_size, shuffle=True, num_workers=num_workers)
```

In Table 3 we show the results where we vary the number of workers from 0 to 8 (1% of the total training dataset on 2 Leonardo booster nodes and batch size = 96).

Table 3 Time taken as a function of the number of workers.

1% of the training dataset on 2 Leonardo Booster nodes setting batch size = 96	
Number of workers	**Time taken for epoch [s]**
0	49.02
2	50.14
4	50.37
8	51.67

The Accelerate library uses blocking data transfers by default for its automatic device placement, which can result in potentially unwanted CUDA synchronizations. Therefore, to avoid these synchronizations, we tried memory pinning and non-blocking transfers using the following code:

```
accelerator_config["dataloader_config"] = DataLoaderConfiguration(non_blocking=True)
accelerator = Accelerator(**accelerator_config)
loader = DataLoader(data, batch_size=batch_size, shuffle=True, pin_memory=True)
```

Unfortunately, these modifications led to no significant improvements. We then considered possible optimisations in the communications. For example, we found that by reducing the number of allgather steps used for collecting metrics such as accuracy and loss, we could reduce the percentage communication time from 1.4% to less than 0.1 % of the total time.

We also tried different DDP Communication Hook:s which control how gradients are communicated across workers. In particular, we considered:

- *FP16 Compression Hook,* which compresses gradients by casting them to torch.float16, reducing communication overhead.
- *BF16 Compression Hook,* which is like using FP16, but uses torch.bfloat16, which can be more efficient on certain hardware.
- *PowerSGD Hook* - an advanced gradient compression algorithm that provides high compression rates and can accelerate bandwidth-bound distributed training.

For instance, we modified the source code as:

Co-funded by the European Union
EuroHPC Joint Undertaking

```
ddp_kwargs=DistributedDataParallelKwargs(comm_hook=DDPCommunicationHookType.POWER_SGD)
accelerator_config["kwargs_handlers"] = [ddp_kwargs]
accelerator = Accelerator(**accelerator_config)
```

and similarly for the other Hooks. Unfortunately, none of these were proven more effective than the vanilla `AllReduce` in `DistributedDataParallel`, all resulting in comparable time per epoch. We also considered Local SGD, a technique for distributed training where gradients are not synchronized every step.

```
with LocalSGD(accelerator=accelerator,model=model,local_sgd_steps=8, enabled=True) as local_sgd: # Then the training loop
```

By considering `local_sgd_steps=8`, we reduced the `ncclDevKernel_AllReduce_Sum`, from about 53% of the activities on the communication stream to about 29% (see Figure 1)



Figure 1 Nsight sys traces highlighting the ncclDevKernal_AllReduce_Sum functions used by the Allgather collective call, before optimisation (top) and after optimisation (bottom). (project 32)

Unfortunately, due to the overlap between computation and communication of the already well optimized Accelerate library, together with the fast communication provided by the NVLink, all the communication optimisations we tried did not result in significant speed-up nor improvements in the time per epoch.

The final optimisation considered was JIT-compiling the PyTorch model. Note that on the first few executions *torch.compile* took extra time to compile the model:

```
self.model = torch.compile(self.model)
```

However, starting from the second epoch, we observed a speed-up of around 10%.

Co-funded by the European Union
EuroHPC Joint Undertaking

The final optimisations suggested are reported in Table 4, using the full training dataset running on 64 Leonardo Booster nodes.

Table 4 Training times using the final set of optimisations

Using the full training dataset on 64 Leonardo booster nodes	
Batch size and optimisations	**Time per epoch [s]**
4	323
96	146
96 + torch.compile	130

2.2.7 Improvements achieved

Moving from a batch size=4 (corresponding to a memory footprint of ≈6.7 GB per GPU) to batch size=96 (corresponding to about 54 GB per GPU) leads to a speed-up of a factor 2 x on 64 compute nodes using the full dataset. JIT-compiling the PyTorch model leads to an additional speed-up of 10%.

2.2.8 Summary and lessons learned

In conclusion, the most effective improvement for the CAPHARD code is to fully exploit all the GPU available memory thus increasing the parallel computations of the mini batches. Increasing the number of workers in the data loader, using memory pinning and asynchronous data transfer, as well as optimisations in the communication stream have not led to significant improvements. JIT-compiling the PyTorch model can be very useful especially when the model needs many epochs to converge.

2.3 Project 34: Unsupervised artificial intelligence models for Down syndrome neuroimaging research: from automatic brain alteration detection to the generation of synthetic 3D magnetic resonance images (Lead: BSC)

2.3.1 Project overview

The project aimed to produce a generative AI pipeline for medical 3D magnetic resonance images of the brain. The project worked with data of Down Syndrome neuroimaging.

Co-funded by the European Union
EuroHPC Joint Undertaking

2.3.2 HPC systems used in the project

The project targeted the Accelerated partition of MareNostrum 5.

2.3.3 Code description

The project mainly consisted of two Python scripts utilizing the PyTorch and MONAI libraries. The latter is wrapped around PyTorch and specialized in medical data and machine learning algorithms. The researchers of the EuroHPC project provided us with custom written training loops, for the following models:

1. **generative.networks.nets.AutoencoderKL**: MONAI implementation of the Kullback-Leibler Variational Autoencoder. A type of model that learns to compress 3D volume inputs to a latent (lower dimensions) space and decode that latent space to recreate the original input.
2. **generative.networks.nets.DiffusionModelUNet**: MONAI implementation of a UNet based diffusion model. This model learns to efficiently reduce noise from image and if appropriately injected in the latent space stage of a Variational Autoencoder, it can assist in the creation of newly unforeseen 3D images.

2.3.4 Status before optimisation

The initial code provided was written and only worked for 1 node and 1 GPU. The researchers had little-to-no experience on how to scale for multiple GPUs either in a single node or for multiple nodes.

2.3.5 Optimisation objectives

The optimisation objectives were to:

1. Achieve optimal data loading times.
2. Enable Multi-Node Multi-GPU training.

2.3.6 Code optimisation

The changes and strategies applied to the two training scripts were similar and, in this section, they are described together.

Dataloading

Data preprocessing transformations

By adding the following line of code:

Co-funded by the European Union
EuroHPC Joint Undertaking

```
transforms.ToDeviced(keys=[“image”], device=device)
```

to our input data transformation pipeline, we could accelerate the per sample pipeline by a factor of 4, from 4-5 seconds to about 1 second.

By processing the transformation pipeline on the GPU side, we could avoid spawning extra parallel processes for data loading. This is advantageous because having multiple worker processes often leads to complicated errors, such as deadlocks or unexpected worker crashes, especially when they need to access the GPU's memory safely and efficiently. By setting the configuration parameter DATALOADER_WORKERS=0, the same main program that is performing the training calculations also takes full responsibility for loading the next batch of data whenever needed.

Data transforms were also separated for training and validation in order to try different combinations of parameters for the train loader routine. Another way to speed up CPU-side data loading is to set DataLoader.num_workers > 0 so worker processes prefetch and prepare batches in host memory ahead of the next iteration. However, that was not deemed as efficient as running the pipeline on the GPU for every item in a batch on every iteration of our training loop.

Distributed sampler and dataloading

When using the DDP training method, we found it necessary to use a torch.utils.data.distributed.DistributedSampler instead of the single process torch.utils.data.Sampler. This sampler class makes sure that each computer/process gets a unique, non-overlapping slice of the total data, ensuring the entire system works together efficiently to cover the whole dataset

Preparing and model wrapping for DDP

DDP requires that the class to be trained is wrapped and we had to make sure that data such as hyperparameters or model weights were available to all ranks or processes while other data, such as data batches, were available to the correct specific ranks.

AutoencoderKL & Discriminator (train_3dKLAE_clean_DDP.py)

In this part of the code, we initialized objects for the `AutoencoderKL`, `PatchDiscriminator`, `PerceptualLoss` and `PatchAdversarialLoss` classes and moved the corresponding rank’s processes to the GPU device as follows:

```
discriminator = torch.nn.SyncBatchNorm.convert_sync_batchnorm(discriminator)
# DDP: wrap models

model = DDP (model, device_ids=[local_rank], output_device=local_rank,
                                    find_unused_parameters=True)

discriminator = DDP(
discriminator,device_ids
=[local_rank],output_device=local_rank,find_unused_parameters=True
)
```


Co-funded by the European Union
EuroHPC Joint Undertaking

DiffusionUnet (train_3dLDM_clean_DDP.py)

We initialize the AutoencoderKL, which loads its corresponding pre-trained autoencoder, as well as the DiffusionModelUNet and DDPMScheduler. Each of these objects is then moved to the GPU associated with the corresponding process rank.

The scale factor is computed **only on rank 0**, using its first input batch, and is then broadcast to all other processes. This is necessary because each rank receives a *different* batch due to the DistributedSampler. Computing the scale factor independently on each rank would therefore yield inconsistent values. By calculating it once on rank 0 and broadcasting it, we ensure that all processes operate with the same, synchronized scale factor.

The following lines were used to wrap our DiffusionUnet model to prepare for DDP training:

```
unet = torch.nn.parallel.DistributedDataParallel(
        unet,
        device_ids=[local_rank],
        output_device=local_rank,
        find_unused_parameters=True)
```

2.3.7 Improvements achieved

AutoencoderKL & Discriminator (train_3dKLAE_clean_DDP.py)

The maximum per device batch size was found to be 2 samples, meaning that the effective batch size in DDP is 8*(total number of GPUs). Therefore

- 1 node: batch size = 2 * 1 node * 4 GPUs per node = 8 samples
- 2 nodes: batch size = 2 * 2 node * 4 GPUs per node = 16 samples
- 4 nodes: batch size = 2 * 4 node * 4 GPUs per node = 32 samples

The performance results obtained for the AutoencoderKL and Discriminator are shown in Table 5.

*Table 5 Training performance for AutoencoderKL and Discriminator as a function of multi-GPU nodes. * In this column the reduction in the number of updates in the weights compared to the baseline is indicated. For example, with 4 GPUs, 64 fewer updates are performed*

Nodes	GPUs	Batch Size	Epoch Time (minutes: seconds)	Speedup	Fewer Weight Updates*
1	1	1	7:44	1× (baseline)	–
1	4	2	2:30	3	8
2	8	2	1:15	6	16
4	16	2	0:50	9	32

Co-funded by the European Union
EuroHPC Joint Undertaking

8	32	2	0:28	15.5	64

DiffusionUnet (train_3dLDM_clean_DDP.py)

The maximum per device batch size was found to be 8 samples, meaning that the effective batch size in DDP will be 8*(total number of GPUs). Therefore, we have:

- 1 node: batch size = 8 * 1 node * 4 GPUs per node = 32 samples
- 2 nodes: batch size = 8 * 2 node * 4 GPUs per node = 64 samples
- 4 nodes: batch size = 8 * 4 node * 4 GPUs per node = 128 samples
- etc.

The results obtained for training with DiffusionUnet are summarised in Table 6:

*Table 6 Training performance for DiffusionUnet as a function of multi-GPU nodes. * In this column the reduction in the number of updates in the weights compared to the baseline is indicated. For example, with 4 GPUs, 32 fewer updates are performed.*

Nodes	GPUs	Batch Size	Epoch Time (minutes:seconds)	Speedup vs Baseline	Weight Update Reduction*
1	1	1	3:40	1 (baseline)	–
1	4	8	2:40	1.4	32
2	8	8	1:20	2.8	64
4	16	8	0:35	6.3	128
8	32	8	0:05	35–40	256

The results when using 8 nodes appears to be an outlier. This is probably because the dataset sample used for debugging fits exactly the available resources without causing any idle times.

2.3.8 Summary and lessons learned

We showed that scaling up the training of Variational Autoencoders and Unet Diffusion models can give very good performances The execution speed-ups achieved in both cases scaled linearly in respect to the resources (GPUs) used. We are confident that, given more time, we could have investigated more frameworks or model partitioning techniques. However, the speed-up achieved was sufficient, in our basic first approach.

Co-funded by the European Union
EuroHPC Joint Undertaking

2.4 Project 50: Machine Learning of Exchange Energy Density and Integration with Python-based Electronic Structure Codes (Lead: LuxProvide, Collaborators: UAntwerpen)

2.4.1 Project overview

Quantum chemistry plays a crucial role in predicting and understanding molecular behaviour at the atomic level, driving advances in fields such as materials science, drug discovery, and nanotechnology. The quest for computational chemistry models that achieve high-level accuracy while maintaining numerical efficiency continues to push the boundaries of mathematics, physics, chemistry, and computer science. A key aspect of this progress is the translation of formal theoretical frameworks into computationally efficient code. For this reason, EPICURE support was requested to enhance multiprocessing and other parallelization techniques within the existing implementation of the code developed by the researchers of the EuroHPC project. A particular focus was devoted to leveraging high-performance computational resources to manage memory-intensive evaluations and reduce their execution times. Porting to GPUs was also considered, but this was not achieved during the lifetime of the project.

2.4.2 HPC systems used in the project

The system used for this project was MeluXina.

2.4.3 Code description

The code was primarily written in Python, utilizing PyTorch for ML model development and JAX/XLA for tensor operations. The integration with PySCF ensures compatibility with existing quantum chemistry workflows. The opensource code is hosted in a github repository.

2.4.4 Status before optimisation

Before the EPICURE support, the energy density generator encountered a memory bottleneck due to the complexity of its high-dimensional tensor contractions, which demanded significant computational resources and memory allocation. We first reproduced the issues encountered by the PI teams and profiled the application to exactly locate the memory bottlenecks in the code. This was necessary before any optimisation process could be attempted.

Profiling was performed on the original code using Linaro-Map on a large memory node (4 TB). The automatic report (see Figure 2) provided by this profiling tool showed that the code uses a maximum memory far beyond what is possible on a standard CPU node (~ 1.8 TB).

Co-funded by the European Union
EuroHPC Joint Undertaking



Figure 2 Linaro-Map Automatic report (project 50)

Co-funded by the European Union
EuroHPC Joint Undertaking

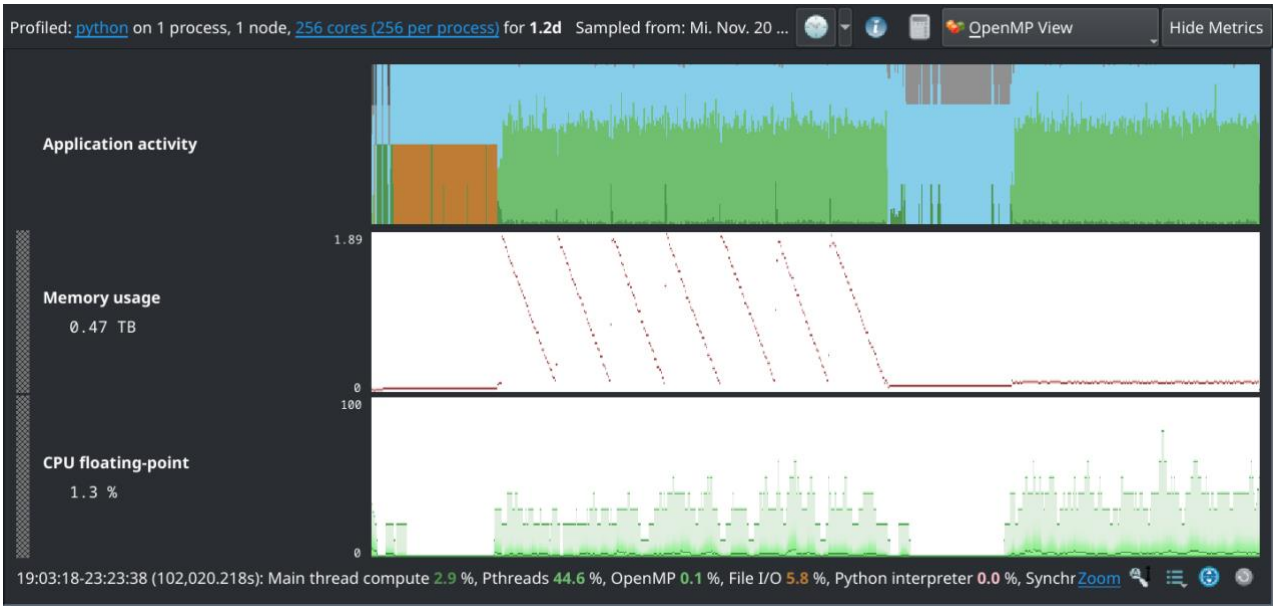


Figure 3 Detailed profiling (project 50)

We then profiled a smaller instance (see Figure 3) and identified that intermediate results are created for each thread (e.g.,the `occ_virt_basis` function is the same for each thread). The sawtooth pattern observed for the memory was also an indicator of poor memory management. To address this problem, support action was required to optimize memory usage through advanced memory management strategies and the implementation of efficient multiprocessing techniques.

2.4.5 Optimisation objectives

The optimisation objectives focused on allowing the code to handle larger chemical systems with more complex evaluations without exceeding memory limits. The functional evaluation component, which combines a sophisticated interplay between analytical libraries and the resolution of high-dimensional integral equations, also faces similar computational challenges. To overcome these, the implementation of parallelization techniques, including distributed computing and multiprocessing, was necessary to ensure that intermediate and nonlocal objects are processed more efficiently.

2.4.6 Code optimisation

The code optimisation in this project can be divided into two steps:

1. Distributed memory computing
2. Memory optimisation on a single node

Co-funded by the European Union
EuroHPC Joint Undertaking

Distributed memory computing:

Since this application is a Python code, we proposed to the PI teams to use the Dask framework – not only is `Dask.distributed` a lightweight library for distributed computing in Python but Dask can also be used in the future for porting to GPUs. We started by performing some refactoring of the code. First, the `ThreadPoolExecutor` was replaced by a Dask worker which solves a synchronization problem encountered by the PI team. Using Dask, we then scattered data to the Dask workers before any computations, hence avoiding the constant creation and deletion of data.

```
T = part_mp2_amplitude(self.mol,self.mo_coeff, self.mo_energies,
                       self.mo_occ,self.kappa)

# Handling partial MP2 doubles amplitude (a<->b)
# We need to copy here because of
T_transpose = np.transpose(T,(0,1,3,2)).copy()

# 1st Intermediate T_eval:
'0.5*T_{ijba}-T_{ijab}'
T_inter_ba = 0.5*T_transpose-T

# 2st Intermediate T_eval:
'0.5*T_{ijab}-T_{ijba}'
T_inter_ab = 0.5*T-T_transpose
```

Figure 4 Avoiding data duplication among workers (project 50)

We also decided to remove any data duplication among workers (see Figure 4). The large matrices `T_inter_ba` and `T_inter_ab` were clearly replicated by all workers. These matrices are read-only which allows us to use memory mapping to share the data among the workers from the same node.

Memory Optimisation on a single node:

We investigated and implemented several improvements to the single-node execution process. As a result of these efforts, the out-of-memory error was resolved across all tested scenarios, and the cores of the single node are now fully and correctly utilized.

Our optimisation involved removing several pieces of user-added code that were found to be redundant and inefficient. This included an outer-scope multithread pool, an associated lock mechanism, futures, and manual garbage collection triggers. This custom code was running concurrently with, and on top of, the built-in concurrency features already provided by the `opt_einsum.contract()` function. This redundant layering increased memory allocation significantly and negatively impacted performance, often leading to slower execution times or system crashes without the explicit lock in place. By removing this extraneous code, we now rely exclusively

Co-funded by the European Union
EuroHPC Joint Undertaking

on the built-in, optimized concurrency capability of the `opt_einsum` Python library, resulting in a more stable and efficient execution environment.

2.4.7 Improvements achieved

The performance improvements were tested using a benzene molecule simulation as the target use case. The original, unoptimized code required approximately 32 hours to complete and reached a peak memory usage of nearly 3.4 TB. By implementing our first optimisation strategy—which focused on both memory efficiency enhancements and scaling the execution across multiple nodes—we significantly reduced these metrics. Running the improved code on three nodes cut the total execution time down to nearly 11 hours and reduced the maximum memory peak to just 92 GB.

2.4.8 Summary and lessons learned

The collaboration with the PI team was key in this project. While the AST team was not expert in Quantum Chemistry, the different valuable exchanges with the PI team helped to make this project successful. The use of Dask makes distributed computing more accessible to non-expert users in HPC. Furthermore, the Principal Investigator (PI) team has indicated a future requirement to utilize GPUs; integrating this functionality should be straightforward using the Dask framework.

2.5 Project 57: Quantum Monte Carlo generation of machine learning potentials for hydrogen-based high-temperature superconductors (Lead: CINECA)

2.5.1 Project overview

The project aimed at producing high-quality electronic energies and nuclear forces datasets for hydrogen-based compounds by means of Quantum Monte Carlo (QMC) calculations. The datasets are planned to be used for generating machine learning potentials in a follow-up project.

2.5.2 HPC systems used in the project

The project was carried out on the Leonardo Booster partition.

2.5.3 Code description

TurboRVB is an open-source computational package for ab initio Quantum Monte Carlo (QMC) simulations of both molecular and bulk electronic systems.

It is written in Fortran and ported to GPUs via OpenMP offload plus an interface, written in C, to the NVIDIA cuBLAS and cuSOLVER libraries.

Co-funded by the European Union
EuroHPC Joint Undertaking

2.5.4 Status before optimisation

The WP3 activity was strictly intertwined with the WP2 one in this case. The code could not compile with the newer versions of the NVIDIA compilers (from nvhpc/23.11 on), which hampered the usage of the latest optimisations available for the cuBLAS and cuSOLVER libraries. The baseline was therefore considered as the TurboRVB code compiled with nvhpc/23.1.

2.5.5 Optimisation objectives

The main objective was to make the code compile and run with the newest NVIDIA toolkit available (nvhpc/24.3 at the time) which, from the WP3 point of view, would bring more efficient linear-algebra libraries and result in a shorter time-to-solution when running on NVIDIA GPUs.

2.5.6 Code optimisation

The C-interface to the cuBLAS libraries used by TurboRVB was updated to allow for the new cublas_v2 calls. Specifically, this required the update of the cuBLAS calls parameter list to comply with the new function prototypes (see Figure 5), and consequently the creation and management of the cuBLAS handles, i.e. pointers to the cuBLAS library contexts (cublas_v2 is a stateful library).

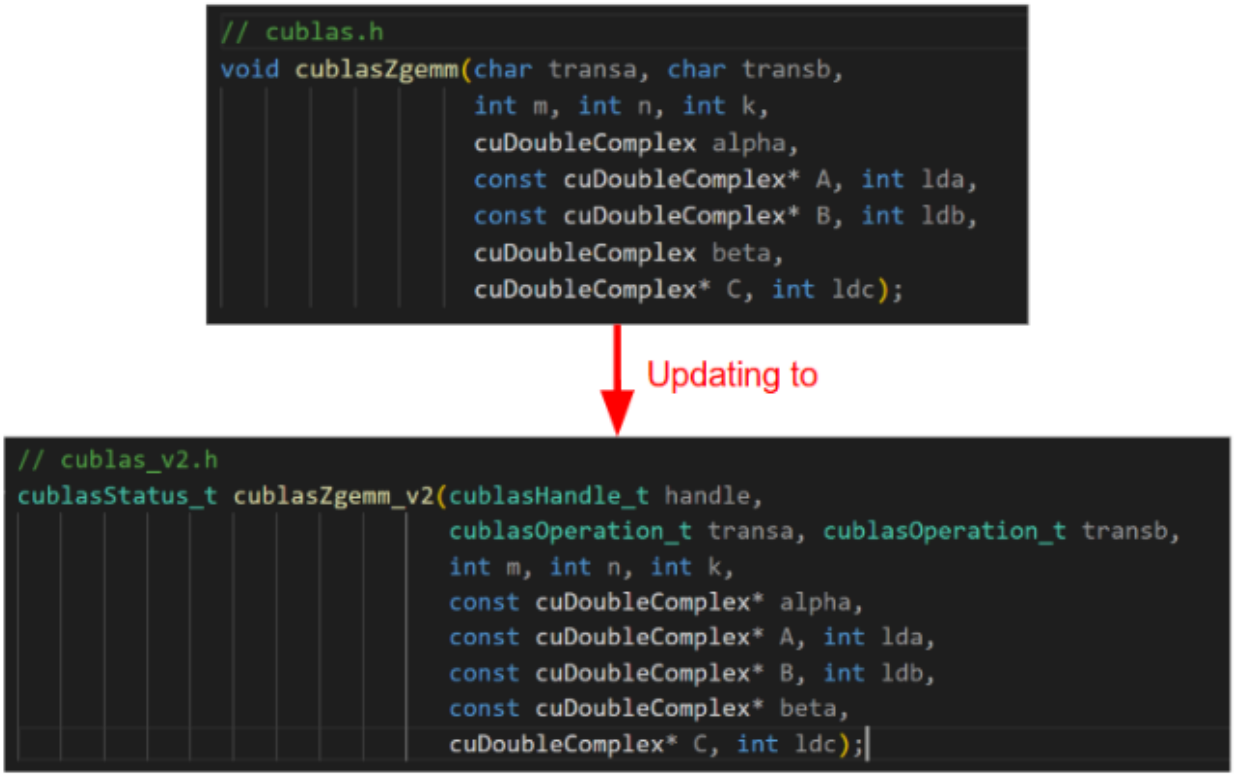


Figure 5 Function prototypes for the (old) cublas and the (new) cublas_v2 APIs. The ZGEMM operation (matrix-matrix multiplication of double precision complex numbers) was taken as example reference. Notice that in cublas_v2 the handle referring to the library state needs to be explicitly passed as a parameter, and that other changes in the function signatures occurred (alpha and beta passed by reference instead of by value, custom data types are used to define the matrix-matrix operation type). (project 57)

Co-funded by the European Union
EuroHPC Joint Undertaking

These changes allowed the code to compile also with the newer versions of the NVHPC toolkit and brought an overall speed-up to the time-to-solution.

2.5.7 Improvements achieved

The optimized version was benchmarked with a production-like workload consisting in 1 step of a Variational Monte Carlo simulations plus wavefunction optimisation. The benchmark was run on 64 nodes of Leonardo Booster (i.e. on 256 custom A100 NVIDIA cards). It must be noted that a full production run consists of N steps of the workflow above, hence the following percent improvement is representative for an end-to-end simulation. Moreover, QMC algorithms are (almost) embarrassingly parallel. Hence the following results do not depend significantly on the number of nodes used in the calculation. The results are reported in Table 7. Notice that the initialization time, which consists mostly of I/O operations, was subtracted from the total walltime to remove systematic errors due to disk access.

Table 7 Walltime comparison between the baseline (Old C interface) and the optimized version (New C interface), for a production-like run of TurboRVB on 64 nodes of the Leonardo Booster partition. The initialization time was subtracted to remove time due to I/O operations.

Code version	Software stack	Walltime w/o initialization [seconds]
Old C interface	nvhpc/23.1	158.84
New C interface	nvhpc/24.3	128.97

Enabling the usage of the newest NVIDIA libraries brought an overall 18-19% improvement in the time-to-solution.

2.5.8 Summary and lessons learned

The Quantum Monte Carlo TurboRVB code was adapted to support the newest versions of the NVIDIA compilers and libraries, which brought a speed-up of 18-19% with respect to the baseline.

2.6 Project 89: Predictive Exploration of Correlation in Specific Interactions and Optimisation Networks (PRECISION) (Lead: CINECA)

Co-funded by the European Union
EuroHPC Joint Undertaking

2.6.1 Project overview

In CADD, a potential drug molecule (or ligand) for a disease or condition can be identified by how well it binds to a protein involved in that condition. The extent of binding to the protein is called its "binding affinity" and when calculated for a collection of ligands can be used to find the best drug candidates in the list. One method to calculate protein-ligand binding affinities is by determining the free energy of binding which gives a measure of the stability of the protein-ligand complex when compared to the unbound state. The research group who applied for this project have developed a pipeline which analyses molecular dynamics trajectories to calculate binding free energies for protein-ligand complexes. They wished to use this pipeline on a database of over 4000 MD trajectories simulated with the GROMACS software (amounting to ca. 40TB of data). Since this dataset is very large, the processing pipeline must be parallelised and very efficient.

2.6.2 HPC systems used in the project

Since GPUs were not required, the HPC system targeted was the Leonardo DCGP (CPU) partition.

2.6.3 Code description

The Binding Free Energy Exscalate (BFEX) pipeline is designed to extract interaction fingerprints and energetic contributions from extensive MD simulations. The initial version of the pipeline consisted of a collection of Bash and Python scripts used to automate different analysis steps.

The analyses performed on the MD trajectories include:

- binding free energy calculations with the MMPBSA and MMGBSA methods[1]
- MMGBSA decomposition
- Solvent Accessible Surface Area (SASA)
- Distances between selected atoms or residues
- Hydrogen bond analysis

These analyses were performed using cpptraj, the analysis suite provided by the AMBER molecular dynamics package.

The workflow was originally organized in nested loops:

- A first loop processes the trajectory in windows of 10 frames, updating the cpptraj input files for MMGBSA, MMPBSA and distance calculations by modifying the starting and ending frames for each window. After updating the input files, the analyses are executed.
- A second loop identifies residues located within 10 Å of the ligand detected during the distance analysis. For each selected residue, the pipeline automatically updates cpptraj input

[1] MM/GBSA (Molecular Mechanics/Generalized Born Surface Area) and MM/PBSA (Molecular Mechanics/Poisson-Boltzmann Surface Area) are popular computational methods to estimate binding free energies for molecules (like drugs to proteins) by breaking down the energy into molecular mechanics (MM) and solvation terms.

Co-funded by the European Union
EuroHPC Joint Undertaking

files to compute hydrogen bonds, SASA and MMGBSA decomposition for the residue-ligand pair. As in the previous loop, after updating the files the analyses are executed.

Although functional, this structure involved a high number of repeated file updates and sequential operations.

2.6.4 Status before optimisation

The initial version of the BFEX pipeline was implemented through a combination of Python and Bash scripts built around *cpptraj*, which was used to perform all analyses required to extract interaction fingerprints and energetic contributions from MD trajectories.

Before the optimisation effort, the workflow suffered from several limitations:

- Fully serial execution: all analyses were executed sequentially. Each run processed only one analysis at a time, for a single 10-frame window and a single residue-ligand pair. Most *cpptraj* based analyses exploited only one CPU core, resulting in low single-node performance and poor scalability across large datasets.
- High computational cost: the most expensive task, MMPBSA and MMGBSA calculations. were the only components executed in parallel, as they relied on externally available MPI-enabled implementations. These analyses were therefore the *only* parts of the pipeline able to process multiple frames simultaneously using multiple processes.

No GPU support and limited vectorization, as all *cpptraj* based routines run on CPUs without any explicit optimisation for modern architectures

2.6.5 Optimisation objectives

The optimisation work focused on significantly improving the performance, scalability, and maintainability of the BFEX pipeline. The main objectives were:

- Performance enhancement: accelerate the analysis phase by enabling the execution of multiple analyses simultaneously, by exploiting the computational resources available on the HPC system.
- Parallelization: introduce an efficient parallel workflow capable of distributing independent analysis tasks across multiple CPU cores or nodes.
- Refactoring and code modernization: replace the original collection of Bash scripts with a unified, structured Python implementation. This change enables a cleaner code and simplifies the integration of parallelization strategies using widely available Python libraries.
- Portability and maintainability: improve code readability and portability across HPC environments, allowing the pipeline to be more easily extended with new functionalities.

Co-funded by the European Union
EuroHPC Joint Undertaking

Overall, the objective was to deliver an optimized and HPC-ready version of the BFEX baseline, capable of handling large-scale datasets within short time.

2.6.6 Code optimisation

The optimisation was implemented using the *mpi4py* Python library, enabling an efficient parallel distribution of the analysis workload across multiple CPU cores. The main strategies adopted are summarized below.

1. A NumPy array-of-arrays is first generated. Each sub-array contains a list of frame indices which defines a small trajectory segment (10 frames) to be analyzed independently.
2. MPI processes are created, and each process receives as input the full MD trajectory and one sub-array of frame indices. Each MPI rank extracts the corresponding frames and generates a reduced trajectory containing only the selected frames. Between one extracted segment of trajectory and the next, the workflow applies a 50-frame skip, reducing correlation and ensuring statistically meaningful sampling. All reduced trajectories are stored in a list for the subsequent analysis step.
3. A second MPI-distributed loop is applied to the list of trajectory segments. Each MPI process receives one reduced trajectory as input and performs the full set of analyses and writes the results independently.

This approach enables full task parallelism, as each trajectory segment represents an independent analysis unit. Each MPI process runs on a single CPU core, considering that each reduced trajectory contains only 10 frames. Each node of the LEONARDO DCPG partition has 112 CPU cores, therefore using a single node it is possible to analyze 112 trajectory segments simultaneously. On average, the complete analysis of each small trajectory takes approximately 5 minutes on a single core, enabling substantial reductions in total walltime.

All the required packages (NumPy, Pandas, MDAnalysis, pytraj, mpi4py) were installed using mamba. Since several analyses are based on using AMBER software, ambertools22 was also installed via mamba to reduce the dependency from external system modules.

A dedicated script, *env.sh*, was added to the repository to automate the creation of the virtual environment, the installation of all required packages and the configuration of the execution environment.

This ensures portability and reproducibility across HPC systems and development machines.

Co-funded by the European Union
EuroHPC Joint Undertaking

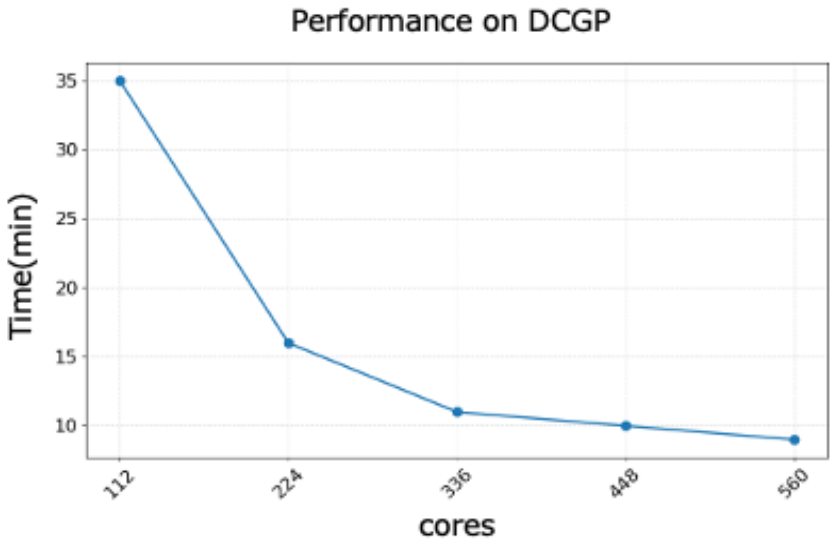


Figure 6 Scaling curve for the parallel Python program (project 89)

2.6.7 Improvements achieved

Benchmarks were performed in order to compare the old serial version of the BFEX code with the new MPI parallel implementation. Using the original serial pipeline, the analysis of a selected dataset was executed on a single core of the LEONARDO DCPG partition. After 24 hours of continuous execution, the workflow was still running. This result highlights the limitations of the previous implementation. A second test was performed using the optimized MPI-parallel version of the pipeline, scaling from 1 to 5 nodes (112-560 cores, see Figure 6). The results show a reduction in total analysis time: 35 minutes using 1 node (112 cores), around 16 minutes with 2 nodes (224 cores) and less than 10 minutes using 5 nodes (560 cores). We see that the parallel scaling tails off about 3 nodes, but the execution time at this point is probably dominated by I/O or other serial processes.

2.6.8 Summary and lessons learned

In this project a workflow based on Bash was converted to Python and then parallelised in a relative straightforward manner to achieve a vastly more efficient software code. One very positive result of this activity was that the PI and his colleagues, from an Italian pharmaceutical company, were extremely happy with the outcome and in fact invited the EPICURE AST team to give a seminar to their company.

2.7 Project 99: Monte Carlo simulations of spin glasses for quantum-simulation experiments (Lead: CSC)

Co-funded by the European Union
EuroHPC Joint Undertaking

2.7.1 Project overview

The original application was developed by the research group that applied for the support. It harnessed both shared-memory and distributed memory parallelism. The first approach was implemented using the numba `prange` facility, while the second implemented embarrassing parallelism using mpi4py (distributing disorder realizations). The code has already been tested on LUMI-C, but additional benchmarks were considered to be useful. The main goal was to use containers properly to minimize impact on the file system.

2.7.2 HPC systems used in the project

The HPC system targeted was LUMI.

2.7.3 Code description

The original code was a Python script, contained in 2-3 files, that relied on many libraries (such as numpy, scipy, numba, mpi4py or PyTorch) to perform scientific calculations. In particular, in the original setup, it used a Bash script to configure every run. This Bash script creates a folder for every MPI rank, adapting the input from a single file to a set of files (1 per rank) that are slightly different (according to some predefined variables that use the process id to select the values). This process is repeated for each of the input files (4-5 of them are used). Then every MPI rank also creates its own output folder, which contains a small set of files (output and debug logs). It is a pure CPU code, so no GPUs are involved.

2.7.4 Status before optimisation

The main issue with this approach is that it doesn't scale well. There are well known issues with the parallel file system from both the Python side (having all the ranks run a lot of imports means a lot of requests to the metadata server) and from the IO side (having all the ranks read a lot of different input and producing different output files is a well-known problem in Lustre scaling). These problems do not immediately appear in our profiling runs as the number of ranks involved is very low (just 10 ranks were used, in 1 single node). However, should the user scale this to the full HPC system, both these issues would affect the machine.

In WP2 profiling of the code was done using cProfile (see Figure 7) and this was used to guide the optimisation activity.

Co-funded by the European Union
EuroHPC Joint Undertaking

```
       86129095 function calls (84996085 primitive calls) in 2025.055 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    17184  951.299    0.055  951.369    0.055 dmc.py:434(ZxpHot)
     2000  349.828    0.175  354.157    0.177 dmc.py:900(skmagfact)
        1  306.546  306.546 2024.826 2024.826 dmc.py:935(rundmc)
    12012  104.883    0.009  104.883    0.009 {built-in method numpy.asarray}
  4800000   90.051    0.000   91.876    0.000 dmc.py:470(PupHvecto)
  1600000   35.010    0.000   36.302    0.000 dmc.py:510(summer)
    40000   32.988    0.001  226.700    0.006 dmc.py:532(kstep)
  1600000   24.480    0.000   25.137    0.000 dmc.py:480(PupVvecto)
  1823928   20.897    0.000   20.897    0.000 {built-in method numpy.array}
    17184   13.987    0.001   14.007    0.001 dmc.py:396(epotALL)
  1600000   11.050    0.000   49.600    0.000 dmc.py:492(sampleH)
    40057   10.815    0.000   11.381    0.000 ffi.py:190(__call__)
  1600000   10.046    0.000   42.253    0.000 dmc.py:501(sampleV)
  3236361    7.355    0.000    7.355    0.000 {method 'rand' of 'numpy.random.mtrand.RandomState' objects}
```

Figure 7 cProfile output of the original code (project 99)

2.7.5 Optimisation objectives

In WP3 the objectives were to:

1. Improve, if possible, the performance of the Python script itself
2. Create an ad-hoc container for all the required Python libraries
3. Rework the IO system completely, to introduce the usage of HDF5 to lessen the pressure on the Lustre parallel file system.

2.7.6 Code optimisation

In WP3 the first action was to improve the Python code by adding one more function to the set of functions that are compiled with the JIT compiler, resulting in a small performance improvement (around 5% on the total runtime).

The second action was to create an ad-hoc container to include all the Python libraries needed by the code for running. The goal for this was twofold:

- isolate all the Python small files into a single big file (i.e. the container image). It has been proven that on the LUMI Lustre filesystems (see https://lumi-supercomputer.github.io/LUMI-training-materials/2day-20251020/205-Containers/#but-what-can-they-then-do-on-lumi) this is very beneficial for performance since it allows multiple Python module loading to be avoided.
- provide container-optimized software for Lumi by bind mounting the most important libraries (e.g. cray-mpich). This ensures that key Python packages, such as mpi4py, are explicitly compiled and linked against the optimized system libraries, delivering maximum performance rather than relying on potentially suboptimal binary compatibility layers. We also need to introduce some extra dependencies that were not needed before to manage the new model of I/O (i.e. h5py).

The input management architecture has also been revised. We decided to perform data preprocessing on the fly rather than relying on a separate step. This approach removes

Co-funded by the European Union
EuroHPC Joint Undertaking

dependencies on extraneous folders and files, which better accommodates distributed filesystems and enables greater scaling (i.e., a higher number of ranks/nodes can be utilized). In the new implementation, rank 0 is responsible for loading the data and distributing it via an MPI broadcast to all participating ranks. Subsequent adjustments are made by individual ranks according to their respective rank_id.

The output stage suffered from a similar problem: the previous method created several files per rank within dedicated folders. We decided to condense all this data into a single HDF5 file. An exception was made for certain "status dumps" needed for debugging. These dumps do not have a fixed size, so agreeing on a universal size across all ranks during the write operation would be computationally expensive. This is not an issue, however, as these files should not exist during production runs. This consolidation prompted us to develop an additional Python script to help developers easily extract the necessary results from the HDF5 file during postprocessing.

2.7.7 Improvements achieved

The overall speedup is not outstanding: just about 5% as can be seen comparing Figure 7 and Figure 8 (before and after optimisation).

```
         60990039 function calls (60149962 primitive calls) in 1886.539 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    17184  951.249    0.055  951.324    0.055 dmc.py:440(ZxpHot)
        1  330.777  330.777 1886.317 1886.317 dmc.py:950(rundmc)
     2000  187.187    0.094  187.204    0.094 dmc.py:914(skmagfact)
    12012  111.496    0.009  111.496    0.009 {built-in method numpy.asarray}
  4800000   90.168    0.000   91.999    0.000 dmc.py:476(PupHvecto)
  1600000   35.676    0.000   37.077    0.000 dmc.py:518(summer)
    40000   33.085    0.001  226.994    0.006 dmc.py:541(kstep)
  1600000   24.969    0.000   25.640    0.000 dmc.py:486(PupVvecto)
  1823928   20.950    0.000   20.950    0.000 {built-in method numpy.array}
    17184   14.046    0.001   14.067    0.001 dmc.py:400(epotALL)
  1600000   10.993    0.000   49.541    0.000 dmc.py:498(sampleH)
  1600000    9.879    0.000   42.069    0.000 dmc.py:508(sampleV)
31078/31072    7.413    0.000    7.864    0.000 ffi.py:190(__call__)
  3236361    7.071    0.000    7.071    0.000 {method 'rand' of 'numpy.random.mtrand.RandomState' objects}
```

Figure 8 cProfile output of the improved code (project 99)

However, the code has a better organisation, and we removed the necessary input preprocessing step in favour of better scaling to multiple nodes and ranks. The whole approach is more HPC friendly and avoids overloading some of the weak points of HPC systems (such as the metadata server in parallel file systems).

Co-funded by the European Union
EuroHPC Joint Undertaking

2.7.8 Summary and lessons learned

Lessons learned include:

- using JIT for random number generation leads to instability of the code
- compiling Python packages from sources instead of using pip wheels sometimes gives some performance increases, but these are limited and probably not worthwhile for future projects. For example, `numpy` from the wheel (i.e. with OpenBLAS backend) is almost as fast as `numpy` compiled (i.e. with `libsci` backend). On the other hand, `PyTorch` was a little slower (compared to the wheel compiled by AMD targeting ROCM), so it was dropped, and the Python wheel was used instead.
- `cProfile` was a useful tool, working "out of the box".

2.8 Project 107: BCPNNSim (Lead: CINES)

2.8.1 Project overview

This project aimed to port BCPNNSim, a custom-built neural network simulator, to HPC systems while ensuring scalable performance. The first step focused on bringing the HIP-based LUMI code to the same performance level as the CUDA implementation. The second step involved profiling the code to identify performance bottlenecks and applying targeted optimisations. Finally, in the third step, the goal was to enable the code to run on multiple GPUs using MPI, implementing model parallelism by distributing the network across the GPUs.

2.8.2 HPC systems used in the project

The HPC system targeted was LUMI.

2.8.3 Code description

The code is written in C/C++ parallelized with HIP (+ MPI for multiple GPU) and implements brain-like neural networks (biologically plausible learning such as rewiring, which is different from off-the-shelf deep learning software). It mainly uses vector-matrix dot products and matrix-matrix outer products as core computations. The code is optimized for high sparsity (1-10% connectivity) using a packed dense connection matrix (custom-made).

2.8.4 Status before optimisation

Before EPICURE support, the HIP version of BCPNNSim tested on LUMI exhibited low single-GPU performance. Multi-GPU execution was not yet implemented, and the code had not been profiled. The measured runtimes and accuracies (train/test) are summarized below for three different set-ups: a workstation, the Dardel system at KTH and LUMI (see Table 8).

Co-funded by the European Union
EuroHPC Joint Undertaking

Table 8 Performance comparison before EPICURE support. The train and test accuracies are given in parentheses.

Training / Test patterns	Workstation (RTX A4000 with CUDA)	Dardel (HIP)	LUMI (HIP)
1k/1k	32s (99.5/90.6)	2m32s (97.9/86.4)	38m45s (98.0/85.6)

The data highlight the poor single-GPU performance on HPC systems and the need for multi-GPU support. EPICURE WP2 support determined that the initial runtime on LUMI (38 minutes) was mainly caused by incorrect CPU binding and suboptimal handling of managed memory. In particular, enabling GPU page fault support for managed memory (HSA_XNACK=1) resulted in a significant performance improvement. Additional gains were obtained through improved compiler optimization flags (e.g. -flto, -funroll-loops, -fno-math-errno). This helped to achieve similar performance between LUMI and Dardel (around 2min for 1k/1k patterns).

2.8.5 Optimisation objectives

The first optimisation objective was to achieve CUDA-level performance with the HIP version of BCPNNSim, on the LUMI architecture. The aim was then to increase the performance and scalability. Finally, we attempted to enable multi-GPU and multi-node support.

2.8.6 Code optimisation

Several code optimisation activities were carried out to improve performance on LUMI:

- Compilation and launch parameters were tuned by modifying the Makefile and launch scripts.
- Excessive use of `hipMemcpy`, `hipMallocManaged` and `hipDeviceSynchronize` was reduced (Figure 9).
- Profiling was performed using **rocprof** to identify performance bottlenecks (Figure 10).
- Key HIP kernels were optimized for better execution efficiency.
- Most of the code was ported to the GPU to minimize managed memory access.

Co-funded by the European Union
EuroHPC Joint Undertaking

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
47.47	63,900,164,153	144,000	443,751.14	439,193.00	6,733	1,530,285	745,571.25	hipMemcpy
27.31	36,760,495,246	180,306	203,878.38	193,296.00	461	2,744,499	486,988.45	hipDeviceSynchronize
20.53	27,631,904,056	324,334	85,195.83	81,731.50	7,915	343,596,204	719,567.95	hipMemset
3.66	4,921,066,759	1,933,169	2,545.60	1,872.00	1,222	44,818,468	32,397.42	hipLaunchKernel
0.64	862,906,662	71	12,153,614.96	11,899,220.0	7,634	855,010,070	101,458,653.94	hipMallocManaged
0.18	235,792,867	4,042,163	58.33	40.0	40	13,446	30.82	hipPeekAtLastError
0.11	154,155,412	1,933,169	79.74	60.0	9,589	34.06		__hipPopCallConfiguration
0.10	140,384,012	1,933,169	72.62	60.0	7,985	31.77		__hipPushCallConfiguration
0.00	6,228,857	21,075	295.56	171.0	144,759	1,166.84		__hipRegisterFunction
0.00	1,530,888	15,001	102.05	60.0	3,476	55.45		hipGetLastError
0.00	935,647	2	467,823.50	460,000.0	9,118	926,529	648,707.54	hipMemcpyAsync
0.00	97,267	4	24,316.75	22,000.0	3,927	33,615	13,844.78	hipMalloc

Figure 9 Time spent in different HIP operations (project 107)

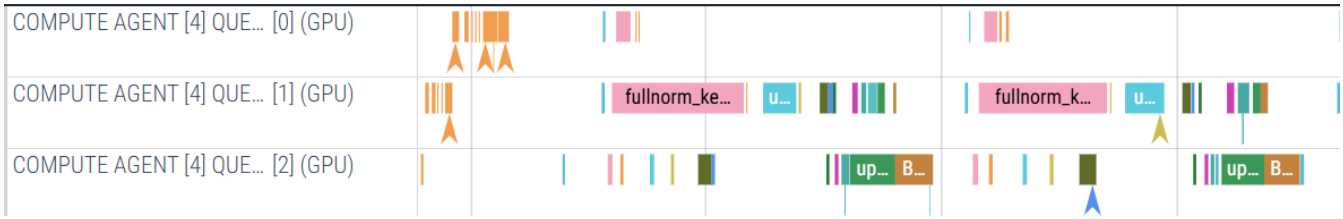


Figure 10 Kernels trace profiled with rocprofv3 (project 107)

These optimisations resulted in significant runtime improvements (Table 9) and consistently high accuracy scores (Table 10) across all tested hidden-layer sizes (N). It is also clear from Table 10 that the HIP implementation does not affect the accuracies when compared to the CUDA results.

Table 9 Runtime comparison between CUDA on a workstation and HIP on LUMI

N	Workstation (CUDA)	LUMI (HIP)
10	684.953	712.726
30	1363.080	915.927
100	4022.644	1712.554
300	12365.389	3887.416
1000	38236.973	12090.986

Co-funded by the European Union
EuroHPC Joint Undertaking

Table 10 Train/Test Accuracy (%) Using CUDA (NVIDIA Workstation) vs HIP (LUMI

N	Workstation (CUDA)	LUMI (HIP)
10	94.1 / 94.1	94.4 / 94.4
30	95.2 / 95.3	95.2 / 95.3
100	95.3 / 95.6	95.4 / 95.6
300	95.5 / 95.5	95.5 / 95.5
1000	95.6 / 95.5	95.6 / 95.7

2.8.7 Improvements achieved

Significant optimisations were achieved in BCPNNSim on LUMI:

- Most of the code was successfully moved to the GPU, which reduces the reliance on managed memory and gives an additional ≈ 2 ×–3 × speed-up for large problem sizes.
- All core computational functions (population updates, projection updates and connection updates) were re-implemented as GPU kernels that run entirely on the device.
- Population and projection updates are now parallelised across GPU threads. For the few operations that must remain serial, we experimented with HIP events to minimise synchronisation overhead.
- The HIP version on LUMI achieved an overall speedup of approximately 150× compared to the initial HIP baseline, with most of the improvements resulting from optimized compilation and launch parameters.

Table 11 Table 11 Performance gain with GPU porting

MNIST dataset train / test	Time (s) before GPU porting	Time (s) after GPU porting
1000/1000	54.63	13.28
6000/1000	276.05	63.95
60000/10000	2752.29	683.56

The optimisation efforts not only improved runtime performance but also established a robust GPU port of the code, providing a solid foundation for future multi-GPU and multi-node extensions.

Co-funded by the European Union
EuroHPC Joint Undertaking

2.8.8 Summary and lessons learned

The porting and optimisation of BCPNNSim on LUMI successfully achieved the main objectives of the project. The HIP version of the code now matches or exceeds CUDA-level performance, with an overall speedup of up to **150 times** compared to the initial HIP baseline and **2-3 times** speedup over CUDA for large datasets. Most of the code ported to the GPU, reducing reliance on managed memory and improving execution efficiency.

Lessons learned:

- Profiling and tuning compilation/launch parameters are critical for achieving high performance on HPC systems.
- GPU porting requires careful management of memory accesses and kernel optimisation to exploit hardware capabilities.
- Multi-GPU and multi-node parallelization require substantial preparatory work and should be planned early in the project timeline.

The code changes developed during this project were integrated into the BCPNNSim repository and provide a robust foundation for future developments.

2.9 Project 108: GPU-enabling of explicit compressible OpenFOAM solvers for flow and acoustic prediction (Lead: CINECA)

2.9.1 Project overview

This project aims to port the *caaFOAM* solver to GPU architectures. Due to its explicit time integration scheme, the solver requires small time-steps to maintain numerical stability, especially when capturing acoustic phenomena. GPU acceleration will substantially reduce computational time, making it feasible to tackle larger-scale problems and enabling more detailed simulations within practical timeframes.

2.9.2 HPC systems used in the project

All the simulations were performed using the GPUs of Leonardo Booster.

2.9.3 Code description

caaFOAM is a density-based compressible Navier-Stokes solver built upon the OpenFOAM framework, designed for acoustic wave propagation and aeroacoustic simulations. Written in C++, the code leverages OpenFOAM's MPI parallelization capabilities for efficient execution on

Co-funded by the European Union
EuroHPC Joint Undertaking

distributed-memory architectures. The solver implements a finite volume discretization on unstructured meshes with an explicit time-stepping scheme, making it well-suited for capturing acoustic phenomena that require fine temporal resolution.

2.9.4 Status before optimisation

The original caaFOAM solver was CPU-only with no GPU support. While MPI parallelization enabled multi-node execution, the explicit time integration scheme required very small time-steps for stability, leading to high computational costs and long wall-clock times, particularly for acoustic simulations that require a fine temporal resolution.

2.9.5 Optimisation objectives

The primary objective was to port the caaFOAM solver to GPU architectures to achieve a significant performance enhancement. Specific goals included:

- Enable GPU execution in the OpenFOAM framework
- Achieve a substantial speedup compared to the CPU-only baseline
- Maintain numerical accuracy and stability of the explicit time integration scheme
- Preserve MPI parallelization capabilities for multi-GPU execution
- Reduce wall-clock time for acoustic simulations, enabling larger-scale problems and finer grid resolutions within practical timeframes.

2.9.6 Code optimisation

The porting to GPU for caaFOAM was performed with the SPUMA package, a specialised C++ library for solving partial differential equations (PDEs) which is also equipped with a CUDA backend for NVIDIA GPU acceleration (see section 2.10.3 for more details and references). The porting effort was accompanied by systematic optimisation work to maximize performance and scalability. This section presents the optimisation strategies implemented and their impact on computational efficiency. Three primary optimisation approaches were evaluated on two benchmark cases: a cylinder flow simulation with 80 million hexahedral elements and a NACA airfoil case with 480 million hexahedral elements. Both test cases were run without turbulence models, and timing measurements were collected over 100 iterations.

The first optimisation involved **caching the velocity gradient field** (grad(U)) to avoid redundant computations. This modification required no changes to the caaFOAM source code and yielded substantial performance gains:

- **80M element case:** 20-25% reduction in execution time across all GPU counts
- **480M element case:** 18-27% reduction in execution time

The benefit was most pronounced at lower GPU counts, suggesting that gradient computation represents a significant portion of the total computational cost.

Co-funded by the European Union
EuroHPC Joint Undertaking

Cell renumbering was applied to improve memory access patterns and cache efficiency. However, the additional benefit beyond gradient caching was modest (only 1-11% additional improvement over gradient caching alone). This limited benefit is attributed to the structured nature of the hexahedral meshes used, which already exhibit favourable memory access patterns. Most Cartesian mesh generators inherently produce well-ordered cell numbering that minimizes matrix bandwidth.

For the 480M element case, an additional optimisation was implemented by **modifying the gradient computation within the viscous flux calculation** in the caaFOAM source code using a cell-based approach. Combined with gradient caching and renumbering, this yielded the best performance with 33-38% reduction compared to baseline. At 256 GPUs, the solver achieved 0.23 seconds per iteration. This optimisation specifically targets the expensive gradient operations required for viscous stress tensor computation in the compressible Navier-Stokes equations.

Table 12 summarizes the performance improvements for both test cases with the optimized configuration.

Table 12 Optimized Performance Summary

No. test Case (elements)	GPUs	Time/Iteration (s)	Speedup vs Baseline	Parallel Efficiency
80M	4	1.33	1.35	100%
80M	64	0.21	1.32	57.8%
80M	128	0.20	1.39	35.5%
480M	32	0.95	1.60	100%
480M	128	0.33	1.49	76.4%
480M	256	0.23	1.50	55.8%

Strong scaling tests were conducted to evaluate parallel efficiency as the number of GPUs increased while keeping the problem size fixed.

For the 80M element case the solver demonstrates excellent scaling up to 32 GPUs with 75.2% parallel efficiency (Figure 11). Beyond this point, communication overhead becomes increasingly significant, with efficiency dropping to 57.8% at 64 GPUs and 35.5% at 128 GPUs. The degradation at the highest GPU counts indicates that the problem size becomes too small relative to the communication-to-computation ratio.

Co-funded by the European Union
EuroHPC Joint Undertaking

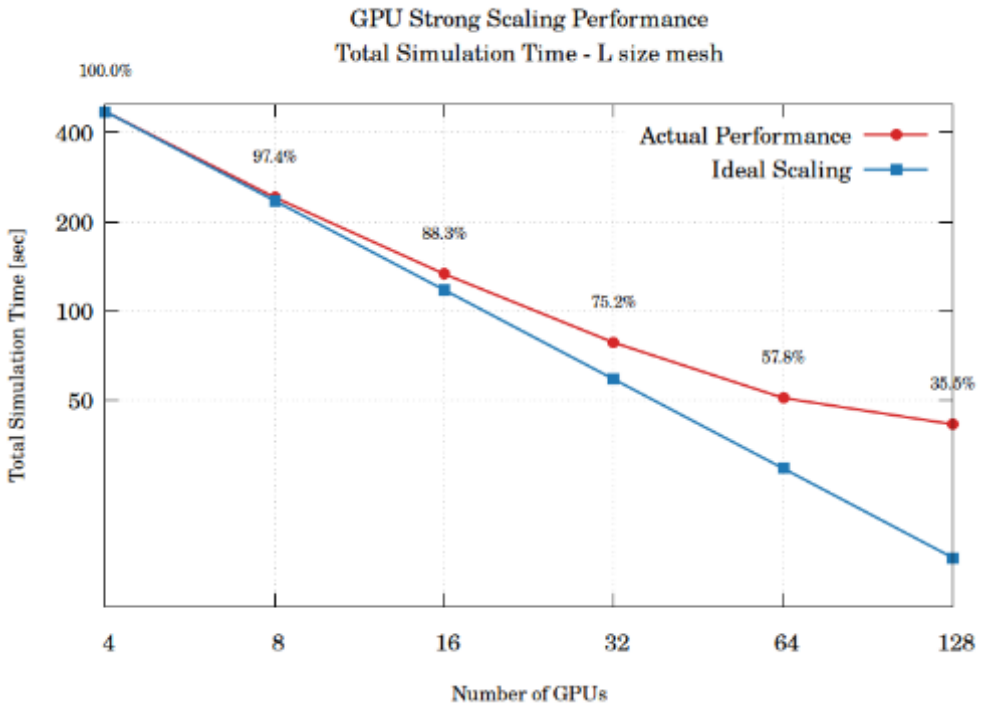


Figure 11 GPU Strong Scaling Performance - Total Simulation Time (80M element mesh) (project 108)

For the 480M element case (Figure 12) the solver exhibits superior scaling characteristics, maintaining 91.3% efficiency at 64 GPUs and 76.4% at 128 GPUs. Even at 256 GPUs, parallel efficiency remains at 55.8%, demonstrating that larger meshes can effectively utilize more GPU resources. This scaling behaviour is typical of explicit time-stepping schemes where each iteration requires only nearest-neighbour communication.

Co-funded by the European Union
EuroHPC Joint Undertaking

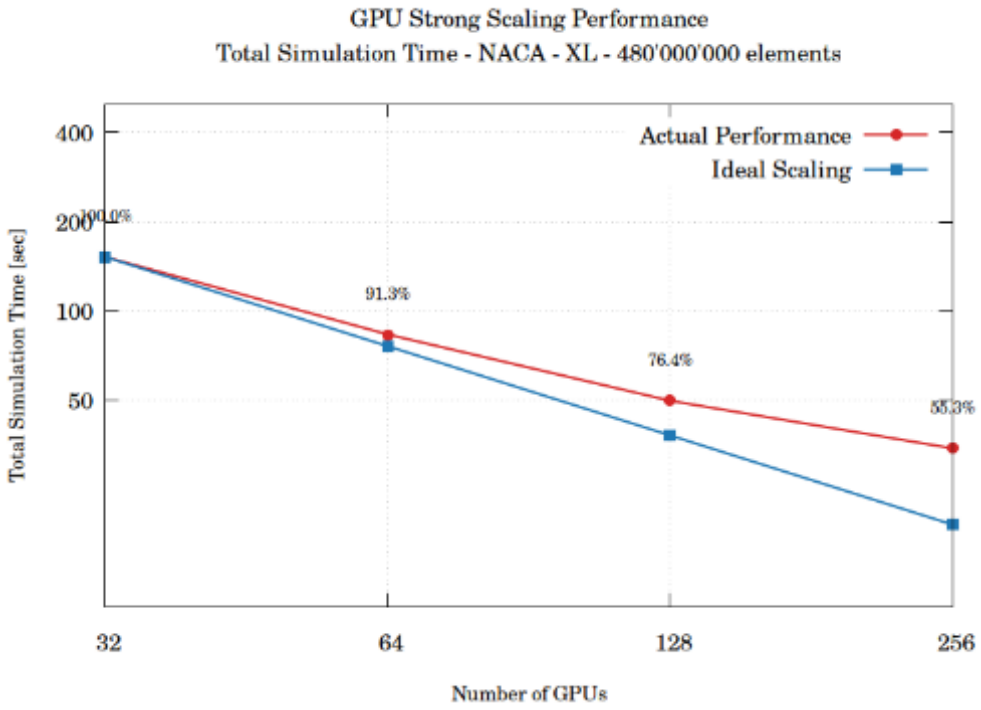


Figure 12 GPU Strong Scaling Performance - Total Simulation Time (480M element mesh) (project 108)

These optimisations collectively make caaFOAM competitive for production-scale aeroacoustic simulations on HPC clusters with GPUs, with the explicit time-stepping scheme benefiting significantly from GPU acceleration despite its inherent limitation to small time-steps.

2.9.7 Improvements achieved

Taking as reference the results from the 480M element case, we can demonstrate that the optimized GPU implementation delivers a **13x speedup** at 256 GPUs compared to the CPU baseline running on 1600 cores, reducing simulation time from 447.4 seconds to 34.4 seconds for 100 iterations. This represents a substantial performance improvement when compared to the CPU-runs (see Figure 13 and Figure 14).

Co-funded by the European Union
EuroHPC Joint Undertaking

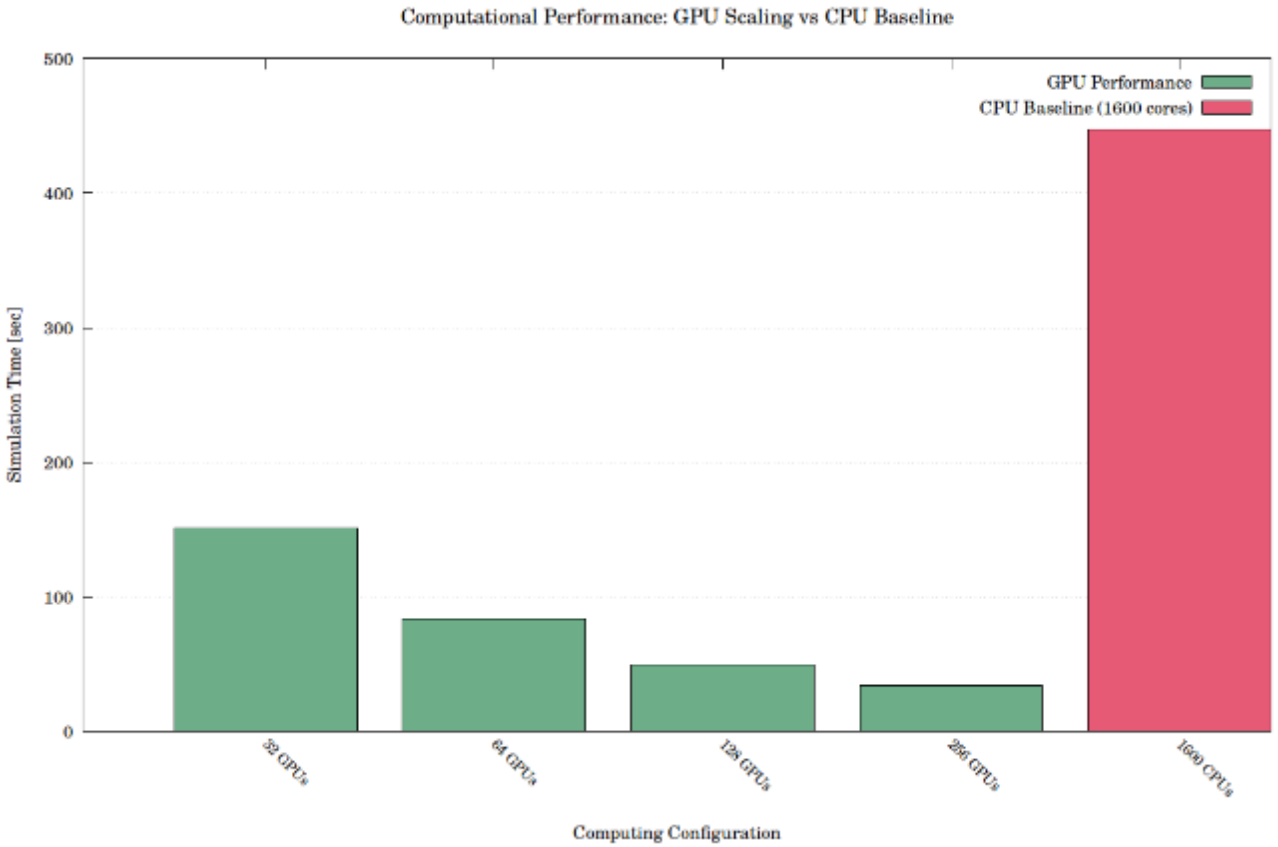


Figure 13 Computational Performance - GPU Scaling vs CPU Baseline (project 108)

Co-funded by the European Union
EuroHPC Joint Undertaking

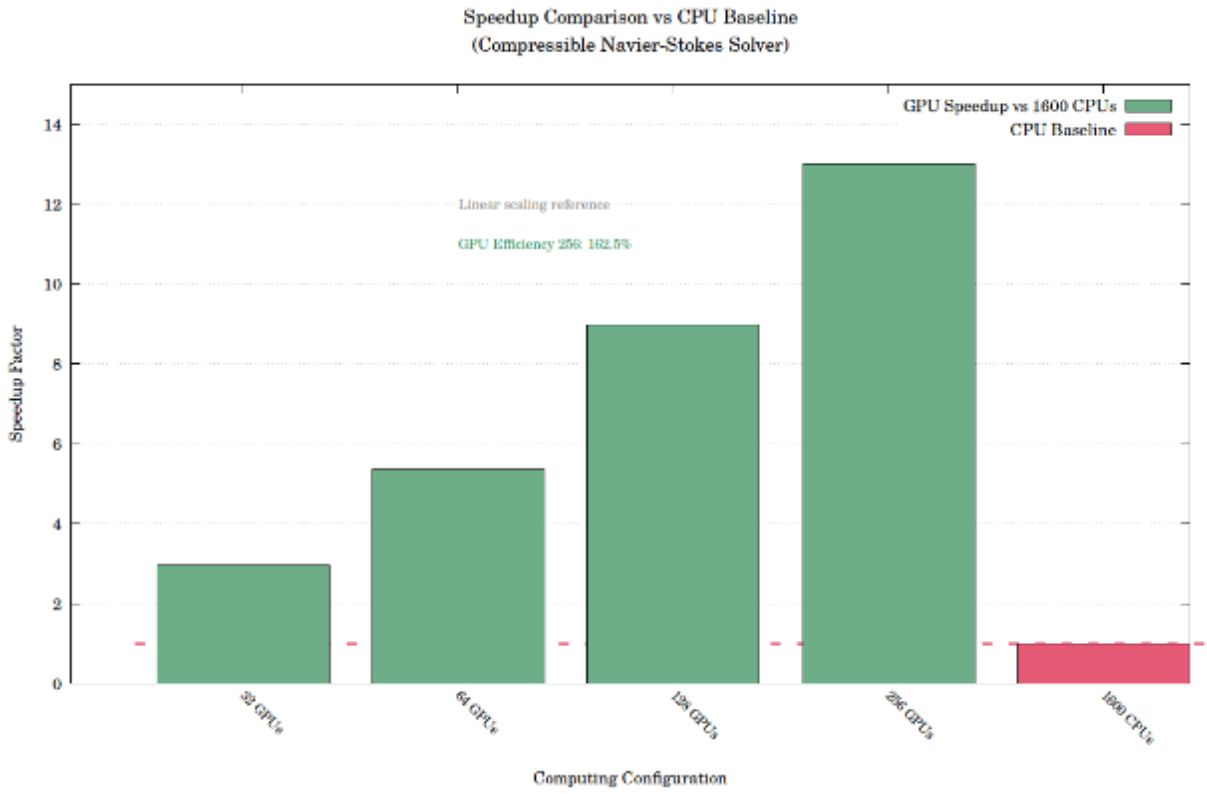


Figure 14 Speedup Comparison vs CPU Baseline (project 108)

2.9.8 Summary and lessons learned

This project successfully ported the caaFOAM solver, an OpenFOAM-based compressible Navier-Stokes solver for acoustic simulations, from a CPU-only implementation to a GPU-accelerated execution. The baseline code ran exclusively on CPUs with MPI parallelization and lacked GPU support, resulting in long computational times due to the small time-steps required by the explicit time integration scheme.

The optimisation process revealed several important insights about GPU porting for CFD applications. Perhaps most surprisingly, the largest performance gains came not from sophisticated GPU programming techniques but from simple algorithm-level changes. Gradient caching alone provided 20-27% improvement without touching the source code, teaching us that understanding computational redundancy and eliminating repeated calculations should always be the first step before diving into complex GPU optimisations.

Co-funded by the European Union
EuroHPC Joint Undertaking

2.10 Project 109: Exploiting GPU computational power to simulate a turbulent flow over a grid spaced 127 pin bundle using SPUMA (Lead: CINECA)

2.10.1 Project overview

In this project, the promising GPU-based CFD code named SPUMA was used to improve the accuracy of CFD simulations in the nuclear reactor field. More specifically, the international benchmark on thermal hydraulic simulations of LFR (Lead-cooled Fast Reactor) components, released under the guidance of the Italian Nuclear Energy Agency and ENEA[2,3], was executed on the Leonardo HPC cluster using the GPU-accelerated version of OpenFOAM developed and released by CINECA. The problem is characterized by a pin bundle with 127 rods of 5.5 mm in radius and 2.67 mm in gap between pins and three spacer grids giving a total axial height of 1.5m (Figure 15). To accurately describe the geometry, several meshes were provided, ranging from a coarse mesh of 80M cells to a fine mesh of 400M cells. The fluid flowing in the bundle is isothermal, turbulent and steady and can be simulated using the *simpleFoam* solver (see Figure 16 and Figure 17).

2.10.2 HPC systems used in the project

All the simulations have been performed using the GPUs of Leonardo Booster and compared with the CPUs of the CRESCO6 cluster hosted in ENEA (CRESCO6 is a cluster consisting of 434 nodes, each consisting of 2x24 core Intel(R) Xeon(R) Platinum 8160 processors and 192 GB RAM).

2.10.3 Code description

SPUMA is a fork of OpenFOAM, a C++ toolbox that implements a DSL (Domain Specific Language) for solving PDEs using the Finite Volume Method. The number of applications of OpenFOAM is huge spanning from complex fluid flows involving chemical reactions, turbulence and heat transfer, to acoustics, solid mechanics and electromagnetics. SPUMA implements a minimally invasive approach to the GPU porting of OpenFOAM and it can run both on AMD and NVIDIA GPUs[4]. The architecture of SPUMA is based on a portable programming model where several computational backends can be developed and registered. In the current version, a CUDA backend is available for NVIDIA GPUs and a HIP backend for AMD GPUs. Parallelisation is obtained using the domain

[2] https://www.oecd-nea.org/jcms/pl_66836/lead-cooled-fast-reactor-benchmark-lfr

[3] Craig Menezes, Trevor Melsheimer, Yassin A. Hassan; Flow field characteristics of a 127-pin rod bundle with hexagonal spacer grids. Physics of Fluids 1 June 2023; 35 (6): 067119. https://doi.org/10.1063/5.0154590

[4] Simone Bna', Giuseppe Giaquinto, Ettore Fadiga, Tommaso Zanelli, Francesco Bottau, SPUMA: a minimally invasive approach to the GPU porting of OpenFOAM, accepted for publication in Computer Physics Communications, 2025.

Co-funded by the European Union

EuroHPC Joint Undertaking

decomposition algorithm and the exchange of boundary data between subdomains is managed by the MPI protocol.

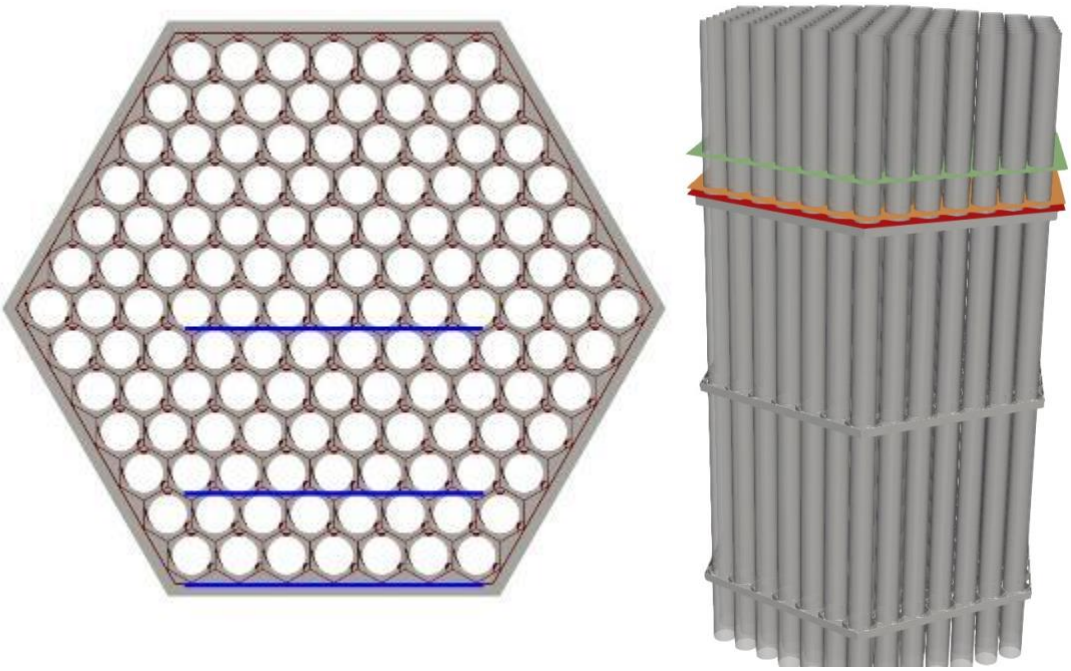


Figure 15 pin bundle representation (project 109)

SPUMA in its current form has only a limited number of solvers, turbulence models and boundary conditions that run completely on GPUs. The benchmark previously described requires some boundary conditions, wall functions and a turbulence model that have not yet been fully accelerated, implying a data transfer between CPUs and GPUs and a consistent slowdown of a simulation execution.

2.10.4 Status before optimisation

The application code has only been partly ported to the GPU.

2.10.5 Optimisation objectives

The objective of this project was mainly related to the GPU porting of the missing features described in the previous paragraphs. Code changes have been included into the development branch of the SPUMA repository.

Co-funded by the European Union

EuroHPC Joint Undertaking

2.10.6 Code optimisation

The NVIDIA profiler was used to locate the parts of the SPUMA code that run on CPU and trigger page migrations, causing a consistent slow-down of the simulation execution. These parts, characterized by for-loops and reductions, were accelerated using the high-level APIs available in the SPUMA framework and using the CUDA backend optimized for NVIDIA GPUs.

Four meshes ranging from 80M cells to 400M cells, with or without the boundary layer, were generated as reported in Table 13. The two meshes with the boundary layer (partial or full) failed to converge and no performance results were provided. For the other two meshes, we ran successfully the *simpleFoam* solver for 20k iterations using two different linear algebra solvers for the pressure equation, i.e the GAMG native solver and the NVIDIA AmgX package. Validation of the results were carried out together with ENEA and we concluded that the discrepancy with respect to the available experimental data cannot be attributed to SPUMA implementation errors because SPUMA and OpenFOAM vanilla are in perfect agreement.

Performance results obtained in the Leonardo cluster using the M80 and M320 meshes are reported in Table 14; for the M80 case only, we have also reported the reference performance result obtained in the CRESCO6 cluster hosted at ENEA.

Due to the small size of the ENEA cluster, the maximum number of CPUs available was 288 cores, resulting in long simulations (around 34h) even for the coarse mesh M80 and for only 6100 steps.

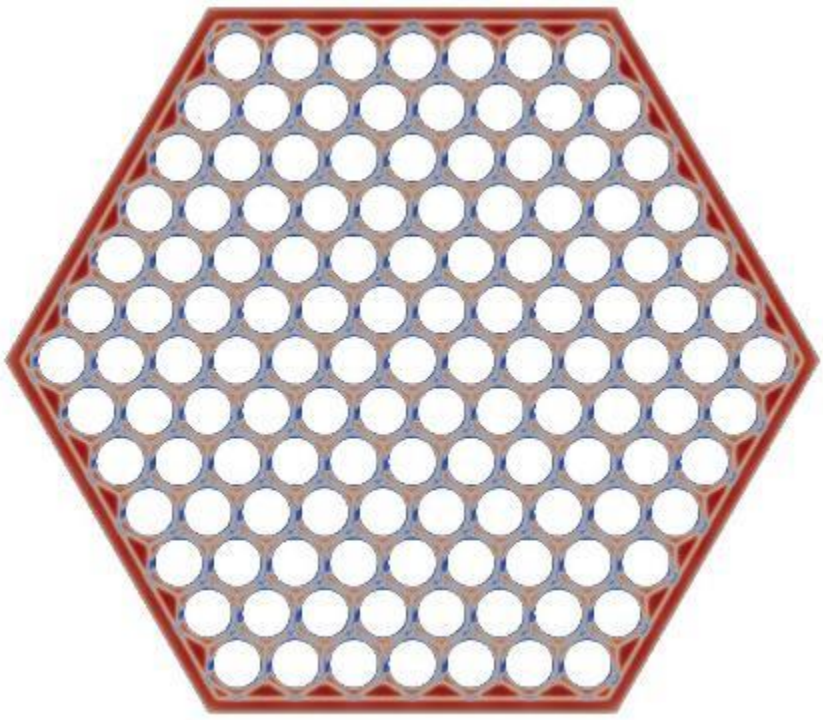


Figure 16 flow representation in the pin bundle (project 109)

Co-funded by the European Union
EuroHPC Joint Undertaking

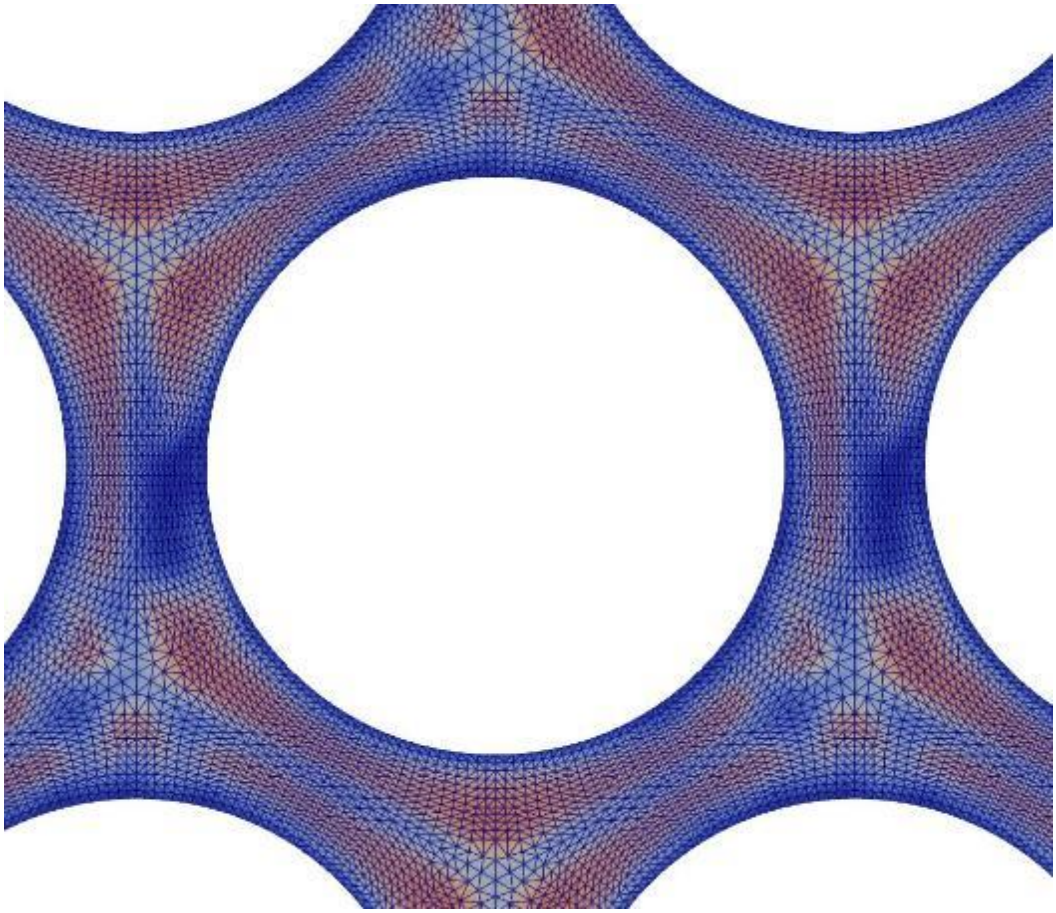


Figure 17 Flow representation (with visible mesh) around a pin (project 109)

Table 13 Test case descriptions

Test case	Number of Cells	Boundary layer	Number of Partitions
M80	80,379,497	no	8
M330	328,989,030	no	32
M330BL	332,306,880	yes (partial)	32
M400BL	392,882,939	yes (full)	40

Co-funded by the European Union
EuroHPC Joint Undertaking

Table 14 Performance results

Mesh	Time steps	Enea CRESCO 6	Leonardo Booster (AmgX linear solver)	Leonardo Booster (GAMG linear solver)
M80	6100	124284 s on 288 cores	10895 s on 8 GPUs	12887s on 8 GPUs
M80	20000	N.A.	26239 s on 8 GPUs	26922s on 8 GPUs
M330	20000	N.A.	42932 s on 32 GPUs	56060s on 32 GPUs

Using only 2 nodes of Leonardo booster, equivalent to 8 GPUs in total, the wall-time was reduced by a factor of 11 and allows 20k iterations to be performed in approximately 7 hours. Time increases from about 7 hours to 12 hours if we use 32 GPUs considering the M330 mesh and the AmgX linear solver. The native GAMG linear solver was revealed to be slower than AmgX, especially for the M330 case.

2.10.7 Improvements achieved

The project achieved the successful porting to GPUs of the entire workflow of a CFD simulation of a nuclear reactor bundle. As reported in the previous section, we achieved speedups of over 11 using 8 GPUs when compared to 288 cores on the ENEA cluster. The SPUMA tool was also updated for GPU-based simulations.

2.10.8 Summary and lessons learned

SPUMA was shown to be a valid tool to accurately simulate the fluid flow in a complex nuclear reactor bundle in the field of nuclear engineering. The GPU acceleration of SPUMA strongly reduces the number of devices and the walltime spent in the simulations.

2.11 Project 113: Compilation and porting of the code THATMPI using the Cray compiler in LUMI (Lead: CYFRONET, Collaborators: KTH)

2.11.1 Project overview

The focus of this project was to optimize the performance of the proprietary code THATMPI on the LUMI machine. This is a Fortran, CPU-only code with MPI and OpenMP parallelization, used for plasma simulations. The initial performance on LUMI, which is an AMD-based system, was not

Co-funded by the European Union
EuroHPC Joint Undertaking

considered satisfactory by the project PI. Since the code can be compiled only with Intel compilers, and those are not available by default on LUMI, the PI of the project assumed that code porting was necessary to achieve good performance. However, he was open to any comments and changes that could improve the code's performance on LUMI and other clusters. The work on this project included both WP2 and WP3 activities, like preparing tests, benchmarking, code refactoring, debugging, implementation of new features, as well as demonstration of proof-of-concept on how the code can be ported to be compiled by Cray and gfortran compilers.

2.11.2 HPC systems used in the project

LUMI (CPU-nodes) and additional resources such as the Helios and Ares clusters at ACC CYFRONET AGH (national access).

2.11.3 Code description

THATMPI is a code that implements the particle-in-cell (PIC) method, used for modelling collision-less plasma systems in plasma physics and astrophysics applications. The code is parallelized with MPI and OpenMP and written in Fortran. It was modernized a few years ago (within national support and PRACE preparatory access) and uses features from the 2003 and 2008 standards. It uses MPI for parallelization and is based on the division of a computational grid into fixed-size domains, where each domain is handled by a single task. Since this leads to an imbalance, OpenMP parallelization can be used to share the most compute-intensive phase among all threads assigned to each process, but each thread is responsible for all other computations in a single domain assigned to this thread. The code uses HDF5 for I/O, with mostly serial operations, but some of them use parallel HDF. The code is enriched with a small C++ tool, whose API is called from Fortran routines, which at the end of the program stores logs with time durations of each of the sections marked in the code. The code is not publicly available, but it is shared internally by members of the scientific group in IFJ PAN (Institute of Nuclear Physics, Polish Academy of Sciences). The code compilation is done by a handcrafted Makefile. It was always compiled with the Intel Fortran compiler (ifort), so it may rely on some Intel non-standard extensions and the way Intel compilers interpret Fortran syntax (which is quite permissive). In particular, the code uses PDTs (Parameterized Derived Types), which is part of the Fortran 2003 standard, but most of the compilers, except Intel ones, lack mature support for this feature.

2.11.4 Status before optimisation

Before EPICURE support, it was hard to compare the performance of the code between different machines, compiler optimisations, and different simulations. OpenMP also did not work correctly for multiple threads per process, as code changes made by users over the last few years were not made with OpenMP in mind. All this led to simulations not always being run in an efficient manner.

WP2 support organized a set of tests within the project repository and enabled a convenient way of building and benchmarking multiple binary versions of the code, each compiled with a different compiler (ifx or ifort) and a different set of optimisation flags. It also provided an initial port to Cray and gfortran compilers, which compiled the code successfully, but there were memory issues at

Co-funded by the European Union
EuroHPC Joint Undertaking

runtime. This activity requires further work, but due to performance improvements for Intel compilers, it is questionable whether this effort is worth continuing. The Makefile was updated, and now the code easily compiles with both Intel compilers – ifort (legacy) and new LLVM-based ifx compiler.

WP3 support implemented performance metrics, which are now printed directly in the output of the simulation. There is a *simulation rate*, measured in Mpart/s (millions of particles processed per second), as well as more detailed values. This work was complementary to the benchmarking effort in WP2, since now the simulation rate is used as a score in the benchmark. The code was checked for OpenMP usage, some bugs were removed, and now this feature can be used to achieve a speedup. Furthermore, a new domain-to-process assignment strategy was implemented, which can be used for long simulations that require many columns in a computational grid of domains. As an additional effort, the code was also refactored in a few places and error handling for I/O operations is now improved. Moreover, a command line interface was implemented. This allows one to change some simulation parameters, turn debugging on and off, or make dry runs, just with arguments passed to the program on the command line. This feature allows for easier testing. One of the modules was refactored and split into multiple files, allowing for mixed compilation with different optimisation flags for the two most compute-intensive phases of the simulation.

Benchmarks on LUMI (AMD CPUs) and CYFRONET's clusters (Ares with Intel CPUs and Helios with AMD CPUs) provide reference performance numbers that can now be used when running new simulations, testing new compilers, or entering a new machine.

2.11.5 Optimisation objectives

The primary optimisation objectives were to:

- Improve performance on LUMI and in general.
- Add performance metrics, such as the average number of particles processed per time step, to enable performance characterisation and comparison of different compilations, variants, and machines.

2.11.6 Code optimisation

The optimisation activity achieved the following:

- Implemented performance metrics that highlight the most important aspects of code performance and horizontal domain computational imbalance – see Figure 18.
- Benchmarked compilation flags found the correct flags for ifort and ifx on LUMI (AMD CPUs) and other clusters – see Figure 19.
- Fixed bugs in the OpenMP variant.
- Implemented a new domain-to-process assignment strategy, called "mirror strategy".
- Code refactored in a few places, better error handling for I/O operations.
- Implemented the command line interface for changing some of the simulation parameters, turning on/off debugging, performing dry runs.

Co-funded by the European Union
EuroHPC Joint Undertaking

- Refactored the modules with the two most compute intensive phases of the simulation, which were split into multiple files to allow mixed compilation with different optimisation flags for each file.

```
============================================================
FINAL PERFORMANCE STATISTICS   1000 steps  (3000-4000)
============================================================
data from all domains (32 = 4x8):
---
# PROCESSED PARTICLES =    19829.7 Mpart (all domains and steps) -- 19.83 | 19.83 | 19.83  (first|avg|last)-step
  particles per domain/step     0.620 [x1.00] first |   0.620 [x1.00] avg |   0.620 [x1.00] last
       > COL 01             --  1.015 [x1.64] first |   1.048 [x1.69] avg |   1.059 [x1.71] last
       > COL 02             --  0.496 [x0.80] first |   0.554 [x0.89] avg |   0.633 [x1.02] last
       > COL 03             --  0.485 [x0.78] first |   0.485 [x0.78] avg |   0.485 [x0.78] last
       > COL 04             --  0.482 [x0.78] first |   0.392 [x0.63] avg |   0.302 [x0.49] last
---
o TIME (without init) =     117.71 s (all steps)  ~>  .118 s/step
  time ditribution/step         23% mover (.027)  21% depos (.025)  56% other (.065)
       > COL 01             --  40% mover (.048)  37% depos (.044)  23% other (.026)
       > COL 02             --  20% mover (.024)  19% depos (.023)  61% other (.071)
       > COL 03             --  17% mover (.020)  16% depos (.019)  67% other (.078)
       > COL 04             --  14% mover (.017)  13% depos (.016)  73% other (.086)
---
* SIMULATION rate =         168.46 Mpart/s (all domains)  ~>  5.26 Mpart/s/domain  [x1.00 avg]  [x0.59 max]
       > COL 01             --  8.90 Mpart/s/domain  [x1.69 avg]  [x1.00 max]
       > COL 02             --  4.71 Mpart/s/domain  [x0.89 avg]  [x0.53 max]
       > COL 03             --  4.12 Mpart/s/domain  [x0.78 avg]  [x0.46 max]
       > COL 04             --  3.33 Mpart/s/domain  [x0.63 avg]  [x0.37 max]
============================================================
```

Figure 18 Performance from a test simulation on LUMI with32 CPUs and 1000 timesteps. The data includes the following results: total for the entire simulation, averaged for a single domain (one core), and averaged relative to the domain location (division of the computational grid into columns). (project 113)

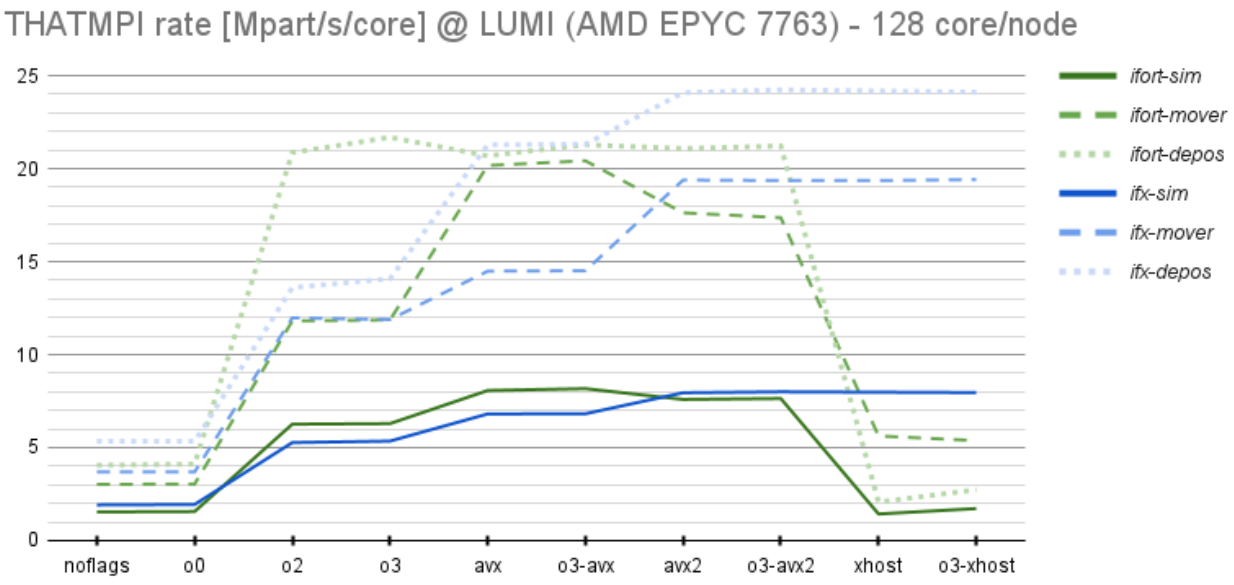


Figure 19 Results of the compiler optimisation flags benchmark on LUMI involving a single full CPU-node and 100 timesteps, the best rates from 3 runs ("sim" stands for total simulation rate, "mover" and "depos" are the two most compute intensive parts of the simulation). (project 113)

Co-funded by the European Union
EuroHPC Joint Undertaking

2.11.7 Improvements achieved

Firstly, the codebase now includes two benchmarks (compiler optimisation flags, OpenMP balancing) and reference performance numbers from multiple machines and CPU architectures (Intel Cascade Lake, AMD Zen 3, AMD Zen 4); it also contains many improvements in terms of usage (e.g. compiling multiple binary versions, dry-run mode, command line interface, performance metrics, etc). These improvements will simplify the future development and optimisation process.

Thanks to performance metrics and benchmarking (joint WP2 and WP3 effort), we were able to:

- discover that the `-xhost` option was responsible for poor performance when compiling with ifort for AMD CPUs (~4x slower) and speed up the simulation about 5.10x thanks to setting the right compilation options:
  - simulation rate (per domain) for base compilation (*-O3 -xhost*) – 1.03 Mpart/s,
  - simulation rate without -xhost option (*-O3*) – 4.02 Mpart/s,
  - simulation rate for optimized compilation (*-O3 -arch avx*) – 5.26 Mpart/s.

- see that different parts of the code may benefit differently from compiler optimisation flags and demonstrate on the Helios machine (AMD Zen 4) that compilation with mixed optimisation flags can provide an extra 1.05x speedup – see Table 15 (simulation rate 10.46 vs 9.99).

Table 15 Performance results of the benchmark on Helios involving a single full CPU-node (2x AMD EPYC 9654, 192-cores) and 100 timesteps. Comparison of mover and deposition rates when using different optimisation flags. Results from a single run.

Compiler optimisation flags	Simulation rate [Mpart/s/dom]	Mover time (share in sim. time)	Mover rate [Mpart/s/dom]	Deposition time (share in sim. time)	Deposition rate [Mpart/s/dom]
base: O3	**7.65**	55%	13.91	30%	25.50
base: O3 + AVX	**9.80**	40%	24.50	40%	24.50
base: O3 + AVX512	**9.99**	48%	20.81	33%	30.27
base: O3 depos: O3 + AVX512 mover: O3 + AVX	**10.46**	43%	24.33	36%	29.06

The effort focused on the OpenMP-based parts of the code allowed us to re-enable the feature of balancing the computations across CPUs assigned to different domains in the computational grid. On a test simulation involving 32 domains (4 columns, each of 8 domains), we demonstrated that it gave a ~1.28x speedup (compare Figure 18 with

Figure 20). In the first case, the job topology was 32 processes x 1 core/process. In the second case, it was 8 processes x 4 cores/process.

Co-funded by the European Union

```
============================================================
FINAL PERFORMANCE STATISTICS   1000 steps  (3000-4000)
============================================================
data from all domains (32 = 4x8):
---
# PROCESSED PARTICLES =    19829.7 Mpart (all domains and steps) -- 19.83 | 19.83 | 19.83  (first|avg|last)-step
  particles per domain/step     0.620 [x1.00] first |   0.620 [x1.00] avg |   0.620 [x1.00] last
       > COL 01             --  1.015 [x1.64] first |   1.048 [x1.69] avg |   1.059 [x1.71] last
       > COL 02             --  0.496 [x0.80] first |   0.554 [x0.89] avg |   0.633 [x1.02] last
       > COL 03             --  0.485 [x0.78] first |   0.485 [x0.78] avg |   0.485 [x0.78] last
       > COL 04             --  0.482 [x0.78] first |   0.392 [x0.63] avg |   0.302 [x0.49] last
---
o TIME (without init) =      91.66 s (all steps)  ~>  .092 s/step
  time ditribution/step         30% mover (.028)  27% depos (.025)  43% other (.038)
       > COL 01             --  30% mover (.028)  47% depos (.044)  23% other (.020)
       > COL 02             --  30% mover (.028)  24% depos (.023)  46% other (.041)
       > COL 03             --  31% mover (.029)  20% depos (.019)  49% other (.044)
       > COL 04             --  31% mover (.029)  17% depos (.016)  52% other (.047)
---
* SIMULATION rate =         216.35 Mpart/s (all domains)  ~> 6.76 Mpart/s/domain [x1.00 avg]  [x0.59 max]
       > COL 01             -- 11.43 Mpart/s/domain  [x1.69 avg]  [x1.00 max]
       > COL 02             --  6.05 Mpart/s/domain  [x0.89 avg]  [x0.53 max]
       > COL 03             --  5.29 Mpart/s/domain  [x0.78 avg]  [x0.46 max]
       > COL 04             --  4.28 Mpart/s/domain  [x0.63 avg]  [x0.37 max]
============================================================
```

Figure 20 Performance metrics output from a test simulation on LUMI involving 32 CPUs and 1000 timesteps, executed as 8 processes, 4 OpenMP-threads each. Now cores assigned to all domains spent the same amount of time in computing the mover phase. This results in a lower idle time in less occupied domains, which can be observed in time spent on "other" computations.

2.11.8 Summary and lessons learned

This project improved the code quality and functionality of the THATMPI astrophysics PIC code. Its performance can now be easily tested on multiple machines and binary versions, compiled with different optimisation flags. Thanks to selecting optimal compilation flags and re-enabling OpenMP balancing, significant speedups were observed, 6.5x on a sample test on LUMI. Moreover, the effort invested in this project will simplify the future optimisation and development process, as well as the choice of the right compilation parameters when landing on a new machine.

Key lessons learned:

1. Intel Fortran compilers can be successfully used for AMD CPUs, like in LUMI, but you must be careful:
    - Do not trust the *-xhost* option! It seems that for ifort it blocks standard optimisations, but for ifx it seems to be fine (however, you never know how it will behave in the future).
    - The new Intel compiler, ifx, can achieve similar performance to ifort, when used with the best optimisation options; for standard optimisations, ifort seems to be better; it is interesting that one compiler is better in optimizing for AVX and the other for AVX2.
    - Options like `-xAVX` work only on Intel CPUs, so instead use `-arch`.
2. Higher optimisation levels (AVX, AVX2, AVX512) do not always imply better performance. Different functions may benefit from different sets of options, so always benchmark the performance on a given machine.

Co-funded by the European Union
EuroHPC Joint Undertaking

3. Do not use PDTs if you want to develop a portable Fortran code. It appears that currently, only Intel compilers provide reasonable support; the new LLVM Flang may extend support for this feature in the future.

The project has been completed, but we see that work on THATMPI can be continued. The code has potential for further optimisation, and a more detailed performance analysis should be applied. We suggest that porting the code to other compilers should not be a priority at this stage, but thanks to the built-in performance indicators, if someone decides to continue this work, it will now be easy to assess whether a compiler switch gives performance gains or not.

2.12 Project 128: Past Stall Operation of Hydrogen Multistage Radial Compressors (Lead: CINECA, Collaborators: IT4I@VSB)

2.12.1 Project overview

The goal was to investigate the operation of multistage radial compressors to distribute, store, and pressurize H_2 in heavy past-stall off-design conditions and to determine the risk of insurgence of low frequency pulsations, build-up of temperature, and the associated mechanical risks. The study was done by means of CFD simulations using the TRAF code.

2.12.2 HPC systems used in the project

The Leonardo Booster partition was used throughout the project.

2.12.3 Code description

TRAF is a CFD solver for three-dimensional Reynolds-averaged Navier-Stokes equations. It is written in Fortran and parallelized with an MPI+OpenMP scheme. It runs on NVIDIA GPUs thanks to a mixed CUDA-Fortran+OpenACC port. Notably, it heavily exploits multi-streaming with each OpenMP thread dispatching asynchronous OpenACC kernels.

2.12.4 Status before optimisation

The original support request concerned the best setup to be used on Leonardo in terms of thread and process binding, plus an analysis of inter-process communication. While performing these tasks, it was found that a single GPU kernel accounted for roughly 44% of the compute time, hence it was agreed with the PI to see whether the kernel could be further optimized. The kernel, consisting of the inversion of a tridiagonal system of floats for multiple right-hand sides with the well-known Thomas algorithm, will be referred to as the ‘smooth kernel’ in the following, being the core of the implicit residual smoothing step (along the x direction) in multi-grid methods. Considering that

Co-funded by the European Union
EuroHPC Joint Undertaking

1. The relative impact of the smooth kernel varies according to input grid size.
2. Even during a single optimisation step, the smooth kernel is called by the same thread with quite different input grid sizes.
3. Several concurrent smooth kernels are present at the same time on the same GPU, due to the asynchronous kernel spawning of each OpenMP thread.

it was agreed with the PI to work on two different benchmarks, a medium one (M) and a large one (L). Moreover, two metrics were used to characterize the optimisation:

- Metric 1: the kernel duration, averaged over all the OpenMP threads and MPI tasks.
- Metric 2: the internal 'Main loops' clock, which corresponds to the walltime of the simulation minus initialization and I/O time.

While metric 2 is the most relevant for what concerns the end user, metric 1 was necessary to assess the quality of local optimisations of the kernel, which could otherwise be washed out by intra-GPU or inter-process synchronization effects. The baseline was hence characterized as in Table 16

Table 16 Baseline characterization for the optimisation of the smooth kernel. The main loops clock corresponds to the total walltime for 10 optimisation steps minus initialization and I/O, and it is natively provided by the TRAF application in its log file. The average kernel time was extracted by profiling all the smooth kernels occurring over 3 optimisation steps with Nsight Compute and averaging over all the Duration values. The GPU compute time is the one provided by Nsight System, profiling as well 3 optimisation steps. Being this last value per MPI process, the min and max have been reported.

Benchmark size	Main loops [s]	Average kernel time [ms]	GPU compute time [%]
M	92.90	4.91	43-46
L	70.8	17.1	34-38

We note that since each MPI process has a slightly different workload, the min-max range is reported in the GPU compute time. In addition, the M benchmark was run on 3 nodes (12 custom A100 NVIDIA GPUs) of Leonardo Booster, while the L benchmark on 16 nodes (64 GPUs) of the same partition, which explains the lower walltime with a longer kernel duration. In both cases 8 OpenMP threads per MPI process was used.

2.12.5 Optimisation objectives

The objective was to reduce the Main loops walltime by optimizing the main bottleneck, i.e. the smooth kernel computation on the GPU.

2.12.6 Code optimisation

The smooth kernel consists of four nested loops, ported to GPUs with an OpenACC collapse clause over the two outermost loops. Optimisation was attempted with two strategies:

Co-funded by the European Union
EuroHPC Joint Undertaking

1. Enforcing a small CUDA block size via the `vector_length` clause, since a large fraction of the smooth kernels are called on quite small workloads, from ~40 to ~200 warps (1 warp=32 CUDA threads). In these cases, a smaller CUDA block size (with respect to the usual size of 128 CUDA threads per block) allows for a more efficient utilization of the GPU resources, with 64 CUDA threads normally being the optimal compromise between the warp scheduling cost and the streaming multiprocessor (SM) utilization.
2. Improving memory access (MA) patterns, by removing strided accesses and by making CUDA threads of the same warp access contiguous memory addresses.

The results of these optimisations are reported in Table 17.

Table 17 Results of the optimisation performed for the smooth kernel compared to the baseline for the medium (M) and large (L) benchmarks. Each optimisation is characterized by the improvement of memory accesses (MA) with respect to the baseline, and by the enforced CUDA block size (in terms of CUDA threads). The baselines have no improved memory access pattern, and a default CUDA block size of 128 CUDA threads

Benchmark Size	Kernel version (MA opt., CUDA block)	Main loops [s] (wrt baseline)	Average kernel time [ms] (wrt baseline)
M	Optimized (yes, 32)	92.69 (-0.2%)	4.60 (-6.3%)
M	Optimized (yes, 64)	92.21 (-0.7%)	4.60 (-6.3%)
M	Optimized (yes, 128)	92.40 (-0.5%)	4.68 (-4.7%)
M	Optimized (yes, 256)	93.08 (+0.2%)	4.92 (+0.2%)
M	**Baseline (no, 128)**	**92.90**	**4.91**
L	Optimized (yes, 32)	67.2 (-5.1%)	15.7 (-8.2%)
L	Optimized (yes, 64)	67.0 (-5.4%)	15.5 (-9.4%)
L	Optimized (yes, 128)	67.3 (-4.9%)	16.3 (-4.7%)
L	Optimized (yes, 256)	69.8 (-1.4%)	17.0 (-0.6%)
L	**Baseline (no, 128)**	**70.8**	**17.1**

2.12.7 Improvements achieved

From the results reported in Table 17 the following conclusions can be drawn:

1. Smaller CUDA block sizes are beneficial as expected, with little difference from the 32 to the 64 CUDA thread case.
2. Memory access optimisation is overall effective.

Co-funded by the European Union
EuroHPC Joint Undertaking

3. The 5% to 10% improvement found in the average kernel time is not reflected in the loops' walltime. This lack of correlation is mostly due to synchronization effects, for both intra-GPU and inter-process.

From the reasons above, it was chosen to include only the memory access (MA) optimisation, since fixing a CUDA block size reduces portability and did not bring any sizable improvement to the Main loops clock.

The result achieved was then a ~5% improvement in the time-to-solution for the large benchmark (representative a real production-like case) thanks to optimized memory access patterns. This improvement is less effective for smaller workload (~0.5% gain in medium-sized benchmarks) where kernel scheduling and inter-process communications are dominant.

2.12.8 Summary and lessons learned

For the 5% improvement for the large benchmark target, the proposed optimisations are less effective when reducing the system size, when inter-process communication and kernel concurrency on the same GPU become dominant. In those cases, it is suggested to carefully inspect and tune the number of OpenMP processes, which are related to the degree of concurrency on the GPU.

The project was quite challenging, due to a non-standard GPU-porting style where multiple small OpenACC kernels are scheduled asynchronously via host-side OpenMP threads. This kind of porting identifies a niche where the normally excellent NVIDIA profilers fall short, i.e. the analysis of several, inhomogeneous versions of the same kernel spawned concurrently.

2.13 Project 133: Scaling Deep Neural Networks for Public Safety: Fire, Flood etc. with HPC (Lead: BSC)

2.13.1 Project overview

The project aimed to use real-time image segmentation techniques in Deep Learning to detect natural disasters such as floods and fires. The idea was to create deployable AI solutions based on Computer Vision that utilize standard video cameras to create wide area monitoring systems.

2.13.2 HPC systems used in the project

The optimisations were applied for the accelerated partition of MareNostrum 5

2.13.3 Code description

The software development centred on the creation and training of Machine Learning models, encompassing architectures like U-Net, DeepLab, and YOLO. The implementation leverages

Co-funded by the European Union
EuroHPC Joint Undertaking

both TensorFlow and PyTorch frameworks, providing seamless support for multi-platform environments, including CPUs, single GPUs, and multiple GPUs. This makes it particularly suitable for parameter-heavy neural networks such as ResNet-152 and ResNeXt101. For distributed execution, TensorFlow employs the tf.distribute.MirroredStrategy to replicate the model and partition input data across the available GPUs. PyTorch, conversely, utilizes torch.nn.DataParallel for model replication, data splitting, and synchronization management. These strategies facilitate the handling of large batch sizes and high-resolution imagery. Robustness is enhanced through implemented model checkpointing, which enables simulations to resume from the last successful checkpoint following a system failure.

2.13.4 Status before optimisation

The developers were facing GPU memory errors during the first epoch with larger batch sizes. They needed advice on obtaining profiling information, handling large batch sizes, optimal settings for DataLoaders, and mixed precision training. The developers did not know which partition of MareNostrum 5 was better suited for such loads and were not satisfied with file I/O optimisation on storage systems. The code further did not support multi-GPU multi-node training.

2.13.5 Optimisation objectives

The optimisation objectives were to:

1. address the OOM (Out Of Memory) error for larger batches
2. obtain profiling information such as GPU utilization and power dissipated (by extracting energy usage)
3. suggest the correct MareNostrum 5 partition
4. enhance the I/O speed
5. enable Multi-Node Multi-GPU training (MNMG).

2.13.6 Code optimisation

Yolo did not support Multi Node Multi GPU (MNMG) execution and the Yolo codebase had to be substantially modified in order to enable MNMG executions. Specifically, the codebase incorrectly used the RANK variable instead of LOCAL_RANK for device assignment, and its internal DDP initialization conflicted with user-defined distributed setup. We also modified the source code of the application to accommodate the changes in the Yolo codebase. After modifications, in the multi-node multi-GPU (MNMG) setup, all GPUs were actively utilized however the performance and GPU utilization remained low.

To address this, several modifications were made to the YOLOv8 codebase so that:

- LOCAL_RANK is used for GPU assignment (see Figure 21)
- WORLD_SIZE is handled correctly.
- YOLO's internal DDP setup does not conflict with the DDP setup in the source code.

Co-funded by the European Union
EuroHPC Joint Undertaking

```python
def _setup_ddp(self, world_size):
    """Initialize and set the DistributedDataParallel parameters for traininng"""
    torch.cuda.set_device(RANK)
    self.deice = torch.device(f"cuda", RANK)

    os.environ["TORCH_NCCL_BLOCKING_WAIT"] = "1"

    dist.init_process_group(
        backend="nccl" if dist.is_nccl_available() else "gloo",
        timeout=timedelta(seconds=108000), # 3 hours
        rank=RANK,
        world_size=world_size,
    )
```

Figure 21 Incorrect passing of RANK argument instead of LOCAL_RANK argument in YOLOv8 codebase (project 133)

These changes were applied by modifying the Ultralytics container and creating a new image (mnmg_ultralytics-8.3.0.sif) with MNMG support.

Additional simplifications were made to the training script and SLURM configuration:

- Distributed initialization and cleanup were left entirely to YOLO.
- Unnecessary TORCH and NCCL environment variables were removed.
- Device assignment was handled automatically by YOLO.
- The global batch size was increased to 64, and data loader workers were reduced for stability.

After these changes, stable MNMG training was achieved, however the utilization across multiple nodes remained low (typically between 16% and 59%).

2.13.7 Improvements achieved

We were able to give insights into the OOM error with a batch size of 64 and were also able to increase the global batch size to 512 with DDP obtaining a good scalability which was near-linear at the beginning when adding more nodes (see **Error! Reference source not found.**). The PI and researchers of the EuroHPC project were provided with a SLURM script that provided profiles of execution with NVIDIA nsys.

Co-funded by the European Union
EuroHPC Joint Undertaking

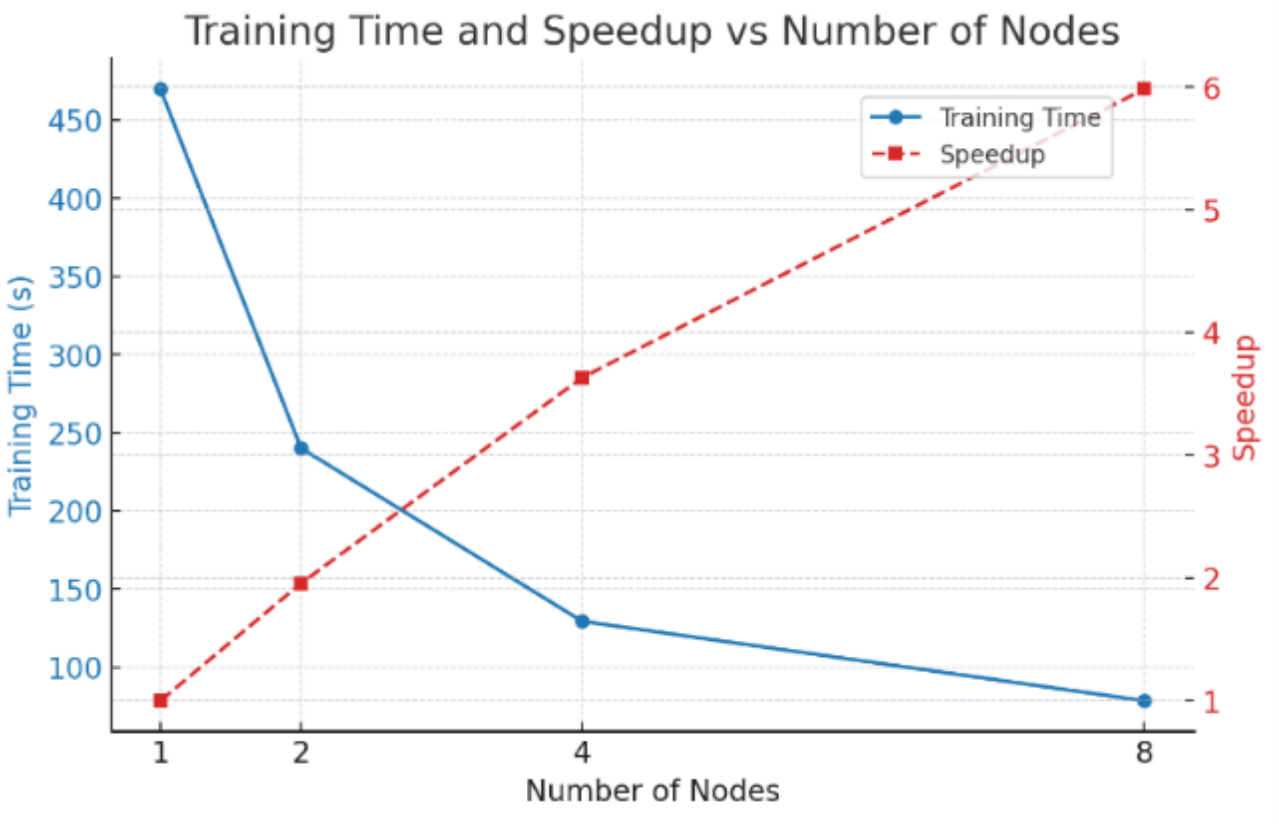


Figure 22 Scalability speedup when using multi-node multi-GPU setups (project 133)

Multiple changes to the codebase of YOLOv8, the source code, and the SLURM script to enable Multi-Node Multi-GPU execution in YOLOv8 were made but when we executed the program, we noticed that adding `torchrun` negatively affected the performance of the executions (see Table 18).

Table 18 multi-node + multi-GPU execution of YOLOv8 after YOLOv8 code-base modification

Nodes	Total GPUs	Batch / GPU	Total batch size	Mem / GPU (GB)	Epochs / 30 min	Yolo Version	torchrun
1	4	60	240	64 GB	14	Unmodified	No
1	4	60	240	64 GB	14	Modified	No
1	4	60	240	64 GB	2	Modified	Yes
1	4	50	200	44 GB	2	Modified	Yes
1	4	40	160	44 GB	2	Modified	Yes
2	8	60	480	64 GB	4	Modified	Yes
2	8	50	400	55 GB	4	Modified	Yes
2	8	40	320	44 GB	4	Modified	Yes
4	16	60	960	64 GB	8	Modified	Yes
4	16	50	800	55 GB	8	Modified	Yes
4	16	40	640	44 GB	8	Modified	Yes

Table 18 shows the unmodified (non-MNMG) and modified (MNMG) execution of YOLOv8 on single and multiple nodes. It can be seen that the modified version yields the same number of Epoch / 30 minutes as the original version when running on a single node with 4 GPUs. However, as soon as

Co-funded by the European Union
EuroHPC Joint Undertaking

torchrun is added to the execution command line, the number of Epochs decreases to 2. Furthermore, adding more nodes (and more GPUs) seems to alleviate the number of Epoch / 30 minutes but still 4 nodes are insufficient to equal the performance of a single node. With 16 nodes (not shown) we were able to obtain more Epochs / 30 minutes as compared to a single node, but we note that this is not a good use of HPC resources. Thus, our recommendation was to use a single training node only.

2.13.8 Summary and lessons learned

To support the primary goal of increasing batch size and scaling training workloads, we strongly recommend using DDP. This strategy enables effective scaling across multiple nodes, allowing significantly larger global batch sizes by distributing the workload efficiently across GPUs. While FSDP and DeepSpeed offer memory optimisations, DDP remains the most reliable strategy for leveraging more nodes to train larger datasets at scale.

YOLOv8 does not support Multi-node Multi-GPU (MNMG) execution out of the box. We were able to modify the codebase of YOLOv8 to achieve MNMG execution. However, it was found that the efficiency of MNMG executions was quite low. More investigation is needed to determine the exact cause of performance degradation.

2.14 Project 137: Distributed LLMs on ARM (Lead: INESC TEC)

2.14.1 Project overview

This project focuses on enabling and optimizing distributed inference of Large Language Models (LLMs) on ARM-based HPC architectures. The work targets the vLLM inference engine running on Deucalion's CPU-only nodes, aiming to extend its capabilities beyond single-node CPU execution by enabling multi-node execution using Ray. The project also aims to further improve scalability, throughput, memory efficiency, and inter-node communication performance. Alternative orchestration frameworks to Ray should be evaluated to ensure suitability for ARM-based distributed LLM inference.

2.14.2 HPC systems used in the project

Deucalion (ARM CPU-only nodes) was used for the project.

2.14.3 Code description

The primary software components utilized in this project are vLLM, a Python-based high-performance inference engine for LLMs employing `PagedAttention`, and Ray, a distributed orchestration and parallel execution framework that manages workers and coordinates parallel

Co-funded by the European Union
EuroHPC Joint Undertaking

inference. Initially, vLLM only supported single-CPU node execution, as well as single-GPU and multi-GPU execution, but not multi-CPU-node execution. Parallelization methods explored include pipeline, tensor, and hybrid parallelism, adapted for multi-node ARM CPU environments.

2.14.4 Status before optimisation

Before EPICURE support, multi-CPU-node distributed execution was not yet available in vLLM. As a result, no baseline performance data existed for multi-node CPU inference because the feature was not operational, and ARM-specific distributed execution configurations for Ray and vLLM had not yet been developed.

WP2 support enabled proper installation of vLLM and Ray on Deucalion ARM nodes and established a functional single-node baseline, which was necessary before enabling multi-node distributed inference.

2.14.5 Optimisation objectives

The primary optimisation objectives were:

- Enable multi-CPU-node distributed inference.
- Improve scalability across multiple ARM CPU nodes.
- Reduce latency and communication overhead.
- Optimize memory usage to allow larger workloads and reduce KV-cache pressure.
- Benchmark and evaluate alternatives to Ray.

2.14.6 Code optimisation

Optimisation activities progressed in several stages:

- Resolving Ray configuration issues preventing worker startup on multiple ARM nodes.
- Adapting vLLM's parallelism mechanisms for ARM, ensuring compatibility with the CPU execution backend.
- Validating successful worker initialization, object store setup, and inter-node communication.
- Reducing serialization overhead affecting communication between nodes.
- Evaluating the limitations of multiprocessing-based intra-node execution and transitioning to Ray communicators to achieve more efficient task coordination.
- Testing combinations of pipeline and tensor parallelism.
- Optimizing parallel partitioning to reduce inter-node tensor communication volume.
- Monitoring CPU and memory utilization, identifying inefficiencies in task distribution.
- Measuring communication overhead, as it is the main scalability barrier.

Co-funded by the European Union
EuroHPC Joint Undertaking

Table 19 Performance of vLLM distributed inference before and after optimisation on Deucalion ARM CPU nodes. All the tests were obtained using the Llama-3.1-8B model and 4 ARM nodes

Configuration	Generation Throughput Before Optimisation	Generation Throughput After Optimisation
Pipeline + Tensor Parallelism	≈3 tokens/s	≈7 tokens/s
Pipeline Parallelism only	≈2 tokens/s	≈5 tokens/s
Tensor Parallelism only	≈4 tokens/s	≈11 tokens/s

2.14.7 Improvements achieved

The optimisation work produced measurable improvements to the overall distributed inference pipeline on Deucalion's ARM CPU nodes, since vLLM now runs successfully across multiple ARM CPU nodes.

Beyond enabling functionality, the performance of the full inference workflow improved significantly across all tested parallelism configurations. Considering end-to-end throughput on 4 ARM nodes, the optimisation resulted in:

- **Pipeline + Tensor Parallelism:** from ≈3 tokens/s → ≈7 tokens/s (**~2.3× speedup**)
- **Pipeline Parallelism only:** from ≈2 tokens/s → ≈5 tokens/s (**~2.5× speedup**)
- **Tensor Parallelism only:** from ≈4 tokens/s → ≈11 tokens/s (**~2.75× speedup**)

Overall, the distributed inference pipeline experienced a 2–3× improvement in generation throughput, depending on configuration. This demonstrates that optimisation efforts improved both functionality (multi-node execution) and performance (speedups of the entire pipeline).

2.14.8 Summary and lessons learned

This project transformed vLLM execution on ARM CPU nodes from a single-node-only setup into a fully operational multi-node distributed inference workflow. The optimisation efforts resolved Ray initialization issues, improved communication efficiency, and enabled stable parallel execution across 4-node configurations. Performance gains of 2 to 3 times were achieved, demonstrating meaningful acceleration of the entire inference pipeline.

Key lessons learned include:

- Enabling multi-node CPU execution requires careful orchestration tuning (Ray's default behavior introduces overhead).
- Parallelization strategies must be adapted to architecture (tensor parallelism benefited most from optimisation).

Co-funded by the European Union
EuroHPC Joint Undertaking

- Identifying bottlenecks in inter-node messaging and worker scheduling guided most of the performance gains.
- ARM architectures can support distributed LLM inference.

The work described in this report represents an intermediate milestone of an ongoing project. Further development, tuning, and validation activities will continue over the coming months to consolidate multi-node ARM performance, explore additional orchestration frameworks, and extend support to larger models.

Co-funded by the European Union
EuroHPC Joint Undertaking

3. Conclusions

In the second year of the EPICURE project, we have described 13 projects for which level 3 advanced optimisation was carried out. For this report particular emphasis was placed on the improvements achieved for each project by comparing the status of the application codes before and after optimisation. Before describing the improvements, it is worth highlighting two of the trends observed:

- A strong predominance of AI/ML workloads, where many users lack experience with multi-node distributed training on HPC clusters (vs. cloud setups).
- Frequent requests for efficient GPU/accelerator porting (NVIDIA/AMD) and optimized use of Python frameworks (PyTorch, JAX, Dask, etc.).

Given the sustained interest in AI and the popularity of Python it is certain that these two trends will be replicated in 2026.

The improvements achieved by the EPICURE ASTs for each project are summarised in Table 20. From the table we can give some rough statistics on the optimisations achieved in the second year of the project:

- In more than two-thirds of the projects a performance speedup of at least 2x was obtained (much more in some cases)
- Porting to NVIDIA or AMD GPUs was successful in about a third of the projects.
- Code improvements in two projects allowed larger model and batch sizes and with reduced training times.
- In about half the projects the code was refactored which, as well as usually allowing performance increases, also resulted in codes more suitable for future optimisations.

Finally, we emphasise that the optimisations reported do not just benefit the EuroHPC projects – the knowledge, auxiliary codes and experience gained by the ASTs is being used for future EPICURE work and being disseminated to the wider HPC community.

Co-funded by the European Union
EuroHPC Joint Undertaking

EPICURE Project no.	Key Benefits	Maximum performance gain	Improved parallel scaling	Porting to new architecture	Code improvements (incl. fixing errors and code refactoring)
32	Reduced training time for larger phage-host datasets	2x			
34	Enabled multi-node training of generative AI models on 3D MRI	6x	X		
50	Reduced memory requirements → allowed simulation of larger systems	3x			
57	Performance increase and porting	1.2x		GPU	
89	Run time reduction from days to minutes	> 100x	X		X
99	Re-factored code for further improvements	1.05			X
107	high speedup on HIP → made GPU acceleration viable for production	2-3x (GPU), 150x (HIP)		GPU	X
108	Performance increase	13x		GPU	
109	Performance increase and porting	11x		GPU	
113	Performance increase	>5x			X
128	Small performance increase	1.05			
133	Reduced memory allowing higher batch sizes, also multi-node runs enabled		X	GPU	X
137	Scaling and performance improvements	2-3x	X		X

Table 20 Summary of improvements resulting from EPICURE optimisation activities in 2025

Co-funded by the European Union
EuroHPC Joint Undertaking