



SCALABLE PARALLEL ASTROPHYSICAL CODES FOR EXASCALE

Performance profiling and benchmarking (Update)

Deliverable number: D2.6

Version 1.0/1.0



Co-funded by
the European Union



EuroHPC
Joint Undertaking

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Belgium, Czech Republic, France, Germany, Greece, Italy, Norway, and Spain under grant agreement No 101093441

Project Information

Project Acronym: SPACE
Project Full Title: Scalable Parallel Astrophysical Codes for Exascale
Call: Horizon-EuroHPC-JU-2021-COE-01
Grant Number: 101093441
Project URL: <https://space-coe.eu>

Document Information

Editor:	Georgios Doulis – GUF
Deliverable nature:	Report (R)
Dissemination level:	Public (PU)
Contractual Delivery Date:	30.06.2025
Actual Delivery Date	30.06.2025
Number of pages:	87
Keywords:	performance analysis, POP methodology, region analysis
Authors:	Andrea Mignone, Marco Rossazza, Stefano Truzzi – UniTo Lubomir Riha, Radim Vavrik, Joao Barbosa – IT4I@VSB Tine Colman, Benoît Commerçon – CRAL CNRS Geray Karademir – LMU Khalil Pierre, Georgios Doulis – GUF Robert Wissing – UiO Luca Tornatore – INAF Pranab J Deka – KU Leuven R.-Paul Wilhelm – KU Leuven Manolis Ploumidis, Giorgos Katevenis – FORTH Gino Perna – ENGINSOFT
Peer review:	Nitin Shukla – CINECA Elisabetta Boella – E4 Manolis Marazakis – FORTH

History of Changes

Release	Date	Author, Organization	Description of changes
0.1	23.01.2025	Eva Sciacca, INAF	Started the document and setup of the document structure from D2.1.
0.2	1.4.2025	Lubomir Riha, IT4I@VSB	Defined the guidelines and templates for individual sections.
0.8	10.6.2025	Science and technology partners	Individual code contributions.
0.9	13.6.2025	Georgios Doulis, GUF	Version ready for internal review.
1.0	30.6.2025	Georgios Doulis, GUF	Final version.

Scalable Parallel Astrophysical Codes for Exascale

DISCLAIMER

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European High Performance Computing Joint Undertaking (JU) and Belgium, Czech Republic, France, Germany, Greece, Italy, Norway, and Spain. Neither the European Union nor the granting authority can be held responsible for them.



The space above and below the message intentionally is left blank.

Executive Summary

This deliverable reports on profiling and benchmarking activities related to the development and optimization of the SPACE codes. Each section is devoted to one code and contains two parts:

Part 1: Benchmarking and Data Collection for Internal Archive

The first component of each section focuses on benchmarking selected scientific codes at different stages of their development. Specifically, performance measurements are conducted for two key commits: the initial version at the project inception (M1) and the current version (M30).

By employing representative use-cases from WP1, each code is benchmarked at scaled-up configurations, evaluating both the overall application runtime and the performance of specific computational kernels or modules identified in D2.1. These measurements aim to demonstrate optimization progress, particularly in code regions where optimization was performed.

For all codes, we run scaling tests up to 64 nodes, to show how the performance improves at scales, as inter-node optimization is one of the goals of the projects. A very brief summary of the results reported in this deliverable is:

- **Pluto** achieved speedup for the entire code up to 1.52 and up to 2.5 - 5x for the Boundary region,
- **RAMSES** achieved speedup for the entire code up to 1.55 (combined the two results) and a similar speedup for the Poisson multigrid solver,
- **ChaNGa** on CPU achieved speedup for the entire code up to 1.65 and up to 2.2 speedup for the Gravity-Hydro region. Also, a speedup of up to 3.4 is achieved by the NVIDIA A100 GPU over the AMD EPYC 7763 CPU,
- **iPIC3D** achieved a speedup of up to 1.37 for the entire code, and a speedup of 4.0 for the Field solver region,
- **BHAC** achieved a speedup of its critical region *Riemann solver* up to 3.3 using GPU acceleration when comparing 1 CPU chip to 1 GPU chip. Keep in mind that the GPU implementation of BHAC is still recent, and most of the effort has been devoted to functionality porting to OpenACC rather than optimization.
- **OpenGadget3** shows speedup up to 1.15 for 64 nodes for the entire code and over 4 for *Domain decomposition* region. Two regions actually show a slowdown, which is related to more processing needed for increased accuracy in physical outputs.
- **FIL** is in the relatively early stage of switching from the Einstein toolkit to GRACE. Therefore, we were not able to measure identical benchmarks running a full scientific case. However, an initial benchmarking performance shows significant speedup achieved by porting to AMD GPUs (MI100 at IT4Innovations complementary systems and MI250 on LUMI). These initial results are presented in Section 7.1.

The common point for all tests is that the best speedups are, in general, achieved for 64 compute nodes, which confirms that the scalability of codes has also significantly improved. We have also used the measurements for this deliverable to further extend the internal data archive of the project with new sets of data.

All benchmarking input data, configuration files, logs, and output metrics are stored in the Internal Data Archive, ensuring reproducibility and traceability.

Part 2: Profiling Activities

This part of each code section focuses on profiling activities aimed at analyzing and improving the performance of the codes across CPU and GPU architectures. In collaboration with the POP3 Centre of Excellence, several audits have been completed, covering mostly CPU versions of the codes.

The most significant profiling is conducted internally by project partners, including detailed runtime analysis, kernel-level evaluation, and architectural comparisons (e.g., MPI, OpenMP, GPU). These efforts help identify performance bottlenecks and guide optimizations, especially in newly optimized code sections.

The performance analysis effort, as reported in this deliverable, points to a set of seven guidelines with broader applicability in parallel scientific computing:

-
1. Overall approach to incremental implementation improvements.
 2. Continuous effort on profiling.
 3. Effort on compiler optimization, esp. vectorization.
 4. Communication enhancements.
 5. Code restructuring to improve modularity and memory access patterns.
 6. GPU-specific optimizations.
 7. Making use of platform-specific functionality, particularly for GPU devices.

We elaborate on each of the guidelines in [Section 9](#).

Contents

1	Introduction	8
2	PLUTO	9
2.1	Use-case description	9
2.2	Benchmarking results	9
2.3	Profiling activities	11
2.4	Status of OpenMP offloading activity	11
2.5	Conclusions	13
3	OpenGadget3	16
3.1	Benchmarking results	16
3.2	Profiling activities	17
3.3	Code Refactoring	21
3.4	OpenMP offloading	26
3.5	Conclusion	26
4	iPIC3D	27
4.1	Code Structure	27
4.2	Use-case description	28
4.3	Scaling of IMM and ECSIM	30
4.4	Comparison to initial version	34
4.5	Profiling activities	34
4.6	Conclusion	36
5	RAMSES	37
5.1	Use-case description	37
5.2	Compiler options for the reference commit	38
5.3	Benchmarking results: OpenMP implementation	38
5.4	Benchmarking results: cpu optimisations	39
5.5	Profiling activities	39
5.6	Conclusion	44
6	BHAC	45
6.1	Use-case description	45
6.2	Benchmarking results	46
6.3	Profiling activities	48
6.4	Conclusion	56
7	GRACE/FIL	58
7.1	Benchmarking results	58
7.2	Profiling activities	60
7.3	Conclusion	66
8	ChaNGa	67
8.1	Use-case description	67
8.2	Benchmarking results	68
8.3	Profiling activities	69
8.4	GPU and MPI Profiling activities	70
8.5	Conclusion	79
9	Conclusions	80
9.1	Benchmarking	80
9.2	Profiling	83

List of Figures

1	Diagram of the Reconstruct-solve-average (RSA) strategy	9
2	Nsight Systems timeline showing kernel execution and memory events for a representative gPLUTO loop.	12
3	Nsight Compute roofline plot for the Reconstruct kernel. The plot shows arithmetic intensity and performance relative to hardware throughput limits.	12
4	Nsight Compute memory chart for the Reconstruct kernel. Memory load/store transactions are broken down by cache level and memory space.	13
5	Rocprofv2 kernel activity trace for a representative time step in gPLUTO. The timeline shows GPU kernel execution and memory transfer events, identifying hotspots and synchronization delays.	14
6	OmniPerf-generated Roofline plot for the ten most computationally intensive kernels in gPLUTO. The diagram illustrates achieved performance in terms of arithmetic intensity and identifies compute-bound and memory-bound regions relative to hardware ceilings, as well as the legend for the ten most time consuming kernels considered.	14
7	Strong scaling on LUMI for a recent code version (blue) and one before the start of the CoE (orange).	17
8	OPENGADGET3 on Leonardo DCGP for up to 16 nodes - strong scaling.	19
9	OPENGADGET3 on Leonardo DCGP for up to 16 nodes - efficiency metrics [%]. Note the Computation scalability submetrics were unavailable due to incompatible tool suite components.	19
10	The absolute error (y -axis) of accelerations due to Gravity, in code units. The plot shows the absolute difference between the acceleration computed by the B&H algorithm and that computed via direct summation as a function of the θ opening angle (x -axis; $\theta = 0$ coincides with the direct summation since all tree nodes are “open” during the tree walk down to the leaves). The red dashed line refers to the original implementation in Gadget while the blue line to the results obtained by the new implementation described in this section when the leaves contain only one particle instead of a bunch of particles. Green and Yellow lines ref to different number of particles being included, in average, in a single Box-Leaf.	23
11	Speedup time of Karras binary tree and octree build.	24
12	OPENGADGET3 scaling of BoxLeaves construction on a LEONARDO-DCGP’s node for 512^3 (512 <i>Million</i>) particles, with 8 and 64 points per BoxLeaf (green and yellow lines, respectively), and for two different particle distributions: homogeneous (continuous lines) and strongly clustered (dashed lines; about 90% of particles into very concentrated knots).	25
13	Scalability of OpenMP offloading in comparison with the CPU version.	26
14	Flow-chart of the Particle-In-Cell (PIC) method.	27
15	Scaling of IMM in iPIC3D on LUMI with a 2D case	31
16	Scaling of IMM in iPIC3D on Karolina with a 2d case	31
17	Scaling of IMM in iPIC3D on Leonardo with a 2d case	32
18	Scaling of the ECSIM on LUMI with a 2D case.	32
19	Scaling of the ECSIM on LUMI with a 3D case.	33
20	Scaling of a 2D simulation with ECSIM on Karolina.	34
21	iPIC3D on Leonardo DCGP for up to 64 nodes - strong scaling.	35
22	iPIC3D on Leonardo DCGP for up to 64 nodes - Efficiency metrics [%].	36
23	Call tree for the Godunov solver in RAMSES, for the Sedov use case. Between parenthesis, we note the percentage of the execution time that is spend in each subroutine (excluding calls to other subroutines), as measured by MAQAO on MeluXina. The boxes with dashed contours are only used when the AMR is activated.	41
24	MAQAO profiling results for the cosmo use-case run on half a MeluXina node. Top: an overview of the most time-consuming subroutines. Bottom: vectorization rate of the main loops.	43
25	Distribution of time spent in CUDA Runtime API calls (Rank 0)	50
26	Top kernels by total execution time (Rank 0)	51
27	Kernel occupancy histogram (Rank 0)	52
28	Kernel bottleneck distribution (Rank 0)	52
29	GPU memory usage over time	53
30	MPI communication matrix showing data exchange between ranks	54

31	Comparison of cell updates per second during evolution of GRACE and FIL on the IT4I p03 partition and the LUMI-G partition.	59
32	Comparison of percentage of time spent in Kernel 1 (Add fluxes to RHS) and Kernel 2 (conservative to primitive solver) for FIL and GRACE. This data was produced when FIL was run on a single processor on the p03 partition of the IT4I CS and a single M250 GPU on LUMI-G.	59
33	FIL-HOC performance variability: unigrid and AMR strategies	61
34	FIL-HOC: Evolution time across different iterations of different runs	61
35	FIL-HOC: CCTK_EVOL load imbalance	62
36	XHC Allreduce on a single Karolina node	63
37	Single-node collectives latency	63
38	XHC Allreduce on a single Karolina node, 4 cores idle on each socket	64
39	FIL-HOC: Weak scaling up to 256x Karolina compute nodes	65
40	FIL-HOC: MPI+OpenMP weak scaling	66
41	The difference in local gravity time from having GPU kernel launches tied to the TreePieces (TP) compared to consolidated kernel launches being handled by the data manager. Performance shows a strong dependence on the number of TPs when launching from TPs, whereas the new implementation exhibits no significant scaling impact.	70
42	GPU scaling tests for the cosmological box case performed on the Karolina supercomputer (details provided in deliverable D1.1 and D2.1). The plot compares three metrics: total runtime (GPU), runtime excluding initialization (GPU-NOINIT), and gravity module execution time (GPU-GRAV). Speed-up is normalized to 1 at 4 nodes (32 GPUs), with relative speed-up shown on the left and parallel efficiency on the right. We see a high parallel efficiency of above 80% up to 128 GPUs for the entire runtime and up to 256 GPUs for the gravity module.	71
43	Inter-rank communication matrix showing data transfer volumes between ChaNGa ranks. The asymmetric pattern reflects the tree-walking algorithm's dynamic load distribution.	73
44	Execution summary showing the distribution of computational work across different GPU kernels for Rank 0	74
45	Comprehensive kernel efficiency analysis showing occupancy, execution time distribution, and performance metrics across all GPU kernels	75
46	Bottleneck analysis showing the distribution of performance-limiting factors across GPU kernels	76
47	Distribution of theoretical occupancy across all kernel executions, showing most kernels achieve good resource utilization	77
48	Analysis of overlap between kernel execution and memory operations	77
49	Top kernel-memory overlap operations showing the most significant concurrent execution patterns	78
50	A summary of benchmarking results of the gPluto code. The table presents speedups of the individual code regions of two commits, both running on the GPU-accelerated nodes on Leonardo. In this particular case, the speedup is not achieved by porting code to the GPU, but by optimization of the GPU version of the code.	81
51	A summary of benchmarking results of the OPENGADGET3 code. The table presents a performance comparison of individual regions running on the CPU. The code was mostly optimised in the <i>Domain decomposition</i> region, where speedup is over 4x. Regions <i>Compute densities and Hydrodynamical forces</i> shows a slowdown in the new version; however, this is caused by the addition of new functionality and the improvement of numerical properties of the code. These regions also cause the overall lower speed of the entire code. In more detail, the performance results are presented in Section 3.1, please read them carefully.	81
52	A summary of benchmarking results of the RAMSES code. Here we present a speedup of the CPU version for individual regions and two overall speedups of the entire application. The <i>Entire code</i> column shows the overall speed achieved by optimization of the individual regions, and the column <i>Entire code impact of hybrid parallelization</i> presents additional speedup achieved by hybrid parallelization. These two optimizations are currently merged into a single branch and, therefore, are measured individually just for this deliverable. Merging these, which is the next step of the work, will lead to a combined speed of up to 1.55 for 64 nodes.	81
53	A summary of benchmarking results of the iPic3D code. Here, the comparison is done for two commits of the CPU version of the code and 3 individual regions.	81

54	A summary of benchmarking results of the BHAC code, which reports the performance speedup achieved by the development of the GPU-accelerated version. The BHAC was benchmarked on runs with a small number of iterations (100). In this configuration, around 25% of the total runtime for the GPU runs was spent on code initialization, whereas initialization accounted for only about 2% of the runtime in the CPU runs. As the number of iterations increases, the relative contribution of initialization to the total runtime decreases. Consequently, in longer runs, the overall speedup of the code will approach the speedup observed for the Riemann solver.	82
55	A summary of benchmarking results of the ChaNGa code. ChaNGa presents two benchmarking results. First we compare the improvement of the CPU version, which is up 1.65 for the entire code for 64 nodes. We can also see that major speedup is achieved by optimization of the <i>Gravity + Hydro</i> and <i>Star formation + Feedback</i> regions of the code. In addition, for ChaNGa we also report speedup achieved by the GPU acceleration, which is 2.9 - 3.4 when comparing one AMD EPYC 7763, 64-core, 2.45 GHz CPU and one Nvidia A100 GPU.	82

List of Tables

1	Benchmarking results of the gPLUTO code on Leonardo Booster partition. The commit 1 columns represent the older version of <code>gPLUTO-dev</code> (December 2022), the commit 2 columns represent the actual version of <code>gPLUTO-dev</code> (May 2025). Table [A] reports the time in second and the gain of the December 2022 vs May 2025 version as the number of nodes increases for runs of 100 steps. The bottoms tables [B], [C], [D] reports the time in ms/ds and the gain of the December 2022 vs May 2025 version for each kernels as the number of nodes increases.	10
2	Benchmarking results of OPENGADGET3 comparing a code version from before the start of the CoE with a recent version.	18
3	Profiling of timings, in seconds, in the FIND_HSML section when using either the CRITICAL construct or the LOCKS, both on LEONARDO nodes and on KAROLINA nodes. We note that for unknown reasons, the CRITICAL construct scales much better in KAROLINA than on LEONARDO even if the thread pinning has been carefully and explicitly set.	21
4	The speedup factor with GPU offloading w.r.t. pure CPU runs after the most recent re-sync of the GPU offloading scheme and the current code version. The timings refer to a group of galaxies at $z = 0$, performed on a single LEONARDO Booster node. All the runs used 2 MPI tasks with 8 OpenMP threads each.	21
5	Build time and efficiency scaling with OpenMP threads for the binary + octree build of box-leaves.	24
6	Base resolutions for the scaling measurements of 2D simulations with IMM.	30
7	Resolutions for the strong scaling measurements of 2D simulations with ECSIM on LUMI.	32
8	Resolutions for the strong scaling measurements of 3D simulations with ECSIM on LUMI.	33
9	Resolutions for the strong scaling measurements of 2D simulations with ECSIM on Karolina.	33
10	Number of maximum particles for one species of particles per MPI process for a given grid resolution on 8 (left) and 64 (right) nodes.	34
11	Benchmarking results of ECSIM for a 2D thermal plasma case.	35
12	Benchmarking results of ECSIM for a 3D thermal plasma case.	35
13	Performance gain for adding additional compiler flags for RAMSES for the sedov use-case. Value obtained by taking the average over 3 executions and comparing to the execution when compiled with only the default optimization flags in the Makefile (-O3). Benchmarks performed on MeluXina for the GNU compiler.	38
14	Comparison of the execution time of the MPI-only version of the Sedov use-case in RAMSES to the new MPI+OpenMP implementation. Benchmarks performed using the Sedov use-case on a 1024^3 grid for 20 steps, on the MeluXina cluster. The reported numbers are the average across 3 identical runs.	40
15	Comparison of the execution time of the MPI-only version of the Poisson multigrid solver in RAMSES to the new MPI+OpenMP implementation. Benchmarks performed using the cosmo use-case on a 1024^3 grid for 10 steps, on the MeluXina cluster. The reported numbers are the average across 3 identical runs.	41
16	Benchmarking results of the RAMSES code on MeluXina. Commit 1 is the commit of the public version of the code that entered the SPACE project. Commit 2 is the current state of the SPACE development branch. These benchmarks have been performed with MPI-only.	42

17	Benchmarking results for the BHAC code.	47
18	System configuration used for performance analysis on the Karolina supercomputer	48
19	Performance metrics summary across all ranks	49
20	Top time-consuming NVTX ranges	49
21	Key parent-child relationships in the NVTX hierarchy (Total Time = Self Time + Children Time)	49
22	Top time-consuming CUDA Runtime API calls	51
23	Top time-consuming kernels	51
24	Memory API usage summary	53
25	MPI call statistics	55
26	Communication hotspots and recommendations	55
27	Optimization recommendations and expected impact	56
28	Current code version (commit 2) compared to the old version of the code (commit 1). We run the simulation on the non-accelerated Karolina nodes across 4 to 64 compute nodes. We list the timings of the full runtime of the simulation, the median time per step of the gravity + hydrodynamics , star formation + feedback, domain decomposition (DD) + load balancing (LB) + tree building (TB) and radiative cooling. Gain is calculated following $1 - t_{\text{commit2}}/t_{\text{commit1}}$	69
29	Performance comparison between GPU and CPU versions on the Karolina supercomputer. The GPU version was executed on accelerated nodes (4-32 nodes, 32-256 GPUs), while the CPU version ran on non-accelerated nodes (16-256 nodes). We present: (1) total simulation runtime, and (2) gravity module timings (median, minimum, and maximum per-step values computed over the last 10 steps). Performance gain is calculated as $1 - t_{\text{commit2}}/t_{\text{commit1}}$	71
30	Performance metrics summary across all ranks. Notes: ¹ GPU Utilization: Percentage of total execution time spent in GPU kernels (Kernel Time / Total Duration). ² Peak GPU Memory usage during execution. ³ Memory Utilization: Peak GPU Memory divided by total available A100 GPU memory (40 GB). ⁴ CUDA Runtime API calls in millions. ⁵ Memory transfer operations (Host↔Device). ⁶ MPI communication calls in millions.	72
31	Communication statistics for Rank 0 (representative of typical rank behavior)	74
32	Memory transfer statistics for typical rank	75
33	Kernel-specific overlap statistics showing distribution of asynchronous execution patterns	76

1 Introduction

The deliverable reports on the ongoing profiling and benchmarking activities undertaken to develop and optimize the SPACE codes. The primary goal of this work is to perform benchmarking of the entire code and selected individual regions, as identified in Deliverable D2.1. This is conducted for at least two versions of the codes: Commit 1 (the version that entered the project at M1) and Commit 2 (a version around M30).

The approach involves using the use-cases defined in WP1 and running benchmarks up to 32 GPU nodes or up to 64 CPU nodes. Performance, measured as runtime, is assessed for both the entire application and individual computational kernels or modules that were identified in D2.1.

The performance results demonstrate the progress in the codes optimization, particularly highlighting optimized kernels/modules. All input data, configuration files, logs, and resulting performance measurements are stored in the Internal Data Archive, ensuring reproducibility and traceability.

Beyond benchmarking, the deliverable also covers profiling activities, including collaborations with the POP Centre of Excellence on audits for codes of iPic3D and OpenGADGET.

The profiling activities reported in this deliverable do not have a rigid structure for all codes, as it has been done in D2.1. Since every code goes through a different path of optimization process, the profiling work also differs based on the developers' specific needs. A brief overview of these activities is presented here:

- **POP Performance Assessment & Bottleneck Identification:** performance assessments using tools like **Extrae**, in collaboration with the **POP Centre of Excellence**, to identify performance bottlenecks and analyze codes (in this deliverable, primarily CPU versions).
- **Detailed Runtime and Kernel Analysis:** In-depth analysis of execution time at the kernel level or for specific modules. This includes analyzing kernel execution times, occupancy, throughput, register pressure, branch divergence, cache hit/miss rates, and stall reasons using tools like **NVIDIA Nsight Systems** and **Nsight Compute** (for GPU-enabled codes), **AMD Rocprofv2** and **OmniPerf** (for gPLUTO on AMD GPUs). **NVTX ranges** were used in BHAC profiling for high-level function analysis.
- **Memory Profiling:** Analysis of memory access patterns, management, and consumption. This involved investigating the impact of **High Bandwidth Memory (HBM)**, analyzing GPU memory usage over time, and identifying memory leaks or excessive allocations using tools like **Nsight Compute memory chart** and **Valgrind's Massif** (for FIL memory consumption scaling).
- **Communication and Synchronization Analysis:** Focusing on MPI communication patterns, synchronization overheads, and data transfers. This includes analyzing MPI call statistics, communication hotspots, collective operation performance (like MPI Allreduce variability in FIL), and the lack of computation-transfer overlap.
- **Load Balance Analysis:** Identifying and analyzing load imbalance among processes or threads, as seen in the profiling of FIL and highlighted in POP audits.
- **Architectural and Comparative Profiling:** Comparing performance across different architectural approaches (e.g., CPU MPI vs CPU MPI+OpenMP vs GPU for iPic3D) and evaluating specific optimization strategies or compiler effects (e.g., RAMSES using **MAQAO**). Profiling of OpenMP offloading for gPLUTO on different GPU architectures (NVIDIA vs AMD) is also included.
- **Framework-Specific Profiling:** Utilizing profiling tools specific to the parallel framework used by a code, such as the **Charm++ Projections tool** for ChaNGa.

The profiling efforts have covered a wide range of performance aspects, moving from initial high-level assessments to detailed analysis of compute, memory, and communication characteristics, tailored to the specific architectures (CPU, GPU) and parallel paradigms (MPI, OpenMP, Charm++, OpenACC) used by the SPACE codes.

2 PLUTO

PLUTO [1] is designed to integrate a general system of conservation laws that we write as

$$\frac{\partial U}{\partial t} = -\nabla \cdot T(U) + S(U). \quad (1)$$

Here U denotes a state vector of conservative quantities, $T(U)$ is a rank 2 tensor (the rows of which are the fluxes of each component) and $S(U)$ defines the source terms. Additional source terms may implicitly arise when taking the divergence of $T(U)$ in a curvilinear system of coordinates. An arbitrary number of advection equations may be added to the original conservation law.

Although the components of U are the primary variables being updated, fluxes are computed more conveniently using a different set of physical quantities, which we take as the primitive vector V . Numerical integration of the conservation law above is achieved through shock-capturing schemes using the Finite-Volume (FV) formalism where volume averages evolve in time. Generally speaking, these methods comprise three steps: an interpolation routine followed by the solution of Riemann problems at zone edges and a final evolution stage. In PLUTO, this sequence of steps provides the necessary infrastructure of the code; see the schematic diagram in Fig.1: first, volume averages U are more conveniently mapped into primitive quantities V . Left and right states $V_{+,L}$ and $V_{-,R}$ are reconstructed inside each zone and a Riemann problem is then solved between them to obtain the numerical flux function F_+ at interfaces. The solution is finally advanced in time.

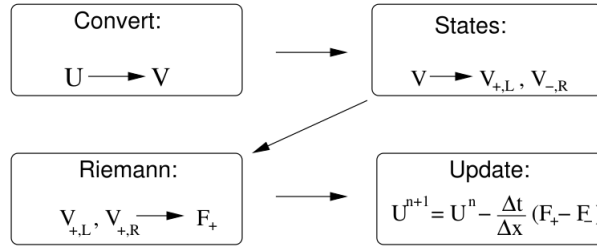


Figure 1: Diagram of the Reconstruct-solve-average (RSA) strategy

2.1 Use-case description

To test the development of the application we considered two status (December 2022 and May 2025) of the GPU-enabled version, gPLUTO. The test used for evaluating the performance between the initial version of the code and the present one has been already described in SPACE D2.1. It consists of a 3D relativistic magneto hydrodynamics (MHD) blast wave in Cartesian coordinates.

The versions of gPLUTO can be founded on GitLab at <https://gitlab.com/PLUTO-code/gPLUTO-dev>: the private developer gPLUTO repository (gPLUTO-dev).

The actual version is: `f5af44e8a0ec4cd4e6d21477e9897d6bebea0472` dated 23 - May - 2025 (1) and the older version is: `1896c391cd81a2a9faf088682ba9c535a2a37c9` dated 07 - December - 2022.

2.2 Benchmarking results

In Tab.1 we report the runtime and gain [%] of the runs of the two versions of the gPLUTO code, performed on the Loenardo Booster Partition up to 64 nodes, with a fixed resolution per process of 320^3 points (4 GPUs/processes per node). The top of Tab.1 reports the entire run time, considering 100 steps, as well as the average runtime of the most time consuming regions, identified and discussed in D2.1.

The performance gains shown in the top table [A] demonstrate that the current versions of gPLUTO achieve an increase of about 15% compared to the old version when using a small number of nodes/communications, and up to about 30% with a larger number of nodes/communications. This dependency on the number of nodes can be explained by looking at the boundary kernel table, which is solely responsible for handling communication both within and between nodes. While other kernels have been optimized for single process execution, the boundary kernel required additional optimizations to make inter-process communication efficient and asynchronous.

Table [A]	Entire code runtime [s] (100 steps)		
# of nodes	commit 1	commit 2	gain [%]
1	84.25	71.27	15.4
2	85.13	71.75	15.7
4	89.00	72.09	19.0
8	93.13	72.31	22.3
16	98.77	73.30	25.8
32	102.46	73.54	28.2
64	110.41	73.33	33.6

Table [B]	Boundary runtime [ms]			Riemann Solver runtime [ms]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
1	2.94	1.58	46.1	44.49	31.53	29.1
2	7.29	2.85	60.9	44.76	31.65	29.3
4	23.49	4.67	80.1	44.94	31.65	29.6
8	22.80	5.85	74.4	45.50	31.74	30.2
16	22.19	7.12	67.9	46.02	31.84	30.8
32	20.01	7.32	63.5	46.20	31.91	30.9
64	21.10	6.58	68.8	46.21	32.12	30.5

Table [C]	Righ Hand Side runtime [ms]			Update Stage runtime [ds]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
1	10.84	4.17	61.5	3.66	2.79	23.6
2	10.61	4.18	60.6	3.63	2.81	22.6
4	10.60	4.18	60.6	3.66	2.81	23.2
8	10.61	4.17	60.7	3.66	2.82	23.1
16	10.60	4.17	60.6	3.66	2.82	23.0
32	10.62	4.18	60.6	3.68	2.81	23.8
64	10.63	4.19	60.6	3.68	2.83	23.3

Table [D]	CT Update runtime [ms]			Advance Step runtime [ds]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
1	2.83	1.90	32.8	8.13	6.82	16.2
2	2.82	1.89	32.7	8.20	6.85	16.5
4	3.00	1.91	36.6	8.60	6.88	20.0
8	2.85	1.91	33.1	8.88	6.92	22.0
16	2.88	1.91	33.7	8.90	6.96	21.8
32	2.77	1.91	31.3	8.68	6.96	19.8
64	2.81	1.91	31.9	8.69	6.97	19.8

Table 1: Benchmarking results of the gPLUTO code on Leonardo Booster partition. The commit 1 columns represent the older version of **gPLUTO-dev** (December 2022), the commit 2 columns represent the actual version of **gPLUTO-dev** (May 2025). Table [A] reports the time in second and the gain of the December 2022 vs May 2025 version as the number of nodes increases for runs of 100 steps. The bottoms tables [B], [C], [D] reports the time in ms/ds and the gain of the December 2022 vs May 2025 version for each kernels as the number of nodes increases.

2.3 Profiling activities

With the frequent addition of new test problems and algorithms to the code, many profiling reports have been produced using **NVIDIA Nsight Systems** to assess the efficiency on GPUs, primarily on our local workstation and on Leonardo Supercomputer. Additionally, Eviden produced a report exploring the impact of High Bandwidth Memory (HBM) on the performance of gPLUTO focusing on the ResRMHD Blast 3 test case using CPUs only (see D2.5). They implemented memory binding strategies to ensure that processes accessed HBM rather than DDR5 memory. Runs using HBM consistently outperformed those using only DDR5. Further profiling of memory access patterns revealed that gPLUTO, at least on this test case, is predominantly compute-bound. Therefore, memory improvements have a limited but still noticeable impact.

2.4 Status of OpenMP offloading activity

One of the goals of gPLUTO is to ensure portability across different GPUs, such as those from AMD and Intel, by maintaining an OpenACC-based implementation. Significant work has already been done in this area, demonstrating excellent speedups. However, achieving this portability is not straightforward, as compiler support for OpenACC-enabled C++ code on AMD and Intel GPUs is currently lacking. To address this, we have adopted the use of OpenMP target directives to offload code to GPUs. This method is supported by several compilers, including the Cray compiler, AMD-CLANG compiler, GNU-Compiler, Intel compiler, and NVIDIA's NVHPC compiler, making it a viable and portable solution. Like OpenACC, OpenMP offers a high-level abstraction that simplifies many aspects of GPU programming—such as device management, thread hierarchy, and memory transfers, while still allowing developers explicit control over data placement and parallel execution. In this section, we present our initial code development work, profiling reports, and benchmark results for the relevant modules. This approach provides a portable and maintainable path for extending gPLUTO GPU support to heterogeneous systems, without requiring significant re-engineering of the existing code base.

In this context, OpenMP offloading is employed to accelerate gPLUTO's core computational kernels on GPU hardware. As a first step, we extracted a representative lite-version application from gPLUTO, focusing on a simple Hydro Riemann 2D test. This allowed us to evaluate the offloading strategy in isolation, establish performance baselines, and refine data mapping schemes. To manage memory across host and device, we make use of both explicit data transfers via `map` clauses and Unified Memory (UM), depending on the target platform and compiler behavior. UM offers simplified memory semantics by enabling shared virtual memory spaces, automatically migrating data between CPU and GPU. However, while UM can coexist with OpenMP directives, its use can override or interfere with explicit map semantics, especially when used with compilers that assume ownership of memory management under UM. Therefore, explicit mapping is preferred in performance-critical paths to retain fine-grained control over memory traffic and achieve deterministic behavior across different architectures. The offloading model uses `target` regions in combination with `teams` and `distribute parallel for` directives to express hierarchical parallelism and effectively map workloads to the GPU architecture.

Involved kernels are:

- `OutflowBoundary`
- `UpdateStage`
- `Reconstruct`
- `HLL_Solver`
- `RightHandSide`
- `AdvanceStep`
- `ConsToPrim3D`
- other small computational kernels.

Initial activity has been performed on Leonardo HPC cluster, using the OpenMP implementation provided by the NVHPC/24.3 compiler suite, together with managed memory. This phase served to validate the correctness of the approach and assess whether OpenMP offloading was a suitable and promising strategy for further development.

Offloading activities were centered around profiling and optimizing individual kernels, analyzed iteratively to understand their behavior and improve performance. A comprehensive performance investigation was, again, conducted using NVIDIA Nsight Systems and Nsight Compute, which provided low-level insights into GPU execution. These tools enabled fine-grained analysis of each kernel execution time, occupancy, throughput, memory access patterns, register pressure, branch divergence, cache hit/miss rates, and stall reasons. This profiling process provided actionable feedback that guided kernel restructuring, tuning of loop scheduling, and adjustments to data placement strategies. A snapshot of the Nsight Systems profile for a gPLUTO loop is shown in Figure 2. Figures 3 and 4 display the roofline and the memory chart plots obtained for the **Reconstruct** kernel, respectively.

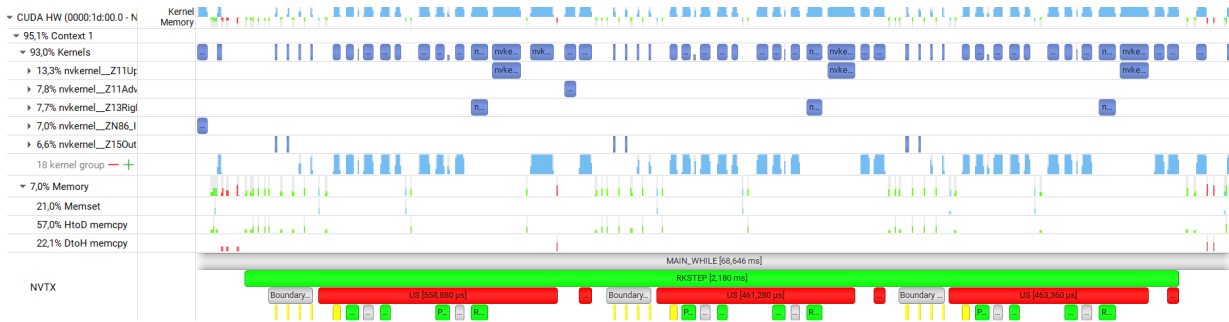


Figure 2: Nsight Systems timeline showing kernel execution and memory events for a representative gPLUTO loop.

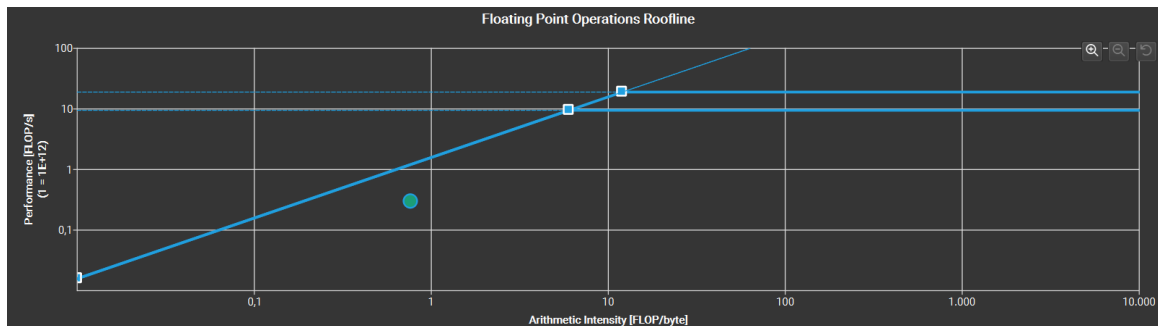


Figure 3: Nsight Compute roofline plot for the **Reconstruct** kernel. The plot shows arithmetic intensity and performance relative to hardware throughput limits.

The optimal GPU OpenMP offloading implementation for the gPLUTO lite-version was carried out during the AMD Hackathon in Oslo (May 12–16), which provided direct access to the LUMI supercomputer and on-site collaboration with AMD experts. During this effort, the application was compiled using the AMD-CLANG 6.0.3 compiler, with support from the ROCm 6.2.4 software stack. A UM environment was adopted for memory management, consistent with the initial approach on NVIDIA systems. Initial performance on LUMI was suboptimal, and several contributing factors were identified:

- Compiler-specific directive semantics: some OpenMP constructs, such as `#pragma omp distribute teams loop` vs. `#pragma omp distribute teams parallel for`, exhibited differences in performance and behavior depending on the compiler. These discrepancies necessitated a refactoring of loop structures within the computational kernels to expose parallelism more effectively and align with AMD’s execution model.
- Missing SIMD-level parallelism: unlike the NVIDIA implementation, which benefits from SIMD vectorization under the NVHPC compiler, the AMD-CLANG compiler currently lacks full support for OpenMP SIMD constructs on GPUs. This limitation reduced computational throughput and motivated architectural adaptations.
- Memory access and cache optimization: detailed profiling revealed inefficiencies in memory coalescence and cache line utilization. These were addressed by restructuring loops — such as reordering loop nests and applying `collapse` clauses — to improve spatial locality, memory alignment, and, hence, throughput.

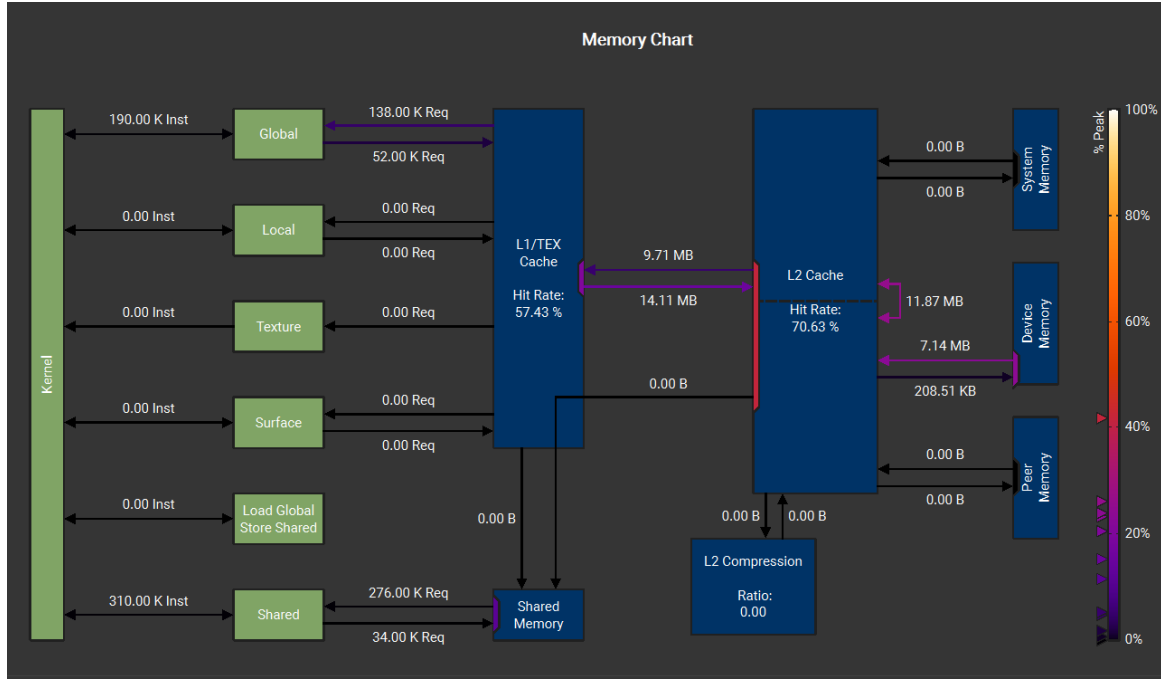


Figure 4: Nsight Compute memory chart for the **Reconstruct** kernel. Memory load/store transactions are broken down by cache level and memory space.

Through these optimizations, a $15\times$ performance improvement was achieved on LUMI compared to the initial OpenMP offloading implementation. The final performance became comparable to the OpenMP version on Leonardo, though both still trail the OpenACC version by approximately $3\times$ on average, primarily due to the maturity and hardware-specific tuning of the OpenACC backend. Profiling and performance analysis on LUMI were conducted using AMD’s tool-chain, including:

- Rocprofv2 – for GPU activity tracing and kernel-level performance breakdowns;
- OmniPerf – to generate roofline models, visualize bottlenecks, and assess memory vs. compute-bound behavior;
- OmniTrace – for detailed execution timelines and synchronization overhead analysis.

Figure 5 presents a Rocprofv2 snapshot highlighting kernel execution metrics during a representative gPLUTO run. Figure 6 shows the roofline model generated by OmniPerf for the ten most computationally intensive kernels, illustrating the impact of optimization on arithmetic intensity and memory bandwidth utilization.

The next phase of development will focus on extending GPU offloading to the entire gPLUTO application, covering scientifically relevant, production-scale simulation cases. Parallel efforts are also underway to reduce reliance on UM, which—while convenient—limits control over data movement and performance tuning. To this end, we have begun implementing OpenMP mappers to explicitly manage the transfer of complex data structures (e.g., class and struct members). However, a CLANG compiler bug currently prevents the utilization of user-defined mappers within gPLUTO implementation, and we are actively tracking its resolution. Further performance improvements are planned through the adoption of multi-GPU execution models also for the OpenMP implementation of the code and asynchronous boundary data exchanges, both of which are critical for scaling gPLUTO on exascale-class systems such as LUMI.

2.5 Conclusions

Our results demonstrate that considerable progress has been achieved on gPLUTO while moving from an earlier commit to the present day version of the code (on the average, 25% speedup). These achievements have been obtained through careful benchmarking and profiling activities followed by optimization actions focusing on specific kernels over 2 years of the SPACE lifespan.

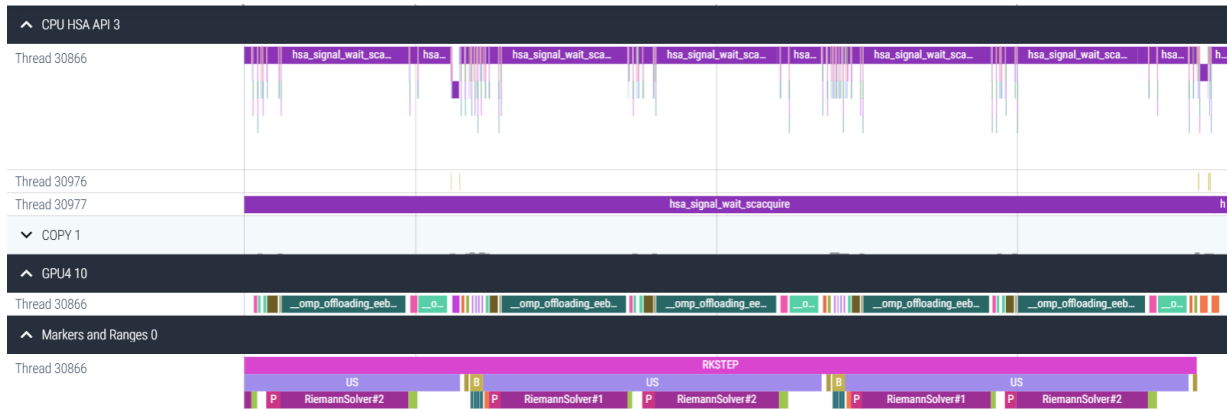


Figure 5: Rocprofv2 kernel activity trace for a representative time step in gPLUTO. The timeline shows GPU kernel execution and memory transfer events, identifying hotspots and synchronization delays.

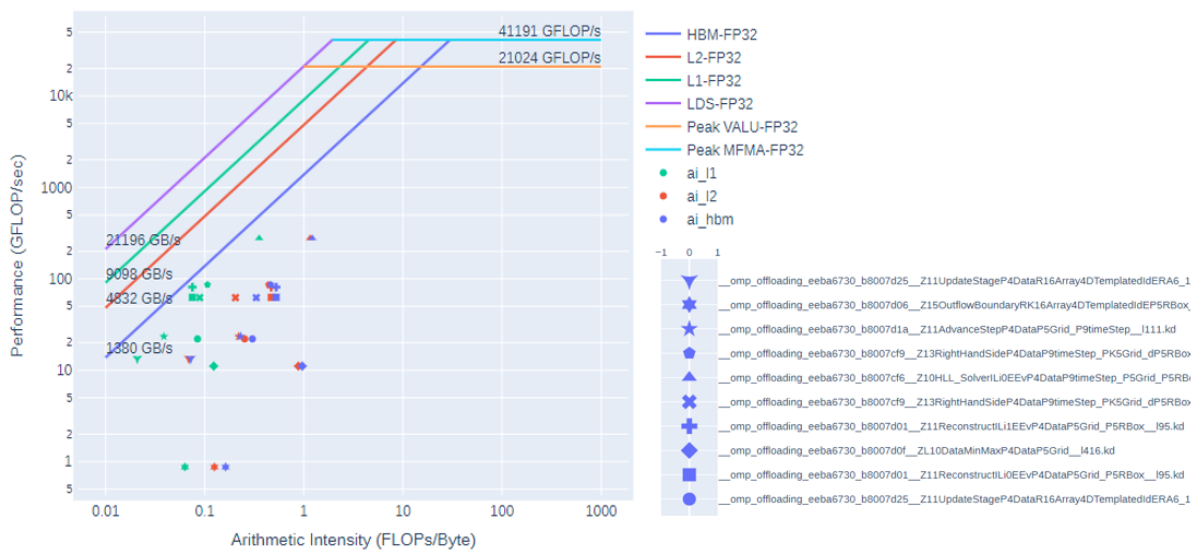


Figure 6: OmniPerf-generated Roofline plot for the ten most computationally intensive kernels in gPLUTO. The diagram illustrates achieved performance in terms of arithmetic intensity and identifies compute-bound and memory-bound regions relative to hardware ceilings, as well as the legend for the ten most time consuming kernels considered.

At the same time, we have added OpenMP `#pragma` directives to the code in order to provide acceleration capabilities on GPUs from other vendors than NVIDIA. More specifically, we benchmarked and profiled a lite-version of the code on AMD GPUs. These first profiling activities show, after an optimization phase, a promising increase in performance (about 15 x) for the gPLUTO lite-version app on LUMI-G (AMD) partition. Using similar optimizations we expect to be able to achieve a performance increase on the full app in the future as well.

3 OpenGadget3

OPENGADGET3 is a collisionless N-Body/Lagrangian cosmological code using the smoothed particle hydrodynamics (SPH) computational method to describe the motion of fluids in addition to the gravitational forces, which are calculated using a tree structure. The code allows simulations in a full cosmological context, i.e. accounting for an expanding background and the presence of matter, both “dark” and baryonic (ordinary matter), and dark energy. In addition to the gravitational problem, it also simulates the evolution of the physical properties of the baryonic matter subject to hydrodynamics and other physical effects such as radiative cooling, star formation, energy feedback, radiative transfer, magnetic fields and others that we collectively refer to as “extra physics”. The code is used to study the evolution of large structures and how they are shaped by several astrophysical properties, such as the clustering of matter and the impact of baryonic processes in the Universe (such as the phase-space distribution, the history of stellar feedback and chemical enrichment, the growth of black holes and their relationship with the environment and hosting objects, and many others).

3.1 Benchmarking results

In our code comparison, we employed a “cosmological box” measuring 128 Mpc/h across and populated it with 2×576^3 particles. The simulation included “full physics”, encompassing gravity calculation, hydrodynamics, and enabled non-radiative and radiative physics. Benchmarking was conducted across 6 to 64 nodes on LUMI, using 16 MPI tasks and 8 OpenMP threads per node. The code was run for 24 steps in addition to the initialisation.

Figure 7 shows the scaling performance of the new code version compared to the old version. Both implementations exhibit similar scaling behaviour up to approximately 32 nodes. Beyond this point, the performance of the old version begins to degrade, while the new version continues to show a flatter but consistent increase in speedup. This demonstrates the improved scalability of the new code version.

The actual runtime results are presented in Table 2. To understand the overall slowdown of approximately 10%, it is essential to analyse the performance of each individual region.

3.1.1 Extra-physics

The region labeled “Extra-physics” encompasses all physical modules beyond gravity and hydrodynamics, including processes such as stellar formation and evolution, as well as black hole and AGN activity. Although several updates have been made to these modules—such as the introduction of spin for black holes—the total runtime remains comparable between the old and new versions of the code, with some stochastic fluctuations across the runs. Since this component contributes only a small fraction to the overall runtime, it is not a primary focus for optimisation or GPU offloading.

3.1.2 Gravitational accelerations

To address the inaccuracies identified in parts of the calculation, we enhanced numerical precision and implemented locks in critical loops to ensure thread safety. These adjustments have improved the reliability and accuracy of our results. However, they have also introduced increased computational demands, particularly in unresolved type conversions. In this test scenario, we have enforced double precision, which avoids costly type conversions and results in a speedup of $\sim 20 - 30\%$ compared to the old code version.

In contrast, in the default setup, a lightweight mixed-precision configuration, where memory usage for various particle structures is reduced by $\sim 30\%$, the updated code exhibits a performance slowdown of $\sim 20\%$ compared to the previous version.

3.1.3 Compute densities

The computation of densities now takes approximately twice as long in the updated code version. This increase in computation time arises from the inclusion of higher-order gradient calculations and the resolution of specific inconsistencies within the code, which require the execution of certain loops twice.

3.1.4 Hydrodynamical forces

As stated before, this test case enforces double precision. While gravitational force calculations benefit from a speedup in this setup (and a slowdown in the default case), the opposite holds for hydrodynamical force

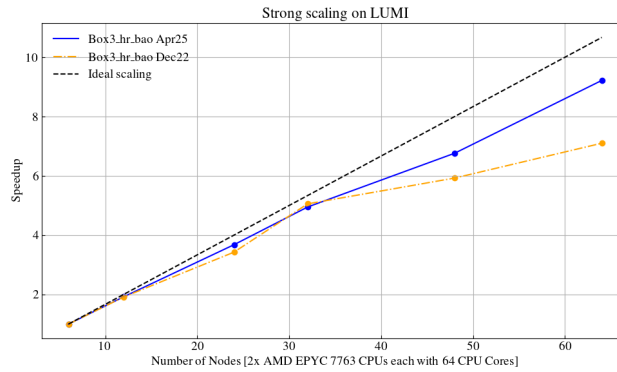


Figure 7: Strong scaling on LUMI for a recent code version (blue) and one before the start of the CoE (orange).

computations: we observe a general slowdown of up to 50% in double precision. This degradation comes from increased memory demands. The use of higher-order gradients over simple kernels requires larger matrices and additional parameters, expanding the particle struct and increasing memory requirements especially in double precision.

In contrast, the default precision case shows a performance gain of approximately 10% when comparing the new code to the old. This improvement confirms that, in the absence of memory limitations, the computational efficiency of the new implementation has improved.

See Sec. 3.2.2 for a discussion of a promising perspective for the improvement of the openmp scalability.

3.1.5 Domain decomposition

In contrast to the initial profiling performed with Extrae, we chose to merge the two separate regions associated with domain decomposition into a single region. Through targeted optimisations and the introduction of OpenMP parallelisation across multiple loops within this module, we achieved a substantial performance improvement. These enhancements resulted in a $\sim 250\%$ speedup compared to the previous implementation. Furthermore, the optimised domain decomposition now exhibits nearly ideal scaling across a $6\times$ increase in computational resources.

3.2 Profiling activities

3.2.1 CPU performance assessment with Extrae in collaboration with POP3 CoE

In the case of `OPENGADGET3`, a “cosmological box” with a box size of 30Mpc/h and 256^3 particles was used. This analysis was performed from 1 to 16 nodes, where both gravity and hydrodynamics were enabled. The last 15 time steps were selected as the Region of interest (RoI). Figure 8 shows that the speedup of the RoI gradually diverges from the optimum, with the efficiency below 80% on 8 nodes (896 MPI processes and 7 OpenMP threads per process). The hierarchical multiplicative model of efficiency metrics on Figure 9 identifies Serialization (SerE) and OpenMP Communication (OCE) efficiencies as the main limiting factor of the RoI scaling. A detailed explanation of the model and the metrics can be found at [2, 3].

	Entire code runtime [s]		
# of nodes	commit 1	commit 2	gain [%]
6	1069.04	1170.48	−9.4%
12	552.2	610.27	−10.5%
24	304.41	318.66	−4.6%
32	207.53	236.86	−14.1%
48	173.16	173.51	−0.2%
64	146.05	127.21	14.8%

	Domain decomposition runtime [s]			Gravitational accelerations runtime [s]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
6	205.78	46.67	340.9%	175.19	130.35	34.4%
12	103.55	28.28	266.2%	90.83	71.46	27.1%
24	64.72	16.84	284.3%	46.04	37.54	22.6%
32	31.41	11.31	177.7%	36.34	28.36	28.1%
48	43.53	12.39	251.3%	26.05	19.82	31.4%
64	27.92	6.75	313.6%	30.29	15.26	98.5%

	Compute densities runtime [s]			Hydrodynamical forces runtime [s]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
6	179.82	428.64	−138.4%	219.91	330.44	−50.2%
12	93.67	217.56	−132.2%	113.15	167.88	−48.3%
24	51.51	113.74	−120.8%	62.23	86.62	−39.2%
32	37.08	85.20	−129.7%	43.94	64.21	−46.1%
48	27.61	58.35	−111.3%	32.52	47.14	−44.9%
64	22.28	44.54	−99.9%	26.36	33.42	−26.7%

	Extra-physics runtime [s]		
6	211.85	207.01	2.3%
12	110.39	109.37	0.9%
24	57.76	54.59	5.5%
32	42.32	39.99	5.5%
48	30.40	28.40	9.3%
64	26.02	20.55	21.0%

Table 2: Benchmarking results of OPENGADGET3 comparing a code version from before the start of the CoE with a recent version.

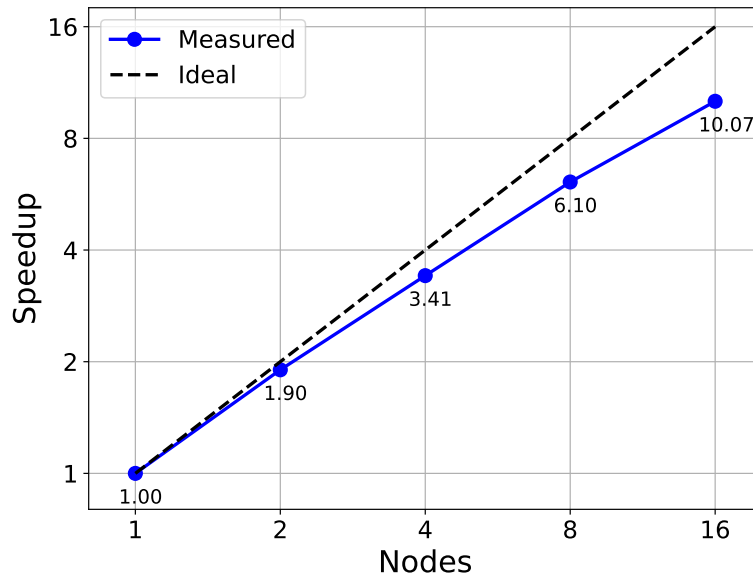


Figure 8: OPENGADGET3 on Leonardo DCGP for up to 16 nodes - strong scaling.

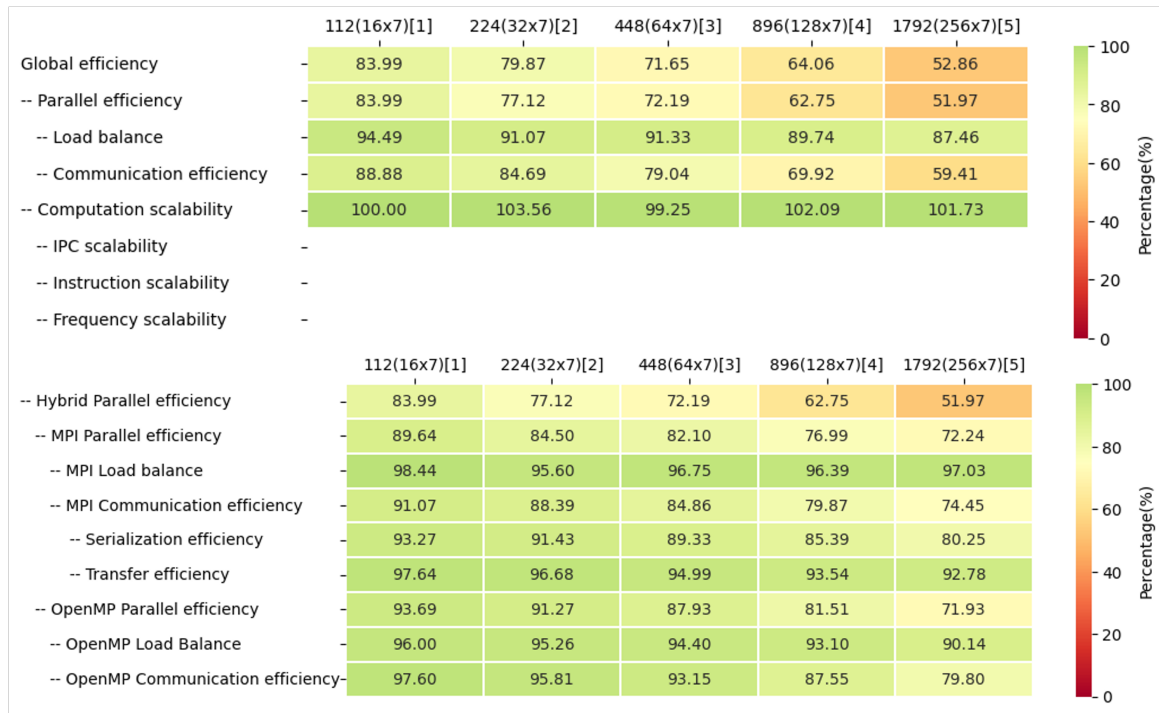


Figure 9: OPENGADGET3 on Leonardo DCGP for up to 16 nodes - efficiency metrics [%]. Note the Computation scalability submetrics were unavailable due to incompatible tool suite components.

Each RoI time step consists of the following set of high-level routines that were instrumented and further analysed: Domain decomposition intensity decision (DD1), Domain decomposition intensity execute (DD2), Gravitational accelerations (GRAV), Densities (DENS), Hydro-accelerations (HYDRO), Non-standard physics (PHYS).

The SerE describes any loss of efficiency due to dependencies between processes causing alternating processes to wait. The lowest SerE (80%) was identified in DD2 routine, but due to its relatively short duration, it is not the main cause of degradation in the test case. The GRAV routine is the longest one, but with a high SerE 92%

it is also non-priority in the SerE optimization. Finally, the second longest DENS routine with SerE 86% is the best optimization candidate. Simulation of an ideal network with infinite bandwidth and zero latency, where each message is transferred immediately, revealed that each process spends in average 12% of the runtime in the MPI_ALLTOALL function waiting for the dependent processes.

Regarding the OCE, which captures the synchronization and scheduling overhead induced by the OpenMP constructs, the lowest values 36% and 24% were observed in DD1 and DD2 respectively, though, being the shortest routines, they are probably not worth optimizing with the given test case. On the other hand, rather small room for improvement remains in the longest GRAV routine showing the best OCE 93%. The best candidate for optimization is again the DENS routine with OCE 80%, followed by HYDRO with 88% and much shorter PHYS with 67%. In the DENS, each process spends a significant part of the total execution time in OpenMP runtime due to very small granularity of the parallel functions, mostly in `find_hsml` and `compute_unified_gradients` functions with 24% and 10% in average, respectively.

In this respect, please see recent results about a possible mitigation as discussed in Sec. 3.2.2.

3.2.2 CPU profiling, follow-up

Following the results of the previous analysis, we delved deeper into aspects related to OpenMP scalability, focusing in particular on the FIND_HSML region, which is among those showing the worst scalability. Specifically, we observed that most of the time is spent in a loop that has the following pattern (in a simplified code listing):

```
#pragma omp for
for ( int i = 0; i < NactiveParticles; i++ )
{
    /* perform operations on active particles */
    ...

    #pragma omp critical ( drift_particles )
    if (target_particle_j is not drifted )
    {
        /* the particle j is not yet drifted          */
        /* which means that it is not active and      */
        /* that the drift operator has not yet been    */
        /* applied to its position                    */
        drift( target_particle_j );
    }
    ...

    #pragma omp critical ( drift_nodes )
    if (target_node_k is not drifted )
    {
        /* see previous comments about drifting particles */
        drift( target_node_k );
    }
    ...
}
```

where, while processing active particles, it is checked whether non-active particles with which every active particle interacts have updated positions. If not, those particles are *drifted*, which means that their properties are updated to the current time using the previous estimates of their derivatives. The same holds for nodes.

Note that the drift operation involves several floating-point operations and is computationally expensive. The fact that they are inserted into CRITICAL regions means that even when more threads are processing active particles there were restrained from entering in that code region until any other thread is in there, even if they would have targeted different particles.

	1TH	2TH	4TH	8TH	16TH
LEONARDO					
<i>critical</i>	14.9704	10.2323	7.93972	7.06009	7.66936
<i>locks</i>	14.9072	9.39104	5.15594	2.14641	1.2148
KAROLINA					
<i>critical</i>	14.975	10.2355	7.05451	5.70552	1.95609
<i>locks</i>	14.9636	9.44223	5.20027	2.16546	1.23701

Table 3: Profiling of timings, in seconds, in the FIND_HSML section when using either the CRITICAL construct or the LOCKS, both on LEONARDO nodes and on KAROLINA nodes. We note that for unknown reasons, the CRITICAL construct scales much better in KAROLINA than on LEONARDO even if the thread pinning has been carefully and explicitly set.

That is, then, an inherent serialisation which tends to worsen when the number of threads grows.

We then experimented with a different parallelisation, based on locks, which by design protects data instead of code region.

Very recent profiling results, obtained with the help of the POP CoE and IT4I, show that this mechanism should be, in principle, very effective, as reported in Table 3.

PAPI counters and further analysis on LEONARDO confirms that the time spent in the lock-based version is much smaller and that the usage of CRITICAL regions actually results in a serialization of the corresponding drifting of the particles even if the targets are not the same for different threads.

At the moment of writing, these are very new and last-minute results, and hence, more details and exhaustive analysis will be given in future documents. However, our preliminary conclusion is that adopting the same lock-based mechanism should bring some positive impact on the overall timing.

3.2.3 GPU offloading

As mentioned above, we have profoundly re-worked the code in several aspects; for instance, the density calculation is now split into two distinct phases to enhance the accuracy of sph-estimate of physical quantities (that depends on the density ρ_i of the i th neighbour particles).

Understandably, that has led to a divergence between the previous GPU offloading scheme and the current code. A significant effort has been spent to synchronise the GPU offloading. At the moment of writing, we are testing it for the gravity and hydro sections, and completing the work for the extra-physics modules.

Preliminary tests for a galaxy group at redshift $z = 0$ (and hence in a configuration that is not "gpu-friendly" because the particles are highly clustered) on a single LEONARDO node show the speed-up reported in Table 4.

Code Region	CPU only	1 GPU per MPI task	gain [speedup]
ALL	5943	3516	1.7×
GRAVITY TREE	4259	1726	2.5×
FIND_HSML	304	175	1.7×
GRADIENTS	220	96	2.3×
HYDRO ACCEL.	138	137	—
THERMAL CONDUCTION	121	74	1.6×

Table 4: The speedup factor with GPU offloading w.r.t. pure CPU runs after the most recent re-sync of the GPU offloading scheme and the current code version. The timings refer to a group of galaxies at $z = 0$, performed on a single LEONARDO Booster node. All the runs used 2 MPI tasks with 8 OpenMP threads each.

3.3 Code Refactoring

Part of the code refactoring is the re-implementation of the Barnes & Hut (B&H) algorithm, with the aim of developing a GPU-friendly algorithm with an optimized implementation.

As detailed in previous deliverables, the primary sources of inefficiency in the “classic” B&H algorithm are essentially two: (i) sparse memory access, which descends from hierarchical time step and not all particles being updated every time, hampers coalesced memory access on GPU, and (ii) thread divergence due to the unique point of view of every particle.

Considering the Peano-Hilbert (PH) Space-Filling Curve (SFC) as our choice for mapping 3D coordinates in 1D, our strategy consists of the following ingredients:

- a. making Oct-Tree leaves contain a bucket of particles instead of single ones (which will be referred to as “BoxLeaves” in the following);
- b. letting the Tree-Walk be unique to all the particles in a leaf by considering their centre of mass as the point of view
- c. making the Oct-Tree sub volumes coinciding with PH volumes, i.e. convex regions of space that are filled by a unique PH segment;
- d. keeping the particles partitioned in memory by their PH key so that particles that belong to a BoxLeaf are also close in memory.

We expect this strategy to have several beneficial effects.

1. building the tree by boxes (step *a*), lowers the tree depth and hence fastens the tree building and the tree walk;
2. considering a unique point of view (step *b*), minimizes the thread divergence on GPU and, with *d*, enhances the efficiency of memory transfer;
3. The two final factors, *c* and *d*, render it easier to develop an effective tree-building and tree-walking based entirely on PH key, which are integers, instead of on 3D floating-point coordinates.

In Sec. 3.3.1 we report the status of the new B&H implementation of the Tree-Walk, while in Sec. 3.3.2 and 3.3.3 we report the progress on a new Oct-Tree construction, which is insertion-free, fully based on the Particles’ PH keys. By design, it produces BoxLeaves that are PH volumes, and hence can be directly interfaced with the new Barnes&Hut strategy that we described in this section.

3.3.1 New Barnes & Hut Module

We have opted to proceed in parallel with the development of the alternative B&H tree walk and the new tree construction. By designing them in a modular way, it will be straightforward to back-integrate them into the code at a later stage.

In this section, we summarise the status of implementing the new B&H tree walk, based on the top-down insertion-based tree building from OpenGadget. At the time of writing:

- we have modified the tree building so that
 - i* the leaves are always PH subvolumes;
 - ii* they contain a bunch of particles, which in turn are partitioned in memory by their PH key;
 - iii* the tree building is now a well-separated module that interacts with the entire code only via API that offers access to domain decomposition, tree building, tree walking for the *knn* problems and “B&H-like” tree walking;
- The new tree walk is also offloaded to the GPU via OpenACC. Currently, on each MPI task, the calculation of gravity due to local particles is not yet overlapped with the all-to-all communication needed to calculate the Gravity due to particles residing on other MPI tasks. The asynchronous implementation is under development.
- Gravity estimate can be executed for all the particles in a Box-Leave instead of just for the particles active in a given timestep; we are still experimenting with it in terms of efficiency and tuning;

- Newly created particles (for instance, due to star formation) are handled when the tree is updated and not reconstructed from scratch. In this case, an additional partition is created in the particles array, which is kept sorted by PH key at each time step. Each Task is then informed of the “interval” of particles assigned to it in that partition, and each time the particles of a node must be scanned, we know exactly where to go in both the “old” particles and the “new” particles partitions.
- A complete test infrastructure has been developed in order to check that the new tree is appropriately built and that the particle-per-particle acceleration at every time step does not deviate from the original implementation (when the leaves contain one single particle, the results are binary equal).

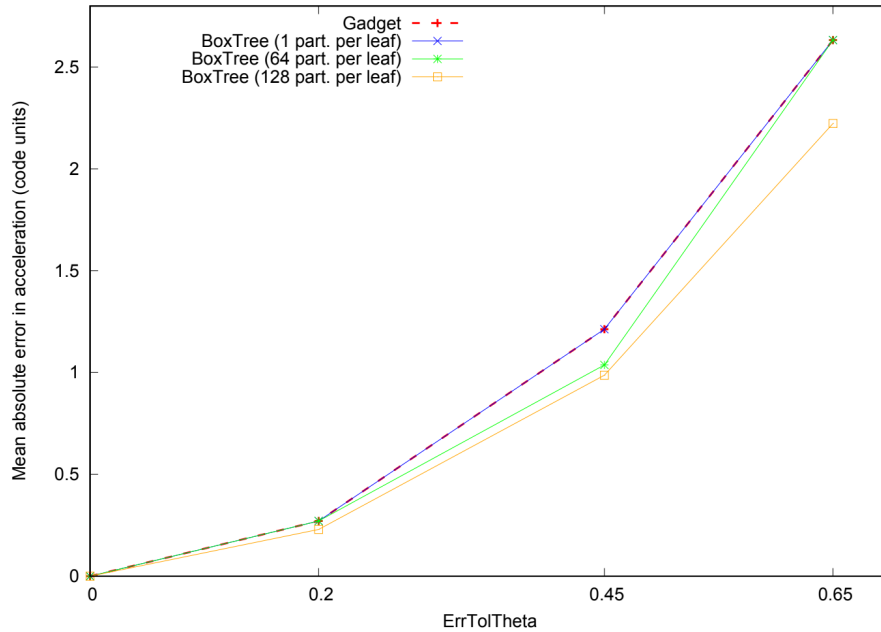


Figure 10: The absolute error (y -axis) of accelerations due to Gravity, in code units. The plot shows the absolute difference between the acceleration computed by the B&H algorithm and that computed via direct summation as a function of the θ opening angle (x -axis; $\theta = 0$ coincides with the direct summation since all tree nodes are “open” during the tree walk down to the leaves). The red dashed line refers to the original implementation in Gadget while the blue line to the results obtained by the new implementation described in this section when the leaves contain only one particle instead of a bunch of particles. Green and Yellow lines ref to different number of particles being included, in average, in a single Box-Leaf.

Figure 10 shows preliminary results from a correctness check, where we test the error in the gravitational acceleration w.r.t. the direct summation estimate for the original code and for our new implementation.

3.3.2 New Octree

As mentioned in previous deliverables, the construction of the Oct-Tree is one of the few inherently serial components within OPENGADGET3. Accelerating this process offers several advantages, including faster domain decomposition and the potential for more frequent tree rebuilding, which, in turn, can increase the accuracy of the Barnes & Hut algorithm.

We decided to implement a parallel-friendly tree build algorithm that we will then connect to OPENGADGET3. For this purpose, we opt for an insertion-free strategy based on the 1D mapping of particles onto a Space Filling Curve (SFC). This strategy has been under development for a long time (see, for instance, [4], [5], [6], [7], [8], [9] and finally the recent SPH-EXA code [10]). We have developed a new Oct-Tree build algorithm based on the parallel tree build presented in Karras et al. [7], which consists of two parts. The first part is an embarrassingly parallel binary tree building based on the Peano-Hilbert (PH) SFC keys of particles (we use Hilbert ordering to preserve better spatial locality). The actual Oct-Tree is then built as a second step, exploiting two facts: (i) groups of 3 bits of a Hilbert key represent Oct-Tree cubes, and (ii) the amount of

common bits in the PH keys of binary-tree-edges provide information on the number of sub-cubes separating the two binary tree edges. A prefix sum is needed, and although we have parallelised it, it slightly lowers the ideal scaling of the tree build.

The complete tree build consists of the following phases: (i) particles sorting by their PH key, (ii) the construction of Box-Leaves (see next section) and (iii) the Karras algorithm for the Oct-Tree build via a Binary Tree helper.

We report here very preliminary results.

On a test case of 512^3 particles with a uniform distribution on a 3D volume, 1 thread completes the tree build with 8 particles per box-leaf in ~ 5.9 [s]. To this, it should be added the time needed for the building of the BoxLeaves themselves, which amounts to ~ 1 [s] and produces $\sim 2 \times 13^7$ leaves (see next section).

To evaluate this timing, let's remind the reader that in production runs a more standard occupation is $\sim 10^7$ particle/task.

We are not accounting here for the time needed to sort the particles by their PH key; however, that is an independent task for which we'll use the best helper function on CPU or GPU.

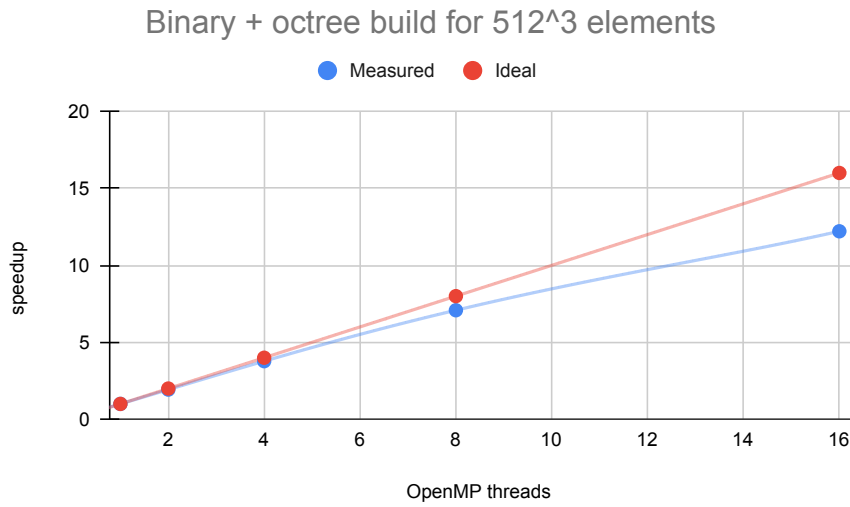


Figure 11: Speedup time of Karras binary tree and octree build.

OpenMP Threads	Build time [s]	Ideal time [s]	Efficiency
1	0.704	0.704	1.00
2	0.365	0.352	0.965
4	0.186	0.176	0.944
8	0.0994	0.0880	0.886
16	0.0576	0.0440	0.763

Table 5: Build time and efficiency scaling with OpenMP threads for the binary + octree build of box-leaves.

In Table 5, we report the OpenMP scaling of the test case with an increasing number of OpenMP threads. There, we see that the parallel efficiency (speedup over thread numbers) is relatively high up to 8 OpenMP threads and decreases slightly (reaching 0.76) as the number of threads increases to 16. We note that we spent most of the effort on writing the algorithm and producing a large number of tests (line of codes of testing routines amount to 45% of the total number of lines). We will devote future effort to improve its parallelism and to port it to GPUs using OpenACC. We report the values of the corresponding speedup in Figure 11.

3.3.3 Box-Leaves constructions

The tree-building algorithm that we briefly introduced in the previous section is agnostic to the objects it deals with. As such, we can apply it on single particle distribution, which would result in a complete Oct-Tree, or

on the Box-Leaves mentioned in Sec. 3.3.1. That renders easy the connection between this recent development and the new GPU-friendly Tree-Walk strategy.

However, we are left with the task of individuating the BL with an insertion-free algorithm. To solve this problem, we have developed an algorithm that, similar to the one for the Oct-Tree building, relies only on the PH key of the PH-sorted particle distribution.

We parallelize the finding of the BoxLeaves by high-level sub-volumes, so that every thread processes independent sub-volumes and, since the particles are sorted by their HP keys, different chunks of particles. By recording in a temporary array how many BoxLeaves are found in every sub-volume, in a second phase, we easily reconstruct the complete array of BoxLeaves for the entire volume.

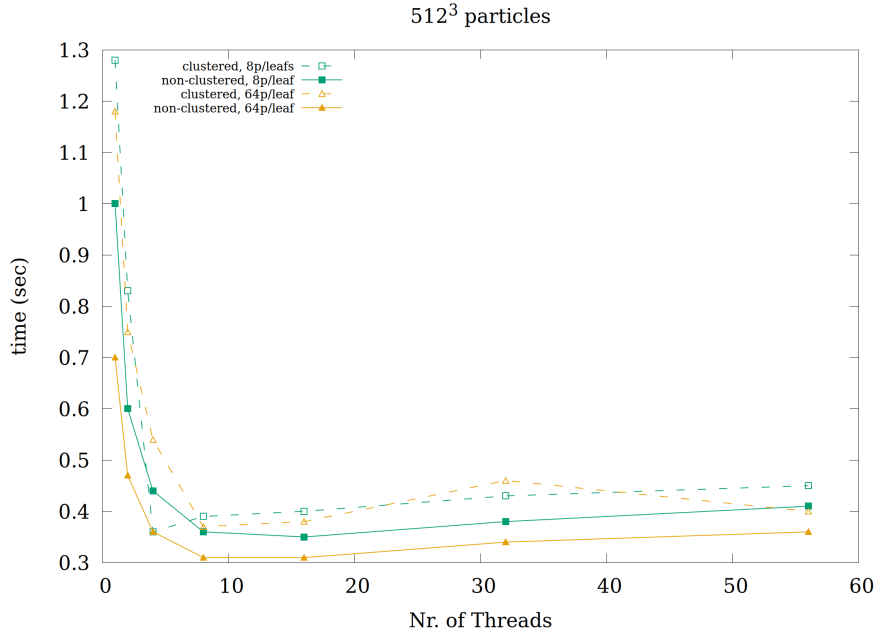


Figure 12: OPENGADGET3 scaling of BoxLeaves construction on a LEONARDO-DCGP’s node for 512^3 (512Million) particles, with 8 and 64 points per BoxLeaf (**green** and **yellow** lines, respectively), and for two different particle distributions: homogeneous (continuous lines) and strongly clustered (dashed lines; about 90% of particles into very concentrated knots).

Figure 12 shows the absolute timing of the BoxLeaves construction for 512^3 particles, for two different cases. A homogeneous distribution (continuous lines) and a strongly clustered one (dashed lines) in which $\sim 90\%$ of the particles live in very concentrated “knots”. A single thread ends the task in ~ 1 [s] and ~ 1.3 [s], respectively.

In a more realistic situation, with $\sim 10M$ particles per task, the run time drops to $\lesssim 0.1$ [s] and $\lesssim 0.15$ [s], respectively.

Our OPENMP parallelisation strategy involves considering the first n levels of the Oct-Tree —i.e., the first 8^n Peano sub-cubes —and assigning those sub-cubes to different threads. In this way, since the particles are ordered by their Peano keys, each thread processes a separate segment of the particle array, encountering independent leaf boxes. This approach should yield zero data races and, at the same time, minimize memory contention. However, as it is evident from Figure 12, the scalability is limited, and our preliminary analysis suggests that this is due to memory-boundness. At the moment of writing, all the loops over the particle’s key are *not* explicitly vectorized, and our first strategy will be to enhance the vectorization, which should increase the ratio `processed_keys / memory_accesses`.

We’ll devote much effort to the optimization of this algorithm and its implementation, as well as of the Oct-Tree building described in the previous section, since they are key components in our new approach.

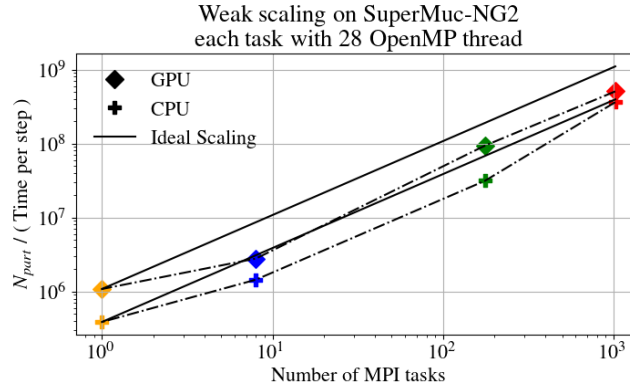


Figure 13: Scalability of OpenMP offloading in comparison with the CPU version.

3.4 OpenMP offloading

Additionally, alongside enhancing and extending the existing OpenACC GPU implementation, we have finished the testing and validation of the OpenMP offloading for the gravity module. This offloading has been integrated into the main branch, making it accessible to all collaborators and it will be included in upcoming public releases. Figure 13 shows the scalability of the OpenMP implementation compared to the CPU. These tests were conducted on SuperMUC-NG Phase 2 at LRZ in Garching near Munich. Each node in this system features an Intel Xeon Platinum 8480+ processor with 112 cores and 4 Intel Data Center GPU Max 1550 GPUs. The results demonstrate an increase in processed particles per unit time across various scenarios, showcasing nearly ideal scaling with available resources. Moreover, Figure 13 highlights that the OpenMP offloading achieves a speedup of approximately $3\times$ compared to the CPU version, which is similar to the performance seen with the OpenACC implementation. This level of speedup was expected, given the similarities between OpenACC and OpenMP offloading in terms of their core operations and data handling. The introduction of OpenMP offloading significantly enhances the versatility of the code across diverse hardware environments.

3.5 Conclusion

Benchmarking on a 128 Mpc/h cosmological box populated with 2×576^3 particles shows improved scalability in the latest version of the code, particularly beyond 32 nodes. Performance gains are especially evident in modules such as gravitational accelerations and domain decomposition, where optimisations have had generally positive effects.

A more detailed performance analysis of OPENGADGET3 was carried out on a smaller cosmological simulation using up to 16 nodes, with both gravity and hydrodynamics enabled. As the number of nodes increased, speedup efficiency dropped below 80% at 8 nodes, highlighting serialisation and OpenMP communication as primary bottlenecks.

Among all routines, DENS emerged as the most promising target for optimisation, due to its moderate serialisation efficiency (SerE, 86%) and OpenMP core efficiency (OCE, 80%), as well as its significant share of total runtime. Analysis revealed considerable MPI overhead (approximately 12% in MPI_Alltoall) and limited OpenMP scalability, especially in find_hsm1 and compute_unified_gradients, where fine-grained parallelism appears to be a limiting factor. In contrast, GRAV, the longest-running routine, showed excellent parallel efficiency.

On the development side, progress continues on a completely new tree and walk strategy. While still under active testing and evaluation, several core components of this approach are already operational and show promising potential.

Finally, the versatility of the code has been enhanced through the introduction of OpenMP-based offloading for the gravity module.

4 iPIC3D

iPIC3D [11] is a (semi-)implicit Particle-In-Cell code to solve the kinetic equations describing the dynamics of collisionless plasmas. At the core of the algorithm the code solves the non-linear Vlasov system

$$\partial_t f_\alpha + \mathbf{v} \cdot \nabla_{\mathbf{x}} f_\alpha + \frac{q_\alpha}{m_\alpha} (\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \nabla_{\mathbf{v}} f_\alpha = 0 \quad (2)$$

via discretizing the distribution functions f_α of each species α through computational markers – the so-called *particles*. As a plasma is a gas of charged particles the electric and magnetic fields in the Vlasov equation are self-induced by the movement of the charged particles and therefore in each time-step one has to solve the Maxwell's field equations for E and B . Solving the Maxwell's equations commonly is done on a grid in the physical domain and the source terms are gathered from the particles, which results in the procedure showcased in figure 14.

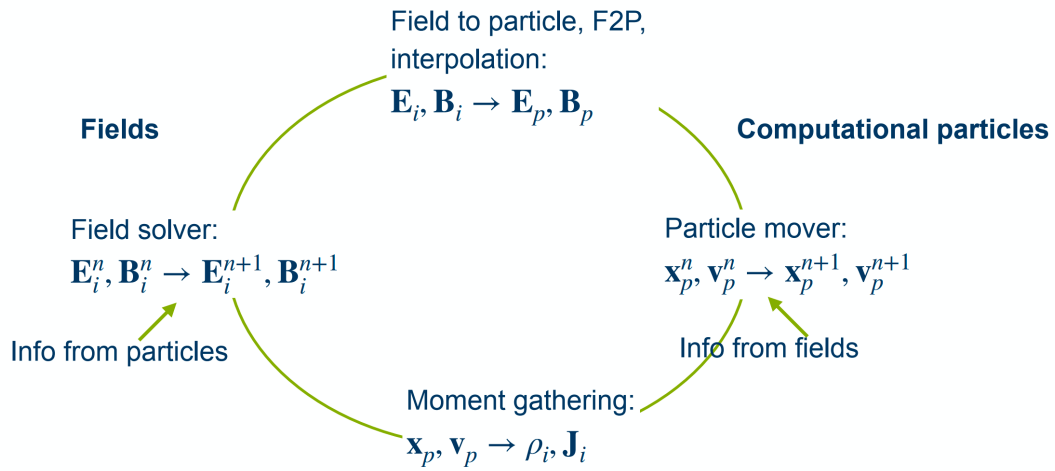


Figure 14: Flow-chart of the Particle-In-Cell (PIC) method.

Most PIC codes solve the coupling explicitly which introduces severe restrictions on the time-step size. To overcome this limitation the *implicit moment method* (IMM) has been introduced through iPIC3D [11]. The IMM solves the field equations implicitly, however, introduces a number of approximations in the computation of the source terms which allow to decouple the particles from the field solver and thereby reducing the dimension of the non-linear system for the field equations. This makes it tractable even in six-dimensional phase-space in contrast to fully-implicit PIC approaches.

The IMM is, however, no longer strictly energy-conserving which for some (astrophysical) setups can be a draw-back. Therefore iPIC3D also includes the *Energy-conserving Semi-Implicit method* (ECSIM) [12], which makes the IMM energy-conserving again. The main differences between the IMM and ECSIM are that while both employ uniform, staggered-in- space grids for the physical domain, ECSIM additionally staggers the values of the electric and magnetic field in time. Connected to this a slightly different approximation is chosen for the source-term computation in ECSIM, which leads to the introduction of small *mass matrices* on each cell, which are required to solve the field equations.

4.1 Code Structure

For both the implementation of IMM and ECSIM the code is structured into 3 submodules following the idea illustrated in figure 14. The modules are the particle mover `KCode.ParticlesMover`, moment calculation `KCode.CalculateMoments`, computation of the electric and magnetic fields `KCode.ComputeEMFields`.

For IMM the particle movement dominates the overall runtime while for ECSIM the moment gathering becomes more costly and therefore dominates the computational cost of the algorithm.

4.2 Use-case description

iPIC3D uses uniform grids in space and the same is also employed for the MPI-parallelization. A classical benchmark to assess the performance of a Particle-In-Cell code is starting a simulation from equilibrium, i.e., a Maxwellian distribution function for both electrons and ions, and considering the dynamics induced by the particle noise. Here we looked at cases with 2-dimensional physical and velocity space as well as 3-dimensional physical and velocity space. The boundary conditions are always chosen to be periodic. A sample input file is given in the following:

```
# INPUT FILE for thermal plasma case
# 2 Species

# %%%%%%%%%% Input/Output flags %%%%%%%%%%
SaveDirName      = data          # Output directory
RestartDirName = data          # Restart directory (for WriteMethod=default)

# %%%%%%%%%% Input/Output flags %%%%%%%%%%
Case              = default      # Case [ GEM | Dipole | ... ]
WriteMethod       = pvtk         # Output method [ default | Parallel ]
PoissonCorrection = no          # Poisson correction [ yes | no ]
SimulationName    = scaling      # Simulation name for the output

# %%%%%%%%%% Magnetic Reconnection %%%%%%%%%%
B0x = 0.00
B0y = 0.00
B0z = 0.00

# External magnetic field parameters:
B1x = 0.000
B1y = 0.000
B1z = 0.000

delta = 0.5

# %%%%%%%%%% TIME %%%%%%%%%%
dt = 0.1          # dt = time step
ncycles = 30      # cycles
th = 0.5          # th = decentering parameter
c = 1.0           # c = light speed

# %%%%%%%%%% SMOOTH %%%%%%%%%%
Smooth = 1.0      # Smoothing value (5-points stencil)

# %%%%%%%%%% BOX SIZE %%%%%%%%%%
Lx = 40.0         # Lx = simulation box length - x direction
Ly = 40.0         # Ly = simulation box length - y direction
Lz = 1.0          # Lz = simulation box length - z direction

x_center = 20.0   # Lx = simulation box length - x direction in m
y_center = 20.0   # Ly = simulation box length - y direction in m
z_center = 0.5    # Lz = simulation box length - z direction in m
L_square = 0.1

nxc = 1024        # nxc = number of cells - x direction
nyc = 1024        # nyc = number of cells - y direction
nzc = 1           # nzc = number of cells - z direction
```

```

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% MPI TOPOLOGY %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
XLEN = 64                # Number of subdomains in the X direction
YLEN = 32                # Number of subdomains in the Y direction
ZLEN = 1                 # Number of subdomains in the Z direction

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PARTICLES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#      0 = electrons
#      1 = protons
#      2,3,4,5,... = ions
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ns = 2                   # Number of particles
rhoINIT      = 0.5   0.5 # Initial density (make sure you are neutral)
rhoINJECT    = 0.0   0.0 # Injection density (make sure you are neutral)
TrackParticleID = 0    0  # TrackParticleID[species] = 1=true, 0=false

npcelx      = 110   110  # Particles per cell in X
npcely      = 110   110  # Particles per cell in Y
npcelz      = 1     1    # Particles per cell in Z
NpMaxNpRatio = 5.0      # Maximum number of particles allocated

qom          = -100.0    1.0 # Charge/mass ratio
uth          = 0.0      0.00 # Thermal velocity in X
vth          = 0.0      0.00 # Thermal velocity in Y
wth          = 0.0      0.00 # Thermal velocity in Z
u0           = 0.0      0.0   # Drift velocity in X
v0           = 0.0      0.0   # Drift velocity in Y
w0           = 0.0      0.0   # Drift velocity in Z

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Periodicity in each direction %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PERIODICX    = 1          # In direction X (1=true, 0=false)
PERIODICY    = 1          # In direction Y (1=true, 0=false)
PERIODICZ    = 1          # In direction Z (1=true, 0=false)

### ===== ###

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% boundary conditions %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# If the PERIODIC flag is active in the previous section
# periodic boundary conditions will be imposed
#
# PHI Electrostatic Potential
# 0,1 = Dirichlet boundary condition ;
# 2   = Neumann boundary condition
bcPHIfaceXright = 1
bcPHIfaceXleft  = 1
bcPHIfaceYright = 1
bcPHIfaceYleft  = 1
bcPHIfaceZright = 1
bcPHIfaceZleft  = 1

# EM field boundary condition
# 0 = perfect conductor
# 1 = magnetic mirror
bcEMfaceXright = 0
bcEMfaceXleft  = 0
bcEMfaceYright = 0

```

Cluster	Spatial Resolution	Particles per cell and per species
LUMI	1024×1024	110×110
Karolina	256×256	110×110
Leonardo	280×256	120×120

Table 6: Base resolutions for the scaling measurements of 2D simulations with IMM.

```
bcEMfaceYleft = 0
bcEMfaceZright = 0
bcEMfaceZleft = 0
```

```
# Particles Boundary condition
# 0 = exit
# 1 = perfect mirror
# 2 = riemission
bcPfaceXright = 1
bcPfaceXleft = 1
bcPfaceYright = 1
bcPfaceYleft = 1
bcPfaceZright = 1
bcPfaceZleft = 1
```

```
### ===== ###
```

```
# %%%%%%%%% Numerics options %%%%%%%%%
verbose = 0 # Print to video results
Vinj = 0.0 # Velocity of the injection from the wall
CGtol = 1E-3 # CG solver stopping criterium tolerance
GMREStol = 1E-8 # GMRES solver stopping criterium tolerance
NiterMover = 3 # mover predictor corrector iteration
FieldOutputCycle = 0 # Output for field
ParticlesOutputCycle = 0 # Output cycle for particles
RestartOutputCycle = 0 # restart cycle
DiagnosticsOutputCycle = 0 # Diagnostics cycle
```

4.3 Scaling of IMM and ECSIM

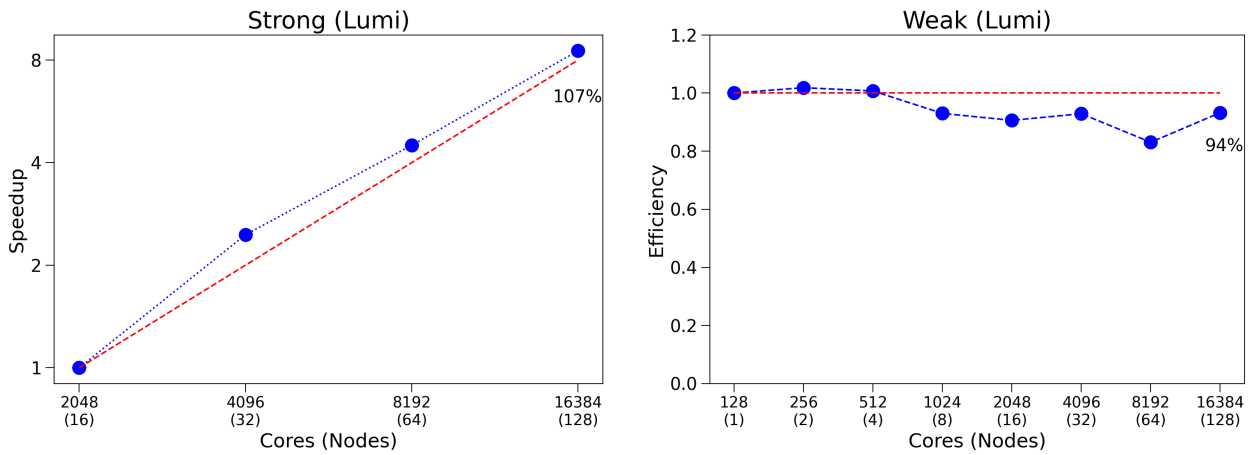
In our first performance evaluation, reported in D2.1, we observed that while in principle the scalability of the submodules of the initial IMM implementation were good, the communication and load-balancing was an issue. In particular a POP analysis showed that the communication efficiency deteriorated down to 13 % for runs with up to 8192 MPI-processes, which resulted in a poor overall parallel efficiency. The reason was that the communication was unnecessarily synchronized at several points in the code. Switching to asynchronous communication led to significant improvement of parallel efficiency for IMM as was reported in short in D2.3.

Since then we additionally ported the old version of ECSIM into the new iPIC3D codebase while also switching this implementation to asynchronous communication. In this report we can now show the scaling results for both the new version of IMM and ECSIM on the clusters *LUMI*, *Karolina* and *Leonardo*.

4.3.1 Scaling of IMM

First we present scaling results for 2D simulation with IMM on LUMI, Karolina and Leonardo. The base resolutions for the simulations can be found in table 6.

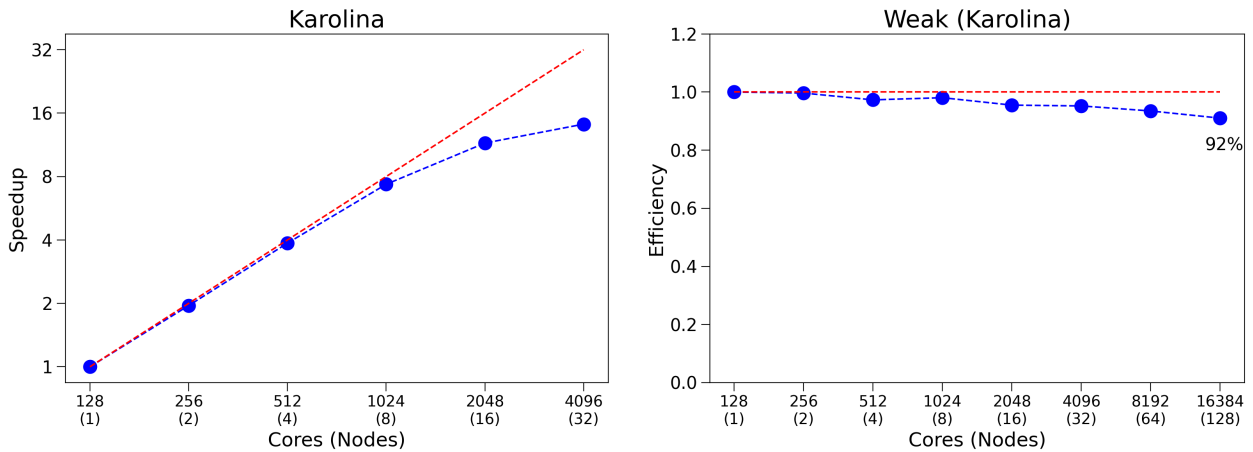
In Figure 16 we show the strong and weak scaling of the new implementation of the IMM algorithm on LUMI. We observe a close to perfect strong scaling for up to 128 nodes in this case. The weak scalability exceeds 90 % up to 128 nodes as well. The slight deviations from the expected behavior of the curves is likely due to system-specific fluctuations and are well within the acceptable tolerance.



(a) Strong scaling of IMM in iPIC3D on LUMI for up to 128 nodes. (b) Weak scaling of IMM in iPIC3D on LUMI for up to 128 nodes.

Figure 15: Scaling of IMM in iPIC3D on LUMI with a 2D case

On Karolina we again observe good weak scaling, however, the strong scalability deteriorates somewhat for more than 8 nodes down to about 50 % for 32 nodes. The weak scalability exceeds 92 % for all node-counts.



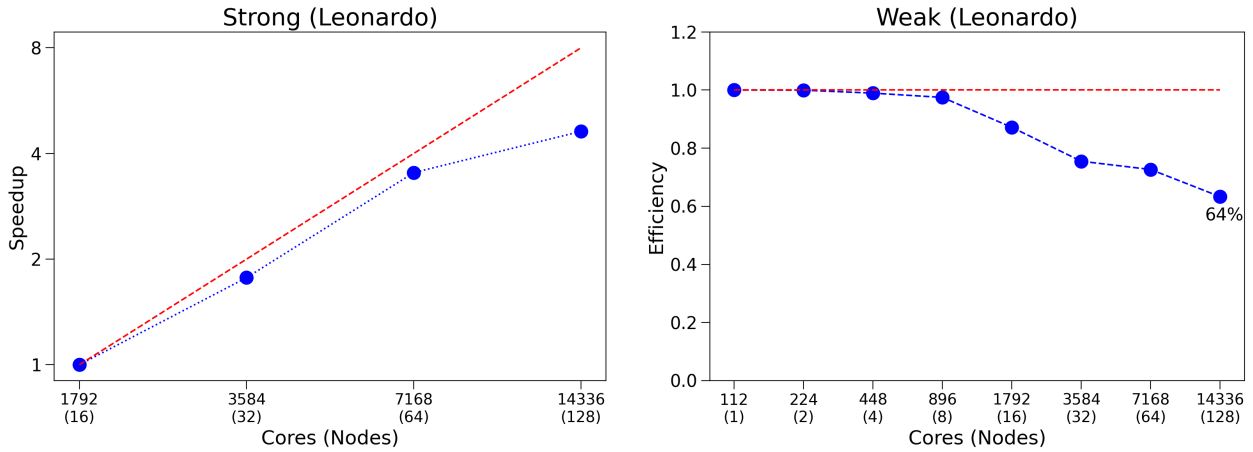
(a) Strong scaling of IMM in iPIC3D on Karolina for up to 128 nodes. (b) Weak scaling of IMM in iPIC3D on Karolina for up to 128 nodes.

Figure 16: Scaling of IMM in iPIC3D on Karolina with a 2d case

Finally on Leonardo we observe substantially worse scalability than for the other two previously mentioned clusters. The strong scalability is good for up to 32 nodes and deteriorates down to about 60 % for up to 128 nodes. The weak scalability deteriorate down to about 65 % for 128 nodes.

Number nodes	Spatial Resolution	Particles per cell and per species
1-4	512×256	80×80
8-32	1024×512	80×80
64-256	4096×2048	80×80

Table 7: Resolutions for the strong scaling measurements of 2D simulations with ECSIM on LUMI.



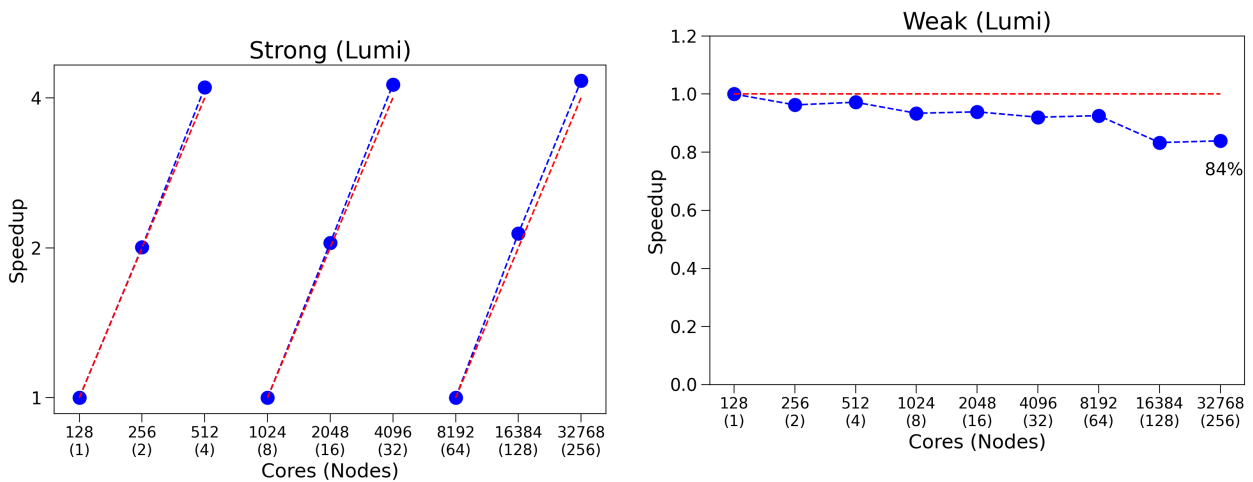
(a) Strong scaling of IMM in iPIC3D on Leonardo for up to 128 nodes. (b) Weak scaling of IMM in iPIC3D on Leonardo for up to 128 nodes.

Figure 17: Scaling of IMM in iPIC3D on Leonardo with a 2d case

4.3.2 Scaling of ECSIM

Next we also looked at the scalability of our new implementation of ECSIM on LUMI and Karolina. This time we looked at the scaling for up to 256 nodes, however, to make the scaling results reflect realistic usage of the code we decided to take 3 baselines for 1 to 4, 8 to 32 and 64 to 256 nodes each, where for each new baseline with higher node we increase the number of grid cells per direction.

First we consider the scalability of the new ECSIM code for a 2D simulation in figure 18. The resolutions can be found in the table 7. Here the strong scalability for all 3 baselines well above 95 % and the weak scalability remains above 84 % for up to 256 nodes.



(a) Strong scaling of ECSIM on LUMI for up to 128 nodes. (b) Weak scaling of the ECSIM on LUMI for up to 128 nodes.

Figure 18: Scaling of the ECSIM on LUMI with a 2D case.

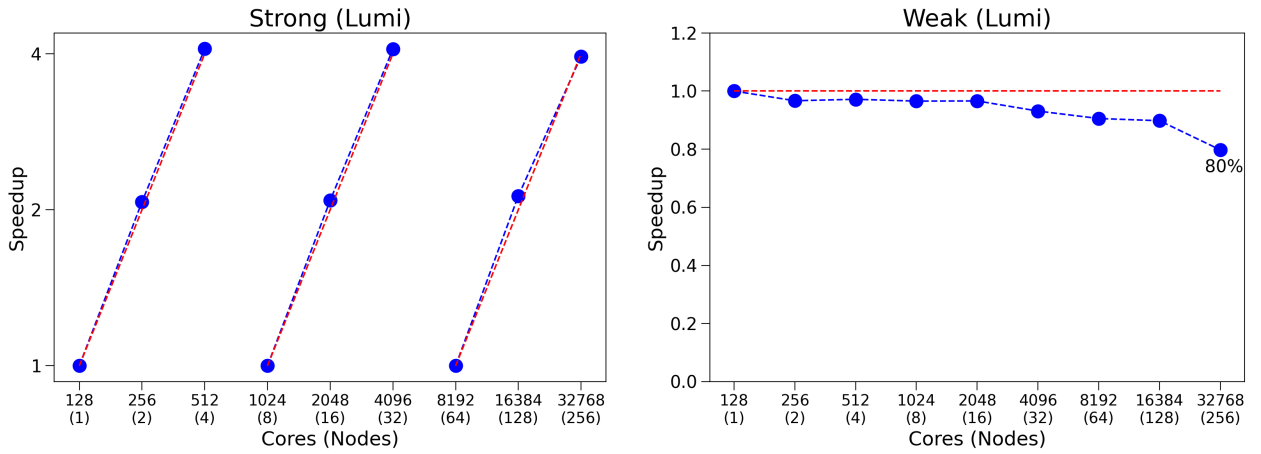
Number nodes	Spatial Resolution	Particles per cell and per species
1-4	$64 \times 32 \times 32$	$20 \times 20 \times 20$
8-32	$128 \times 64 \times 64$	$20 \times 20 \times 20$
64-256	$256 \times 128 \times 128$	$20 \times 20 \times 20$

Table 8: Resolutions for the strong scaling measurements of 3D simulations with ECSIM on LUMI.

Number nodes	Spatial Resolution	Particles per cell and per species
1-8	256×256	120×120
16-128	1024×1024	120×120

Table 9: Resolutions for the strong scaling measurements of 2D simulations with ECSIM on Karolina.

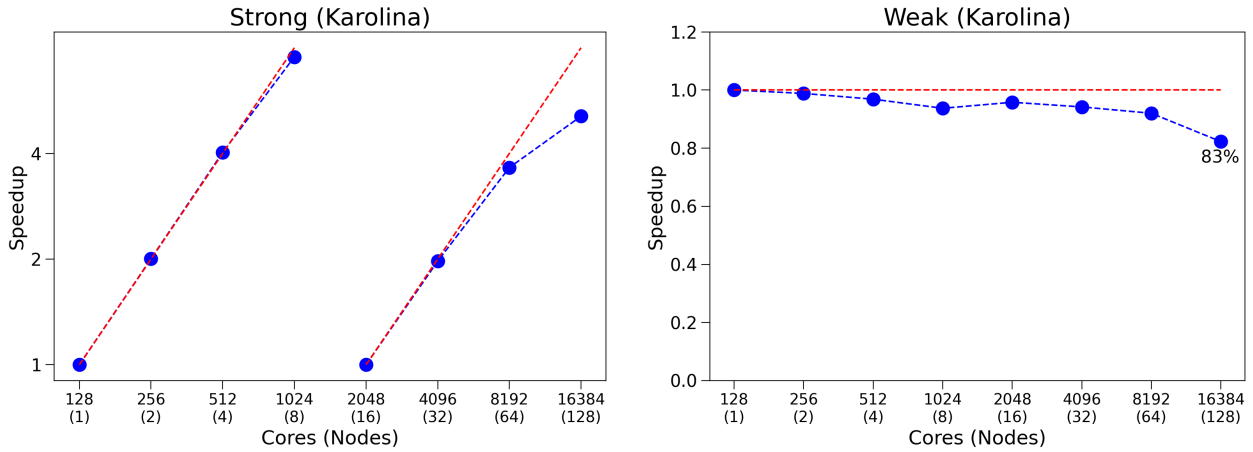
Next we considered a 3D simulation with ECSIM, see figure 19. The resolutions can be found in the table 8. Similar to the 2D case we observe strong scalability above 95 % and the weak scalability remains above 80 % for up to 256 nodes.



(a) Strong scaling of a 3D simulation with ECSIM on LUMI for up to 128 nodes. (b) Weak scaling of a 3D simulation with ECSIM on LUMI for up to 128 nodes.

Figure 19: Scaling of the ECSIM on LUMI with a 3D case.

Finally we also ran 2D simulations on Karolina with ECSIM, see figure 20. The resolutions can be found in the table 9. The strong scaling is good with over 95 % efficiency for up to 64 nodes, however, deteriorates down to about 65 % for 128 nodes. The weak scalability remains above 83 % for up to 128 nodes.



(a) Strong scaling of a 2D simulation with ECSIM on Karolina for up to 128 nodes. (b) Weak scaling of a 2D simulation with ECSIM on Karolina for up to 128 nodes.

Figure 20: Scaling of a 2D simulation with ECSIM on Karolina.

4.4 Comparison to initial version

Before we move on to discussing the improvement in computational time, we want to show case in table 10 that through removing redundant memory allocations the new ECSIM version now allows to use higher resolutions on the same hardware.

Resolution	ECSIM (new) (with ECSIM)	ECSIM	Ratio
2^{24} (1D)	390	230	1.7
2048^2 (2D)	48^2	33^2	2.1
$128^2 \times 64$ (3D)	21^3	17^3	1.9
8192×4096 (2D)	48^2	30^2	2.5
128^3 (3D)	33^3	25^3	2.3

Table 10: Number of maximum particles for one species of particles per MPI process for a given grid resolution on 8 (left) and 64 (right) nodes.

Next we looked at the speed-up one can achieve with the new version of ECSIM over the old implementation for a 2D simulation, see table 11. The simulations were ran on Karolina. We ran 20 time cycles with a base resolution for 8 nodes of 2048×2048 spatial cells and 2 species with 30×30 particles per cell. The total computational time is dominated by the moment gatherer, which uses more than 85 % of the total computational time for the new version. Therefore even though the speed-ups for the field solver and particle mover were between 30 and 300 %, the overall speed-up was limited to about 20-35 % by the limited speed-up of the moment gatherer.

Next we considered 3D simulations with base resolution $128 \times 64 \times 64$ and 2 particle species with $20 \times 20 \times 20$ particles per cell and again the first 20 time cycles. The results are quite comparable to the 2D case: Again the total run-time is dominated by the moment gatherer and the speed-up achieved there reflects in the total speed-up. Again the speed-up for the entire simulation ranges between 25 and 37 % depending on the number of nodes.

4.5 Profiling activities

4.5.1 CPU performance assessment with Extrae in collaboration with POP3 CoE

We consider the test case of a Maxwellian distribution with 95^2 particles per cell with 2 species on a 2D grid with a resolution of 2240×1120 cells. It was limited to only 4 cycles on 2 to 64 nodes, with the second cycle being analysed as the RoI. The strong scaling characteristic in Figure 21 reveals a clear speedup degradation on

# of nodes	Particle Mover runtime [s]			Moment Gatherer runtime [s]		
	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
8	50	35	40	363	307	20
32	75	53	40	546	464	17
64	50	38	30	382	320	20
# of nodes	Field Solver runtime [s]			Total runtime [s]		
	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
8	12	8.4	40	427	360	20
32	279	140	100	905	670	35
64	92	52	80	528	420	25

Table 11: Benchmarking results of ECSIM for a 2D thermal plasma case.

# of nodes	Particle Mover runtime [s]			Moment Gatherer runtime [s]		
	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
8	34	19	80	321	259	20
32	47	25	90	453	368	20
64	48	24	100	416	309	35
# of nodes	Field Solver runtime [s]			Total runtime [s]		
	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
8	1.14	0.48	140	360	286	25
32	4.2	1.6	160	508	407	25
64	4.8	1.2	300	472	345	37

Table 12: Benchmarking results of ECSIM for a 3D thermal plasma case.

64 nodes (7168 processes). The efficiency metrics in Figure 22 point to drop in Transfer efficiency (TE) caused by an extreme growth of time in MPI communication and its latency, as was discovered later.

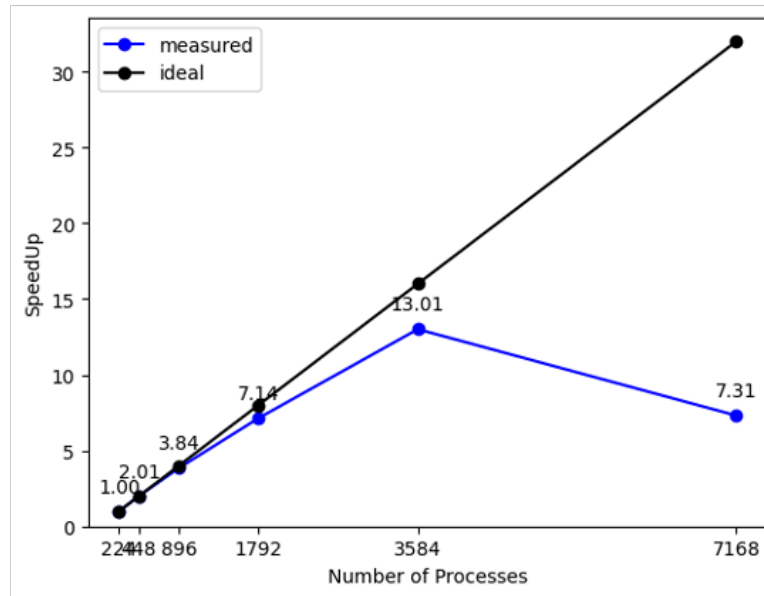


Figure 21: iPIC3D on Leonardo DCGP for up to 64 nodes - strong scaling.

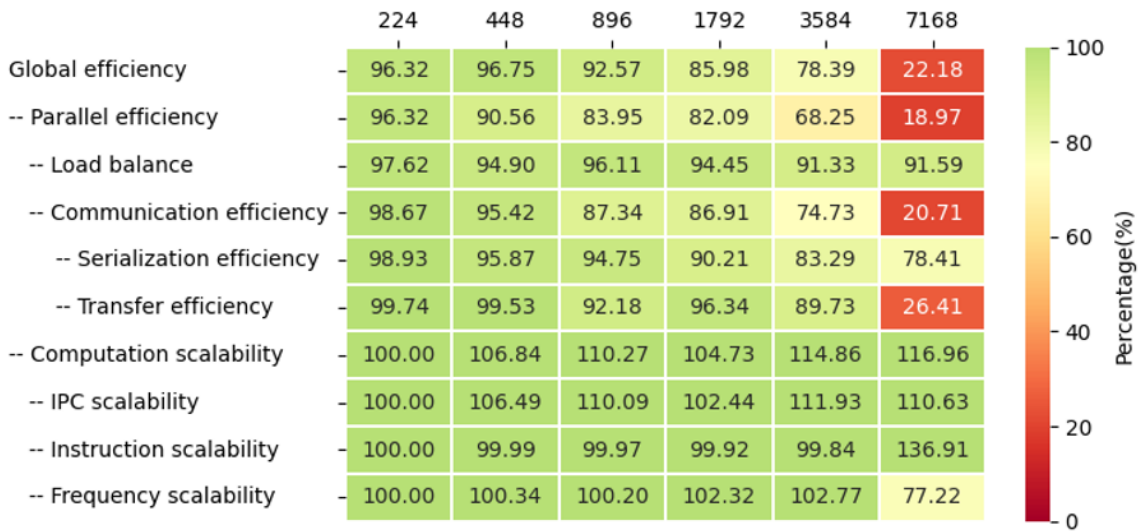


Figure 22: iPIC3D on Leonardo DCGP for up to 64 nodes - Efficiency metrics [%].

Each cycle (RoI) is composed by the three high-level routines: CalculateField, ParticlesMover, and GatherMoments. The strongest TE deterioration at 64 nodes comes from CalculateField with 2%, followed by GatherMoments with 41%. That 2% TE together with 20% of Instruction scalability and other inefficiencies translates, in practice, into 0.2x speedup, i.e. 5x slowdown of the routine compared to the base run. This was caused by an excessive use of MPIBarriers bounding sequences of very small MPI_Sendrecv_replace messages with no computation in between, resulting in significant accumulated latency of the MPI calls. Conversely, the ParticlesMover showed very good efficiencies and almost perfect scaling. In all routines, very high sensitivity for the system preemption was also detected.

These analyses help the code owners to further optimize their code. The next step is to select and implement a suitable optimization technique for each of the performance issues identified above.

4.6 Conclusion

Since the start of the project we ported the old version of the IMM algorithm present in the old version of iPIC3D code into the new iPIC3D codebase. This involved reevaluating data structures and the way inter-node communication was performed. A major change was moving from synchronous to asynchronous communication between MPI processes which for IMM overcame major load-balancing issues. In this report we presented scaling results on LUMI, Karolina and Leonardo to show case the improvements in scalability over the previous version of IMM. Additionally, since the last reports D2.1 and D2.3 we moved the algorithm ECSIM from the old codebase into the new iPIC3D codebase. In particular, the new ECSIM implementation now also benefits from the improved data structures and changed communication paradigm.

The strong and weak scaling results presented in this report for both IMM and ECSIM show promising scaling for both algorithms even though they are strongly-memory bound. For high node-counts, in particular, on Leonardo we still observed a reduction in scalability which now remains to be investigated and could be a hint towards further potential for improvement.

The new version of ECSIM not only allows to use substantially higher resolutions compared to the old version, it also is between 20 and 30 % faster for the considered benchmark cases. Since most of the computational cost is incurred during the moment-gathering phase, we will investigate whether further implementational or algorithmic optimizations are possible.

5 RAMSES

RAMSES [13, 14] is an Adaptive-Mesh-Refinement (AMR) code which is used to study astrophysical fluid dynamics and the formation of structures in the Universe. It is based on an oct-tree structure, where parent cells are refined into children cells on a cell-by-cell basis following some user-defined criteria. RAMSES can deal with 1D, 2D and 3D Cartesian grids.

RAMSES integrates the equations of self-gravitating fluid dynamics in the form

$$\begin{cases} \partial_t \rho + \nabla \cdot [\rho \mathbf{u}] &= 0, \\ \partial_t \rho \mathbf{u} + \nabla \cdot [\rho \mathbf{u} \otimes \mathbf{u} + P \mathbb{I}] &= -\rho \nabla \phi, \\ \partial_t E + \nabla \cdot [\mathbf{u} (E + P)] &= -\rho \mathbf{u} \cdot \nabla \phi, \\ \Delta \phi &= 4\pi G \rho \end{cases} \quad (3)$$

where ρ is the density, $\mathbf{u}$ the velocity, ϕ is the gravitational potential, E the total energy, i.e. $E = e + 1/2 \rho u^2$, with $e = P/(\gamma - 1)$ the internal energy. Without self-gravity, this system can be written in the canonical form

$$\frac{\partial \mathbb{U}}{\partial t} + \nabla \cdot \mathbb{F}(\mathbb{U}) = 0, \quad (4)$$

where the vector $\mathbb{U} = (\rho, \rho \mathbf{u}, E)$ contains the conservative variables. The flux vector $\mathbb{F}(\mathbb{U}) = (\rho \mathbf{u}, \rho \mathbf{u} \otimes \mathbf{u} + P \mathbb{I}, \mathbf{u} (E + P))$ is a linear function of $\mathbb{U}$, and uses the primitive variables ρ , $\mathbf{u}$ and P . This system is closed using a perfect gas equation of state.

RAMSES uses an explicit second-order predictor-corrector finite-volume Godunov scheme to integrate the conservative system of equations. The hydrodynamic solver consists in computing flux at cell's interfaces, dealing with coarse-to-fine and fine-to-coarse interfaces. The discretized scheme writes (here in 1D for simplicity)

$$\frac{\mathbb{U}_i^{n+1} - \mathbb{U}_i^n}{\Delta t} \times V_i = \mathbb{F}_{i+1/2}^{n+1/2} S_{i+1/2} - \mathbb{F}_{i-1/2}^{n+1/2} \times S_{i-1/2}, \quad (5)$$

where $\mathbb{U}_i^n$ is the state variable $\mathbb{U}$ at time n and averaged in the cell i of volume V_i . $\mathbb{F}_{i-1/2}^{n+1/2}$ is the flux at the interface between cells i and $i - 1$, computed using a linear Riemann solver. The initial values of the Riemann problems are obtained from the primitive variables of cells $i - 1$ and i , extrapolated in time and space at the interface in the predictive step. Altogether, the scheme follows the same steps as the PLUTO code.

For gravity, RAMSES solves the Poisson equation on a level-by-level basis. Two methods can be chosen by the user: a classical conjugate gradients and a multigrid solver [15]. In this deliverable, we focus on the optimization made on the multigrid solver only.

RAMSES also integrates dark matter dynamics, coupled with the gas via the gravity. The dark matter solver uses a N-body Cloud-In-Cell (CIC) method to interpolate particle mass from the particle to the grid and to interpolate the gravitational acceleration of the particles from the grid.

The time integration can be also accelerated using the adaptive-time-step implementation, in which each AMR level evolve with its own time-step which satisfies a global synchronisation point at the end of the coarser time-step.

The development version of RAMSES we use in SPACE is now a fork of the main RAMSES organisation repository. It can be found here [RAMSES SPACE](#). Once a development is validated, a pull request is pushed to the main RAMSES organisation repository. All commit tags we give in the following are the one referring to the RAMSES organisation repository.

5.1 Use-case description

The relevant use-cases for this report are the first and second one presented in Deliverable 1.1. The first use-case (sedov) tests the hydrodynamical solver only, while the second (cosmo) tests the gravity and particle-in-cell solvers.

5.1.1 Sedov use-case

The classic Sedov-blast test is the first use-case, run in 3D with a uniform grid in order to focus on the hydrodynamics solver, without any overhead due to adaptive mesh refinement. The physical setup consists of a uniform density and pressure medium at rest (zero velocity) in which an internal energy pulse is put in the

additional GNU compiler flags	gain [%]
-march=native	14.2
-march=native -mtune=native	11.8
-march=native -funroll-loops	11.4
-march=native -flto	16.4
-march=native -flto -fno-automatic	14.9
-march=native -flto -fwhole-program	17.0

Table 13: Performance gain for adding additional compiler flags for RAMSES for the sedov use-case. Value obtained by taking the average over 3 executions and comparing to the execution when compiled with only the default optimization flags in the Makefile (-O3). Benchmarks performed on MeluXina for the GNU compiler.

corners of the computational domain. The internal energy is then progressively converted into kinetic energy via total energy conservation. For this test, the Lax-Friedrich Riemann solver is used. The simulation is evolved for 20 time steps. The uniform grid consists of 256^3 , 512^3 , 1024^3 or 2048^3 cells, depending on the computational and memory load we want to put on the resources. For this deliverable, we use 1024^3 .

5.1.2 Cosmo use-case

The second use-case is a cosmological box with only dark matter particles, that is without hydrodynamics. As in the Sedov use-case, the underlying grid of 1024^3 cells is kept uniform without AMR. This allows us to focus on the performance of the gravity and particle-in-cell algorithm (PIC). We evolve the simulation for 10 time steps.

5.2 Compiler options for the reference commit

The commit of the main version of the RAMSES code that entered the SPACE project is 8c72f569 (Add dev guide), from 8 November 2024. The first task is to determine the optimal value of NVECTOR for the different use-cases on the various clusters and to choose which compiler optimization flags to use.

5.2.1 Sedov use-case

A first investigation of NVECTOR for Sedov, performed on MeluXina and MareNostrum, indicates that NVECTOR=8 gives good results for the execution time of the Godunov solver. We use this value in our further investigation of compiler flags. Our tests show that using the gnu compiler (modules GCC/13.3.0 and OpenMPI/5.0.3-GCC-13.3.0 on meluxina), the flags `-march=native -flto -fwhole-program` produce the most efficient code out of the tested combinations. Remark that, if compiled on nodes which a different architecture than the compute nodes, native should be replaced by the compute node architecture.

Re-evaluating the value for NVECTOR with the chosen compilation flags now lead to NVECTOR=8 and NVECTOR=16 being optimal values for MeluXina, and in extension the other clusters use AMD EPYC 7H12 processors. Remark that the default value of 32 results in code which is about 50% slower.

5.2.2 Cosmo use-case

For the cosmo use-case, we determined an optimal NVECTOR=16.

5.3 Benchmarking results: OpenMP implementation

To improve the strong scaling of RAMSES, we introduce a multi-threading parallelism level, turning the code into a hybrid MPI+OpenMP code. This enables the use of fewer MPI processes, and hence less subdomains, to exploit a larger number of cores to solve the same problem. Each MPI process will run a certain number of OpenMP threads, so that `num processes x num threads` equals the number of cores (or hyperthreads) on the system.

We completed the OpenMP implementation for the full Sedov use-case, and the Poisson multigrid solver, which is one of the three major modules used by the cosmo use-case. The results are presented in the remainder of this section. The OpenMP implementation is being developed on a separate branch which is kept up-to-date

with the other developments. The results listed here add OpenMP on top of commit 9c518f8 of the dev branch (16 April 2025, move `lll` and `mmm` to `amr_constants`). The contributions between the SPACE reference commit and this one are mainly focused on cleaning up various parts of the code, adding tests and documentation.

5.3.1 Sedov use-case

We added OpenMP to the Godunov solver and a few other smaller subroutines that are called in the Sedov test case. The majority of the execution time is spent in the Godunov solver, with only a minor contribution of the MPI communication. So we do not expect a large gain in terms of execution time for this test case. Indeed, in Table 14 we see that only when using 64 nodes (on MeluXina) we have a speedup compared to the MPI-only implementation. We are still investigating ways to improve the performance of our implementation on a lower number of nodes.

The main advantage of using OpenMP+MPI is the major reduction in memory requirements to store the computation domain, due to fewer boundary grids. For example, on 32 nodes with 4 threads the memory required is only a third of what it was with mpi-only. RAMSES simulations performed with uniform grids are often forced to be launched on more nodes than computationally required, simply to be able to fit into memory. A trick used by researchers is to use the full memory of the node, but only half of the available cpus. These types of simulations would thus benefit greatly from having OpenMP.

5.3.2 Cosmo use-case: poisson multigrid solver

The Poisson multigrid solver is one of the most time-consuming parts in the cosmo use-case. The measurements of execution time comparing the MPI-only and MPI+OpenMP implementation can be found in Table 15. We performed tests with 2, 4 or 8 openMP threads. The optimal is found to be 4.

Because the main strength of the hybrid parallelization strategy is to reduce the amount of explicit MPI-communication, the gain increases with the number of nodes. When using 64 nodes, the time spent in the multigrid solver is cut in half when using 4 openMP threads per MPI rank. For the MPI-only implementations, the strong scaling parallel efficiency of this module is 34% at 64 nodes. Our new MPI+OpenMP implementation increases this to 67%.

5.4 Benchmarking results: cpu optimisations

In Table 16, we show the progress of the cpu optimisation of the RAMSES code for the Sedov and cosmo test case. This excludes the OpenMP developments, which were discussed in section 5.3. The times listed under commit 1 are for the reference commit described in section xxx, including the compiler optimizations. Commit 2 is the current state of the SPACE development branch. It contains an optimisation for the routines that search neighboring cells, something which is used extensively in many kernels.

5.5 Profiling activities

5.5.1 Sedov use-case

We performed profiling with MAQAO for the Sedov use-case to identify possible low-level optimizations. The commit used was 9c518f8 of the dev branch (as for the OpenMP results). The profiling was performed on the Meluxina cluster, where we benchmark compilation with GNU. We run the version with 1024^3 cells for 20 time-steps on 64 cores (half a node). MAQAO requires the code to be compiled with the `-g` flag. We also remove the `-fno` options, since they confuse the call tree information.

Figure 24 shows the call tree for the Godunov solver, where we specify the percentage of time spent in each subroutine (excluding calls to other subroutines) for the sedov use-case on a uniform grid. We identified several candidates for optimization.

godfine1 The majority of the time for this subroutine is spent gathering the hydro variables into a contiguous work array which is later sent to the calculation subroutines:

```
do ivar=1,nvar
  do i=1,nexist
    uloc(ind_exist(i),i3,j3,k3,ivar)=uold(ind_cell(i),ivar)
  end do
  do i=1,nbuffer
```

	SEDOV entire code runtime [s]				
	MPI only	MPI+OpenMP			gain [%]
# of nodes		2 threads	4 threads	8 threads	(best)
1	168.226	178.926	177.241	176.937	-5.2
2	86.504	89.889	89.495	90.455	-3.5
4	43.842	45.678	45.339	45.891	-3.4
8	21.981	23.405	23.485	23.674	-6.5
16	11.503	12.081	12.122	12.360	-5.0
32	6.080	6.380	6.478	7.068	-4.9
64	3.434	3.407	3.731	3.379	1.6

	SEDOV hydro-godunov [s]				
	MPI only	MPI+OpenMP			gain [%]
# of nodes		2 threads	4 threads	8 threads	(best)
1	135.021	147.390	147.141	146.958	-8.8
2	68.222	73.812	73.314	73.936	-7.5
4	34.257	36.912	36.753	36.965	-7.3
8	17.064	18.576	18.488	18.547	-8.3
16	8.595	9.290	9.371	9.347	-8.1
32	4.355	4.686	4.699	4.734	-7.6
64	2.169	2.379	2.384	2.388	-9.7

	SEDOV hydro-ghostzones [s]				
	MPI only	MPI+OpenMP			gain [%]
# of nodes		2 threads	4 threads	8 threads	(best)
1	2.117	1.689	1.698	2.076	20.2
2	1.427	1.168	1.175	1.365	18.2
4	0.894	0.781	0.771	0.885	13.8
8	0.608	0.515	0.515	0.629	15.2
16	0.436	0.332	0.357	0.406	23.8
32	0.300	0.293	0.301	0.428	2.3
64	0.313	0.238	0.270	0.174	44.2

	SEDOV hydro-revghostzones [s]				
	MPI only	MPI+OpenMP			gain [%]
# of nodes		2 threads	4 threads	8 threads	(best)
1	4.698	4.829	4.528	3.947	16.0
2	3.374	2.333	2.938	3.117	30.9
4	2.249	1.693	1.695	1.962	24.7
8	1.100	1.219	1.405	1.395	-10.8
16	0.753	0.879	0.830	0.984	-10.2
32	0.450	0.542	0.565	0.998	-20.4
64	0.327	0.329	0.527	0.390	-0.6

Table 14: Comparison of the execution time of the MPI-only version of the Sedov use-case in RAMSES to the new MPI+OpenMP implementation. Benchmarks performed using the Sedov use-case on a 1024^3 grid for 20 steps, on the MeluXina cluster. The reported numbers are the average across 3 identical runs.

COSMO entire runtime [s]					
	MPI only	MPI+OpenMP			gain [%]
# of nodes		2 threads	4 threads	8 threads	(best)
1	270.249	329.966	452.634		-22.1
2	138.427	168.598	228.456		-21.8
4	71.319	85.927	116.021	179.933	-20.5
8	38.424	44.307	58.893	91.787	-15.3
16	21.587	22.965	29.480	46.686	-6.4
32	13.886	12.413	15.470	24.319	10.6
64	10.746	7.311	8.439	13.698	32.0

COSMO Poisson multigrid solver runtime [s]					
	MPI only	MPI+OpenMP			gain [%]
# of nodes		2 threads	4 threads	8 threads	(best)
1	212.197	218.225	226.672		-2.8
2	109.185	112.498	116.584		-3.0
4	57.532	57.550	59.615	64.388	-0.0
8	31.667	30.175	30.416	33.847	4.7
16	18.751	16.041	15.086	17.313	19.5
32	12.621	8.868	8.209	9.415	35.0
64	9.763	5.786	4.945	6.058	49.3

Table 15: Comparison of the execution time of the MPI-only version of the Poisson multigrid solver in RAMSES to the new MPI+OpenMP implementation. Benchmarks performed using the cosmo use-case on a 1024^3 grid for 10 steps, on the MeluXina cluster. The reported numbers are the average across 3 identical runs.

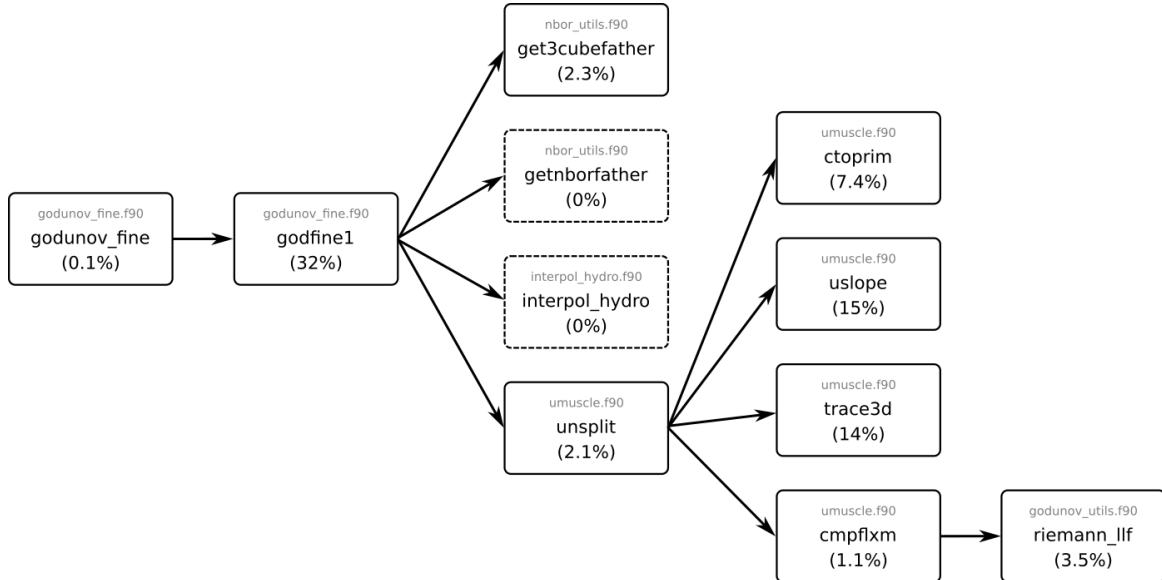


Figure 23: Call tree for the Godunov solver in RAMSES, for the Sedov use case. Between parenthesis, we note the percentage of the execution time that is spend in each subroutine (excluding calls to other subroutines), as measured by MAQAO on MeluXina. The boxes with dashed contours are only used when the AMR is activated.

SED OV entire runtime [s]				SED OV hydro-godunov [s]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
1	173.688	170.697	1.7	140.374	137.681	1.9
2	87.670	87.350	0.4	69.883	69.153	1.0
4	44.408	43.926	1.1	35.291	35.035	0.7
8	22.628	22.617	0.0	17.641	17.408	1.3
16	11.739	11.661	0.7	8.847	8.758	1.0
32	6.100	5.986	1.9	4.479	4.415	1.4
64	3.440	3.486	-1.3	2.233	2.204	1.3
SED OV ghostzones [s]				SED OV revghostzones [s]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
1	2.124	2.134	-0.5	4.784	4.470	6.6
2	1.430	1.417	0.9	2.977	3.475	-16.7
4	0.902	0.891	1.1	1.795	1.662	7.4
8	0.590	0.587	0.5	1.225	1.466	-19.7
16	0.415	0.411	0.8	0.806	0.820	-1.7
32	0.297	0.303	-2.0	0.447	0.414	7.5
64	0.305	0.287	5.9	0.313	0.332	-6.0

COSMO entire runtime [s]				COSMO poisson [s]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
1	274.022	248.847	9.2	221.204	205.756	7.0
2	139.226	126.802	8.9	113.247	105.623	6.7
4	71.775	65.524	8.7	59.480	55.352	6.9
8	38.640	35.067	9.2	32.745	30.560	6.7
16	21.519	19.672	8.6	19.134	18.010	5.9
32	13.303	12.714	4.4	12.276	12.033	2.0
64	10.668	10.104	5.3	9.931	9.524	4.1
COSMO particles [s]				COSMO rho [s]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
1	52.256	43.102	17.5	19.176	16.158	15.7
2	25.941	21.283	18.0	9.770	8.485	13.1
4	12.836	11.065	13.8	5.062	4.216	16.7
8	6.723	5.512	18.0	2.702	2.310	14.5
16	3.486	2.910	16.5	1.549	1.321	14.7
32	2.060	1.740	15.5	0.947	0.843	11.0
64	1.327	1.150	13.4	0.706	0.692	1.9

Table 16: Benchmarking results of the RAMSES code on MeluXina. Commit 1 is the commit of the public version of the code that entered the SPACE project. Commit 2 is the current state of the SPACE development branch. These benchmarks have been performed with MPI-only.

Name	Module	Coverage run_0 (%)	Max Inclusive Time Over Threads run_0 (s)
► gauss_seidel_mg_fine	ramses3d	18.11	83.83
► get3cubefather	ramses3d	13.32	62.23
► interpolate_and_correct_fine	ramses3d	11.12	51.31
► cmp_residual_mg_fine	ramses3d	8.94	41.78
► gauss_seidel_mg_coarse	ramses3d	5.08	24.32
► move1	ramses3d	4.49	21.96
► sync	ramses3d	4.22	20.78
► restrict_residual_fine_reverse	ramses3d	4.13	19.12
► cic_amr	ramses3d	3.34	16.40
► get3cubepos	ramses3d	2.72	13.52
► check_tree	ramses3d	2.69	13.57
► hilbert3d	ramses3d	2.21	10.50
► interpolate_and_correct_coarse	ramses3d	1.83	8.60
► make_virtual_fine_dp	ramses3d	1.61	8.16
► cmp_residual_mg_coarse	ramses3d	1.22	5.85
► kill_tree	ramses3d	1.22	5.83
► gradient_phi	ramses3d	1.1	5.45
► make_fine_bc_rhs	ramses3d	1.04	4.83
► cmp_residual_norm2_fine	ramses3d	1.02	5.04

Loop id	Source Location	Source Function	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
3101	ramses3d - multigrid_fine_fine.f90:261-327 [...]	gauss_seidel_mg_fine	16.16	2.54	21.82
3109	ramses3d - multigrid_fine_fine.f90:477-493	interpolate_and_correct_fine	9.31	0	19.53
3097	ramses3d - multigrid_fine_fine.f90:108-171 [...]	cmp_residual_mg_fine	7.98	1.11	22.08
646	ramses3d - nbors_utils.f90:169-174	get3cubefather	6.91	0	12.5
3102	ramses3d - multigrid_fine_fine.f90:365-396 [...]	restrict_residual_fine_reverse	3.69	0	20.9
645	ramses3d - nbors_utils.f90:180-185	get3cubefather	2.40	0	12.5
1947	ramses3d - particle_tree.f90:321-328	check_tree	1.71	19.05	24.4
3150	ramses3d - multigrid_fine_coarse.f90:579-600	interpolate_and_correct_coarse	1.46	0	19.85
626	ramses3d - nbors_utils.f90:290-296	get3cubepos	1.32	74.78	75.43
1855	ramses3d - move_fine.f90:482-485	move1	1.29	0	20.96
661	ramses3d - hilbert.f90:180-185	hilbert3d	1.22	12.5	25
1816	ramses3d - synchro_fine.f90:447-489 [...]	sync	1.07	36.07	42.62
514	ramses3d - virtual_boundaries.f90:459-460	make_virtual_fine_dp	1.01	0	25

Figure 24: MAQAO profiling results for the cosmo use-case run on half a MeluXina node. Top: an overview of the most time-consuming subroutines. Bottom: vectorization rate of the main loops.

```

        uloc(ind_nexist(i), i3, j3, k3, ivar) = u2(i, ind_son, ivar)
    end do
end do

```

This loop is memory-bound. Due to the non-contiguous indexing, the compiler does not manage to vectorize this loop. To speed up this part of the code, we need to rewrite the way the memory is accessed here.

ctoprim This routine converts the conservative variables to primitive variables. A complex operation is slowing down this subroutine. Indeed, a square-root is used in the calculation of the sound speed. Closer inspection revealed the sound-speed is not used further in the Godunov solver, but only in an alternative hydrodynamics solver which is not used for production runs. To speed up this routine, we will move the computation of the soundspeed to a dedicated subroutine, only called by the alternative solver.

uslope This subroutine lacks modularity, making it somewhat tedious to optimize. The main issue is the presence of if-else branching in the inner loops, preventing vectorization.

trace3d This is one of the computation-heavy routines. It is already fully vectorized.

riemann_llf This subroutine is not fully vectorized. The performance is limited by a square root operations.

5.5.2 Cosmo use-case

For the cosmo use-case, we also performed profiling with MAQAO of commit 9c518f8 of the dev branch on MeluXina. We use the compiler flags `-g -march=native` with `NVECTOR=16`. We run the version with 1024^3 cells for 10 time-steps on 64 cores (half a node).

In general, the vectorization rate for this subroutines probed by this benchmark is very poor. This is largely due to the many if-conditions in inner loops.

gauss_seidel_mg_fine and coarse The inner loop contains various complex conditionals. To optimize, we can attempt to streamline these. Further, expensive conversion instructions have been detected. In some if-conditions the type of the constants does not match.

get3cubefather Closer inspection of this subroutine revealed two things: 1) complex filtering is repeatedly performed, 2) while two quantities were computed, only one of them was requested per function call, 2) complex filtering is repeatedly performed. As a solution, we split the routine into two parts and removed the filtering. Originally, the cost of this routine and its subroutine `get3cubepos` was 75.75 s. Our refactoring reduced this to 23.36 s, which is a 69% gain. This resulted in an overall performance improvement of about 9% for the cosmo use-case as shown in Section 5.4.

interpolare_and_correct_fine and coarse This loop is not vectorized. An if-condition cycles the loop.

cmp_residual_mg_fine This subroutine poorly vectorized. It contains many if-else branches. Further, `maqao` detected complex instructions and expensive conversions.

restrict_residual_fine_reverse Again, this loop is poorly vectorized due to the presence of if-conditions. Additionally, `maqao` detected data structures that cannot be efficiently read/written, because of constant unknown stride and irregular or indirect access.

5.6 Conclusion

5.6.1 Sedov use-case

Optimizing the compilation parameters for the sedov use-case leads to the following improvements, as measured on 1 node.

- Using a value of `NVECTOR=8` at compile-time results in a 32% performance gain compared to the default `NVECTOR` value of 32.
- For compilation with GNU compilers on the compute nodes, adding the flags `-march=native -flto -fwhole-program` results in a 17% performance gain.

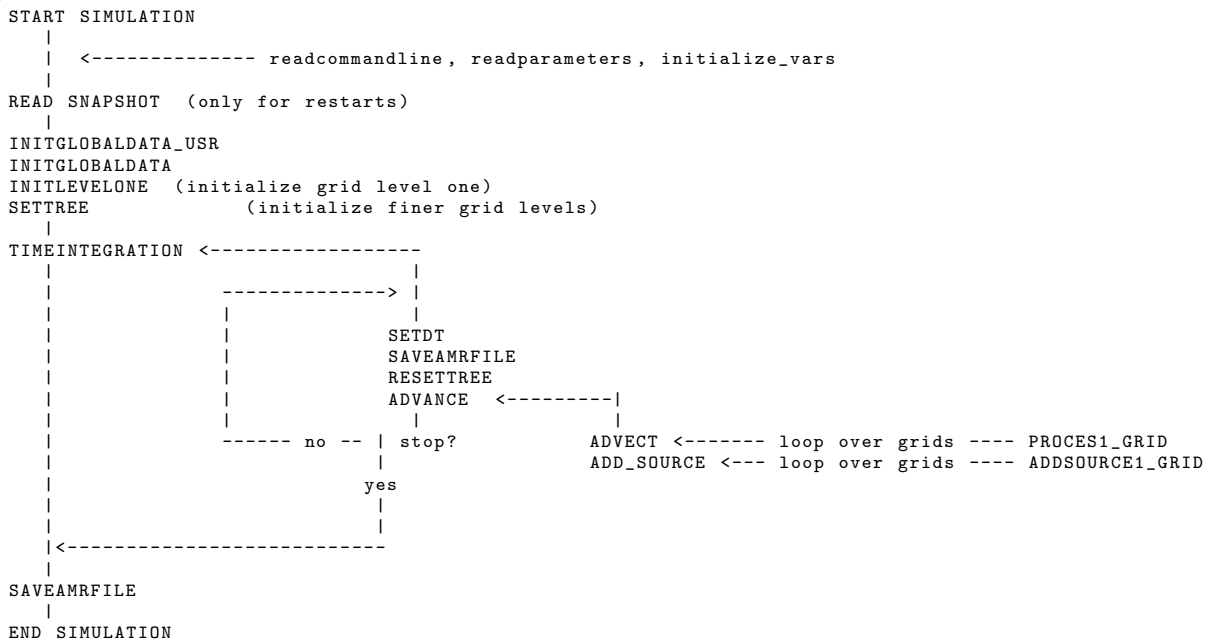
5.6.2 Cosmo use-case

The optimizations for the cosmo use-case can be summarized as follows.

- Our first version of the MPI+OpenMP implementation for the poisson multigrid solver increases the parallel efficiency from 34 to 67% for this module. This module takes 70-80% of the time for the cosmo benchmark, leading to an overall speedup on 64 nodes even though not all modules have been parallelized with OpenMP yet.
- Refactoring of the neighbor-searching routine results in a 9% performance gain.

6 BHAC

BHAC is an open access (<https://gitlab.itp.uni-frankfurt.de/BHAC-release/bhac>) multidimensional General Relativistic Magnetohydrodynamics (GRMHD) code that is mainly used to study accretion flows onto compact objects. BHAC has been designed to solve the GRMHD equations in arbitrary (stationary) space-times/coordinates and exploits adaptive mesh refinement techniques with an oct-tree block-based approach provided by the open access MPI-AMRVAC framework (<https://github.com/amrvac/amrvac>). Originally designed to study black hole (BH) accretion in ideal GRMHD, BHAC has been extended to incorporate nuclear equations of state, neutrino leakage, charged and purely geodetic test particles, and non-black hole fully numerical metrics. BHAC has been employed in a number of studies of accretion into supermassive black holes and other compact objects. In addition, BHAC's results, after a general-relativistic ray-tracing (GRRT) post-processing, can be used to compute synthetic observable images of BH shadows and the surrounding accretion flows. These calculations are performed with the GRRT Black Hole Observations in Stationary Spacetimes (BHOSS) code. The GRMHD simulation data produced by BHAC are used as an input for BHOSS to produce accretion flow and BH shadow images. A high-level description of the code and the main algorithms can be found in §4 of the D1.2 deliverable of SPACE CoE. Below a rough description of the workflow of BHAC is shown.



6.1 Use-case description

The use case selected for benchmarking BHAC's performance in the current deliverable differs from the one described in Section 4.1 of D1.1. The use case described in the latter, i.e. the magnetised torus around a spinning black hole, is currently being integrated into the GPU-enabled version of BHAC (gBHAC). Although the implementation is progressing as planned, it is not yet fully validated and stable enough for use in the context of the current deliverable. By the end of the SPACE project, this use case will be fully implemented into gBHAC, and its performance will be evaluated and reported in a future deliverable.

Instead, here, we use the use case that was employed to monitor the GPU-porting of BHAC, which is fully validated and operational. This use case describes spherical accretion onto a non-spinning black hole with a strong radial magnetic field. It serves as a robust benchmark for testing the stability and accuracy of the conservative GRMHD algorithm implemented in BHAC. The underlying analytical solution is the Michel accretion flow [16], a general relativistic extension of the classical Bondi solution [17]. The presence of a radial magnetic field of the form $B^r \propto \gamma^{-1/2}$ does not alter the steady-state flow, allowing for meaningful comparison against a known solution [18]. This use case is particularly demanding for BHAC, as it tests the robustness of the conservative-to-primitive variable inversion, which is a critical component in GRMHD simulations. The initial conditions follow the prescription of [19], and the strength of the magnetic field is controlled via the

magnetisation parameter $\sigma = b^2/\rho$ at the inner radial boundary. Analytical boundary conditions are applied at both the inner and outer edges of the domain to maintain a steady-state configuration. We employ spherical modified Kerr-Schild coordinates.

An example of the parameter file used for the aforementioned use-case, which serves as basis for the benchmarking results presented in Table 17, is provided below.

```
&filelist
  primnames      = 'rho_u1_u2_u3_p_b1_b2_b3_s_lfac_xi'
  autoconvert    = .true.
  saveprim       = .true.
  convert_type   = 'vtuBCCmpi'
  filenameini    = 'output/data'
  filenameout    = 'output/data'
  filenameelog   = 'output/amrvac'
  nwauxio        = 2
  typepario      = 0
&end

&savelist
  itsave(1,1)    = 0
  itsave(1,2)    = 0
  ditsave(1)      = 1
  dtsave(2)      = 1
&end

&stoplist
  itmax          = 100
  tmax           = 10
  dtmin          = 1.d-6
&end

&methodlist
  wnames         = 'd_s1_s2_s3_tau_b1_b2_b3_Ds_lfac_xi'
  typeadvance    = 'twostep'
  typefull1      = 13*'tvdlf'
  typelimiter1   = 13*'cada3'
  typeinversion  = '2D1DEntropy'
  typeemf        = 'uct2'
  typeaxial      = 'spherical'
&end

&boundlist
  dixB           = 2
  typeB          =
    5*'special' 'cont' 'zero' 'special' 'special' 2*'cont'
    5*'special' 'cont' 'zero' 'special' 'special' 2*'cont'
    'symm','symm','asymm','symm','symm','symm','asymm','symm', 3*'symm'
    'symm','symm','asymm','symm','symm','symm','asymm','symm', 3*'symm'
&end

&amrlist
  mxnest         = 1
  nxlone1        = 51200
  nxlone2        = 4000
  xprobmin1      = 0.587786665
  xprobmax1      = 2.302585093
  xprobmin2      = 0.0d0
  xprobmax2      = 0.5d0
  prolongprimitive = T
&end

&paramlist
  courantpar      = 0.8d0
  typecourant     = 'maxsum'
&end
```

Notice that the AMR level has been chosen to $mxnest = 1$ because we are working with uniform grids.

6.2 Benchmarking results

Benchmarking of BHAC was conducted on the Karolina cluster of IT4I. Commit 1 and 2 were benchmarked on the non-accelerated and accelerated nodes of Karolina, respectively. Commit 1 is the original CPU-only version of BHAC as it entered the SPACE CoE project in January 2023 and is available [here](#). Commit 2 is the current version of gBHAC available [here](#) (commit of 19th of April 2025).

Benchmarking was conducted with the objective of identifying the configuration that yields the shortest execution time, rather than enforcing an equal allocation of CPU and GPU resources. The chosen setups were

optimized to align with the hardware architecture of the respective compute nodes of Karolina. The 720 non-accelerated nodes of Karolina are each equipped with 2 AMD EPYC™ 7H12 processors (64 cores, 2.6 GHz), providing a total of 128 CPU cores per node. In contrast, the 72 accelerated nodes each feature 8 NVIDIA A100 GPUs and are supported by 2 AMD EPYC™ 7763 processors (64 cores, 2.45 GHz), also yielding 128 CPU cores per node alongside the 8 GPUs.

# of nodes	Entire code runtime [s]		
	commit 1	commit 2	gain [%]
1	3846.348	790.089	79.5
2	2043.957	411.199	79.9
4	1064.803	199.581	81.3
8	538.107	100.956	81.2
16	287.924	52.598	81.7
32	148.598	28.328	80.9

# of nodes	Timeloop runtime [s]			Riemann solver runtime [s]		
	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
1	3767.525	589.978	84.3	3596.295	328.809	90.1
2	2004.425	307.869	84.6	1913.941	164.912	91.4
4	1044.997	150.075	85.6	993.991	81.866	91.8
8	528.115	75.069	85.8	506.070	39.461	92.2
16	282.847	39.643	86.0	254.751	20.252	92.1
32	144.450	21.432	85.2	128.962	11.020	91.5

Table 17: Benchmarking results for the BHAC code.

The test case parameters were configured to evolve the system for 100 time steps, providing a reasonable and sufficient execution time for the benchmarking objectives of the current deliverable. The computational domain is discretized using a grid of size $(r, \theta) = (51200, 4000)$, resulting in approximately 2×10^8 total grid points. The domain is further divided into coordinate blocks of size $(r, \theta) = (400, 250)$ corresponding to 10^5 grid points per block for the Commit 1 runs and $(r, \theta) = (800, 500)$, or 4×10^5 grid points per block, for the Commit 2 runs. This yields a total of 2048 and 512 blocks across the grid for Commit 1 and Commit 2, respectively. To focus exclusively on computational performance, all input/output (I/O) operations were disabled during benchmarking.

The decision to use a different number of blocks for each commit, while maintaining the same total number of grid points, is motivated by the different hardware architectures of the nodes utilized in each case. Additionally, this approach aligns with BHAC’s design, which permits configurations where the total number of MPI processes is less than or equal to the number of coordinate blocks. This flexibility enables optimal mapping of computational workload to the available resources. Performance testing on the non-accelerated nodes revealed that the most efficient configuration, in terms of minimizing execution time, was achieved by utilizing all 128 available CPU cores per node. This was accomplished by assigning 32 MPI tasks per node, each employing 4 CPU cores. In this setup, the number of MPI processes corresponds directly to the number of tasks per node, ensuring full exploitation of the node’s computational capacity. On the accelerated nodes, the shortest execution times were achieved by assigning 8 MPI tasks per node, each mapped to one of the 8 available GPUs. Each MPI task was configured to utilize 8 CPU cores, resulting in the use of 64 out of the 128 available CPU cores per node. This configuration balances between GPU utilization and CPU resource allocation, aligning well with the architecture of the accelerated nodes.

The benchmarking results are presented in Table 17, where the performance of the two BHAC versions is compared. The table includes runtime measurements for the entire code, the timeloop, and the Riemann solver across different number of nodes. The GPU-enabled Commit 2 demonstrates substantial performance gains across all measured components, with particularly high improvements in the Riemann solver and timeloop execution. These results validate the effectiveness of the GPU port in accelerating key computational components of the BHAC code, while also showing strong parallel scalability. The comparatively lower speedups observed in the total code and timeloop runtimes, relative to the Riemann solver, can be attributed to the presence of modules that have not yet been fully ported to GPU. Specifically, modules responsible for generating the initial data and imposing boundary conditions currently execute only partly on GPU, limiting the overall acceleration

achieved in these broader runtime metrics.

The parameter files, slurm submission scripts and benchmarking results can be found in the SPACE CoE Internal Data Archive hosted by IT4I at Karolina:

[/mnt/proj3/open-30-28/INTERNAL_DATA_ARCHIVE/BHAC/scientific_cases/magnetised_sph_accretion/](/mnt/proj3/open-30-28/INTERNAL_DATA_ARCHIVE/BHAC/scientific_cases/magnetised_sph_accretion/)

6.3 Profiling activities

This analysis examines the GPU implementation of the Black Hole Accretion Code (BHAC) - an MHD code for simulating astrophysical plasma flows around black holes. BHAC was ported to GPUs using OpenACC directives, maintaining a single source code base while enabling execution on both CPU and GPU architectures.

Performance data was collected using NVIDIA Nsight Systems 2024.1.1.59 on the Karolina supercomputer, capturing GPU workloads, CPU activity, and runtime API calls during a representative 4-rank production run. The profiling data was exported to SQLite format for detailed analysis of kernel execution statistics, memory patterns, MPI communication, and synchronization bottlenecks.

Component	Specification
Processors	2x AMD EPYC 7H12, 64-core, 2.6 GHz per node
Memory	256 GB DDR4 3200MT/s per node
Interconnect	HDR InfiniBand (200 Gb/s)
GPU	NVIDIA A100 40GB SXM4
Software	Version
Nsight Systems	2024.1.1.59
CUDA	11.7
OpenACC	22.3
MPI Implementation	OpenMPI 4.1.4

Table 18: System configuration used for performance analysis on the Karolina supercomputer

Our analysis of a 4-rank BHAC execution reveals five critical performance bottlenecks:

1. **Excessive synchronization:** `cuStreamSynchronize` calls consume 74.93% (35.91s) of CUDA runtime API time, with over 1.2 million calls forcing sequential GPU operation rather than leveraging parallel architecture.

2. **Memory management issues:** All ranks show 3,327 memory allocations without corresponding deallocations, indicating a systematic memory leak totaling approximately 3.25 GB per rank.

3. **No computation-transfer overlap:** Complete absence of overlap between kernel executions and memory operations (0.00%) prevents hiding data transfer latency, despite substantial transfer volume (23.2 GB for rank 0) and numerous memory operations (>380,000 per rank).

4. **Register pressure:** About 10.3% of kernels are occupancy-limited due to high register usage. The most time-consuming kernel (`getaux_acc_395_gpu_1`) achieves only 44% occupancy due to using 66 registers per thread.

5. **Centralized MPI topology:** Rank 0 acts as a communication hub, processing 26,780 MPI_Isend operations (24.87s, 78.5% of MPI time) with 99.7% being large messages (64KB-1MB).

6.3.1 Performance Metrics Overview

The following table summarizes the key performance metrics across all four analyzed ranks:

¹Kernel Time Efficiency: Percentage of total execution time spent in GPU kernels (Kernel Time / Total Duration). ²GPU Compute Utilization: Includes additional GPU activity beyond kernel execution. ³Theoretical Kernel Efficiency: Average efficiency reported by NVIDIA profiler for individual kernels under ideal conditions. ⁴Memory Utilization: Peak GPU Memory (3,327 MB) divided by total available A100 GPU memory (40 GB).

These metrics reveal critical inefficiencies in the OpenACC implementation. The stark contrast between theoretical kernel efficiency (50.3%) and actual kernel time efficiency (14.8%) highlights substantial overhead from the OpenACC runtime environment. Synchronization time (35.9s) exceeds actual kernel execution time (25.8s), showing how OpenACC prioritizes correctness over performance.

Metric	Rank 0	Rank 1	Rank 2	Rank 3	Average
Total Duration (s)	174.3	174.3	175.9	174.3	174.7
Kernel Time (s)	25.8	25.7	25.8	25.8	25.8
Kernel Time Efficiency (%) ¹	14.8	14.7	14.7	14.8	14.8
GPU Compute Utilization (%) ²	15.2	15.1	15.1	15.2	15.2
Theoretical Kernel Efficiency (%) ³	50.3	50.3	50.3	50.3	50.3
Synchronization Time (s)	35.9	35.6	36.0	36.1	35.9
Data Transferred (GB)	22.7	22.7	22.7	22.7	22.7
Download Bandwidth (GB/s)	55.1	50.9	47.5	54.8	52.1
Upload Bandwidth (GB/s)	30.3	31.1	31.0	32.3	31.2
Peak GPU Memory (MB)	3327	3327	3327	3327	3327
Memory Utilization (%) ⁴	8.3	8.3	8.3	8.3	8.3
Kernel Launches	1,088,544	1,088,544	1,088,544	1,088,544	1,088,544
Memory Operations	380,312	380,312	380,312	380,248	380,296
MPI Calls	54,602	54,085	54,085	53,925	54,174

Table 19: Performance metrics summary across all ranks

Bandwidth metrics show asymmetry between download (52.1 GB/s) and upload (31.2 GB/s) operations, both falling short of the A100’s theoretical peak (1.6 TB/s). The nearly identical execution times across ranks indicate good synchronization but suggest potential wait states. The extremely high counts of kernel launches (> 1 million per rank) and memory operations ($\sim 380,300$ per rank) point to fine-grained execution that could benefit from kernel fusion and consolidated transfers. Overall, the application is primarily bound by synchronization overhead and data movement inefficiencies rather than computational throughput.

6.3.2 Top Time-Consuming NVTX Ranges

The NVTX ranges provide insight into the high-level structure of the application and identify the most time-consuming computational phases:

Range Name	Total Time (s)	% of Total Runtime
getBC	78.4	26.8%
step	39.5	13.5%
iteration	30.9	10.6%
call_tvdlf1	21.9	7.5%
call_tvdlf2	21.7	7.4%
getAux	15.7	5.4%
kernel_2	12.8	4.4%
timeintegration	10.4	3.6%

Table 20: Top time-consuming NVTX ranges

Parent Range	Self Time (s)	Children Time (s)	Total Time (s)
timeintegration	0.0	31.7	31.7
iteration	0.0	44.0	44.0
step	5.8	33.7	39.5
call_tvdlf2	0.0	21.7	21.7
mini-advance	2.0	4.4	6.4
pre-iteration	0.8	0.0	0.8
GPU-izable	1.9	0.4	2.3

Table 21: Key parent-child relationships in the NVTX hierarchy (Total Time = Self Time + Children Time)

NVTX ranges analysis reveals critical insights about the application’s execution structure through the relationship between self time, children time, and total time. Table ?? shows the top time-consuming functions, while Table 21 breaks down key parent-child time distributions.

The data reveals four distinct function patterns:

1. **Pure Coordinator Functions** (e.g., `timeintegration`, `iteration`): Zero self time, with total time (31.7s, 44.0s) coming entirely from children.
2. **Hybrid Functions** (e.g., `step`): Mix of self time (5.8s) and children time (33.7s).
3. **Direct Computation Functions** (e.g., `pre-iteration`): All work performed directly (0.8s self time, 0.0s children time).
4. **Balanced Functions** (e.g., `mini-advance`): More even distribution (2.0s self time, 4.4s children time).

The discrepancy between `iteration`’s total time (44.0s) and its reported time (30.9s) indicates asynchronous execution with concurrent child operations. The `getBC` function remains the most significant bottleneck (78.4s, 26.8% of total time) and represents a primary optimization target.

This complex interplay between self and children time reveals a concurrent execution model that is not being efficiently utilized, explaining the gap between theoretical (50.3%) and actual (14.8%) kernel efficiency.

6.3.3 Compute Performance Analysis

This section analyzes the computational performance of the BHAC GPU implementation, focusing on CUDA Runtime API usage patterns, kernel execution characteristics, and compute-related bottlenecks.

The CUDA Runtime API usage profile reveals significant insights into the application’s GPU utilization patterns. Figure 25 shows the distribution of time spent in various CUDA Runtime API calls for Rank 0.

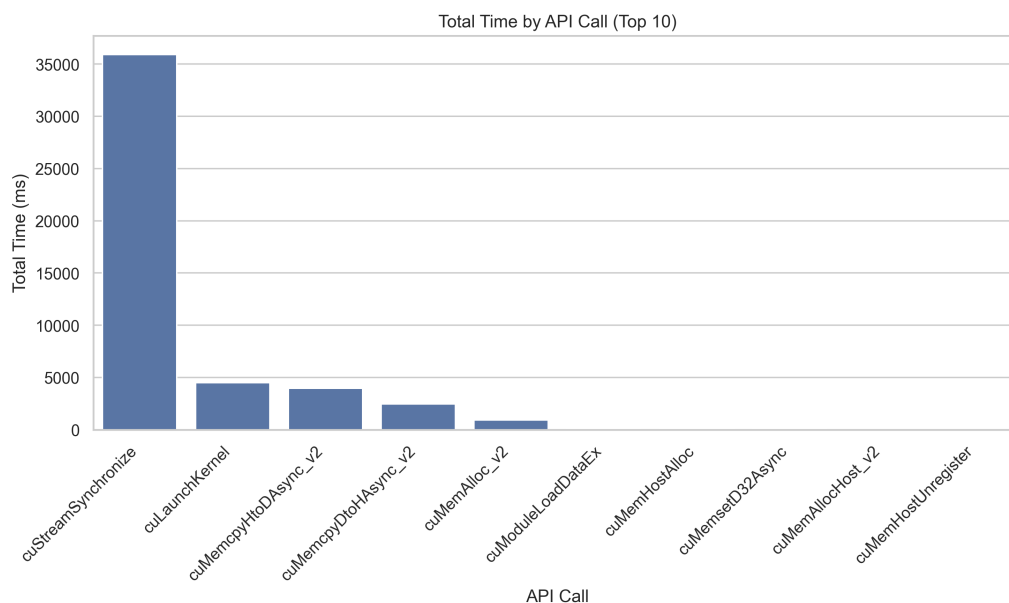


Figure 25: Distribution of time spent in CUDA Runtime API calls (Rank 0)

Analysis of CUDA Runtime API calls reveals how OpenACC directives translate to CUDA operations, highlighting three critical inefficiencies:

1. **Excessive synchronization:** `cuStreamSynchronize` dominates the runtime profile, consuming 74.93% (35.91s) of total runtime API time. This results from OpenACC’s default behavior of synchronizing after each accelerated region, with over 1.2 million synchronization calls effectively serializing GPU operations.
2. **Inefficient memory transfers:** Host-device transfers (`cuMemcpyHtoDAsync_v2` and `cuMemcpyDtoHAsync_v2`) account for 13.39% (6.41s) of runtime API time. Despite the "Async" suffix, profiling shows no effective overlap between these operations and kernel execution, indicating OpenACC is not utilizing asynchronous queues effectively.
3. **Kernel launch overhead:** `cuLaunchKernel` consumes 9.37% (4.49s) of runtime API time, with 1,088,543 launches reflecting OpenACC’s fine-grained parallelization approach. This results in the GPU spending only 25.8s (14.8% of total runtime) actually executing kernels despite the application running for 174.3s.

API Name	Total Time	% of Total	Calls	Avg Time
cuStreamSynchronize	35.91 s	74.93%	1,223,564	29.35 μ s
cuLaunchKernel	4.49 s	9.37%	1,088,544	4.12 μ s
cuMemcpyHtoDAsync_v2	3.95 s	8.25%	344,408	11.48 μ s
cuMemcpyDtoHAsync_v2	2.46 s	5.14%	35,904	68.55 μ s
cuMemAlloc_v2	916.09 ms	1.91%	3,326	275.43 μ s

Table 22: Top time-consuming CUDA Runtime API calls

The kernel execution analysis reveals detailed insights into the computational efficiency of the GPU implementation. Figure 26 shows the top kernels by total execution time for Rank 0.

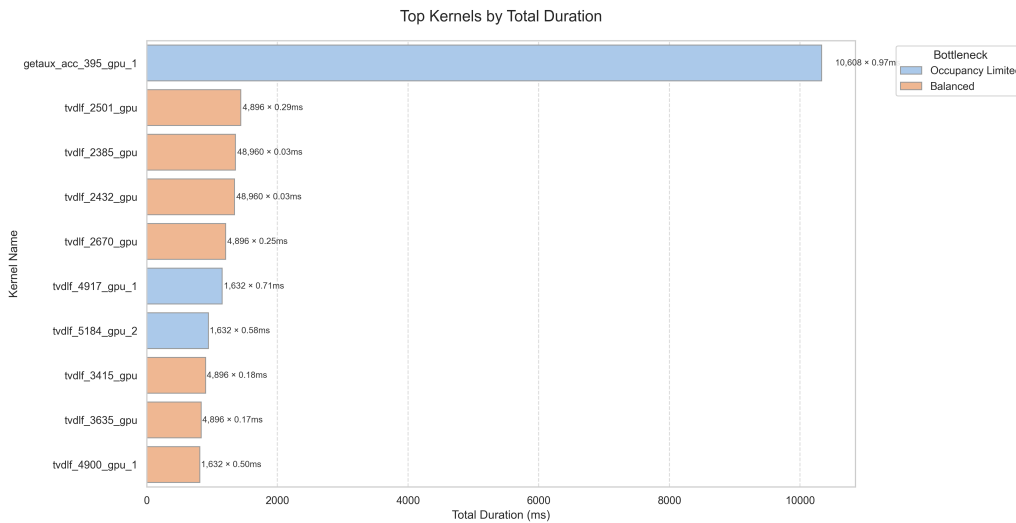


Figure 26: Top kernels by total execution time (Rank 0)

The kernel execution analysis reveals three key characteristics of the BHAC GPU implementation:

- High kernel diversity:** The application uses 184 unique kernels across over 1 million executions, reflecting BHAC's complex computational structure but introducing significant launch overhead (4.49s in `cuLaunchKernel` calls).
- Skewed execution distribution:** The top 10 kernels account for approximately 75% of GPU execution time, with `getaux_acc_395_gpu_1` being particularly dominant (40% of total kernel time). This kernel operates at only 44% occupancy due to high register usage, making it a prime optimization target.
- Short-duration kernels:** The average kernel execution time is just 23.7 μ s, limiting the GPU's ability to reach peak performance as kernels terminate before resources fully ramp up. These brief kernels could benefit from fusion or batching techniques to reduce launch overhead.

Kernel Name	Executions	Total Time	Avg Time	Efficiency	Bottleneck
getaux_acc_395_gpu_1	10,608	10.3 s	973.8 μ s	50.2%	Occupancy Limited
tvdlf_2501_gpu	4,896	1.4 s	294.2 μ s	50.3%	Balanced
tvdlf_2385_gpu	48,960	1.4 s	27.7 μ s	50.4%	Balanced
tvdlf_2432_gpu	48,960	1.3 s	27.4 μ s	50.4%	Balanced
tvdlf_2670_gpu	4,896	1.2 s	246.4 μ s	50.3%	Balanced

Table 23: Top time-consuming kernels

The kernel occupancy analysis provides insights into the GPU resource utilization efficiency. Figure 27 shows the distribution of theoretical occupancy across kernels for Rank 0.

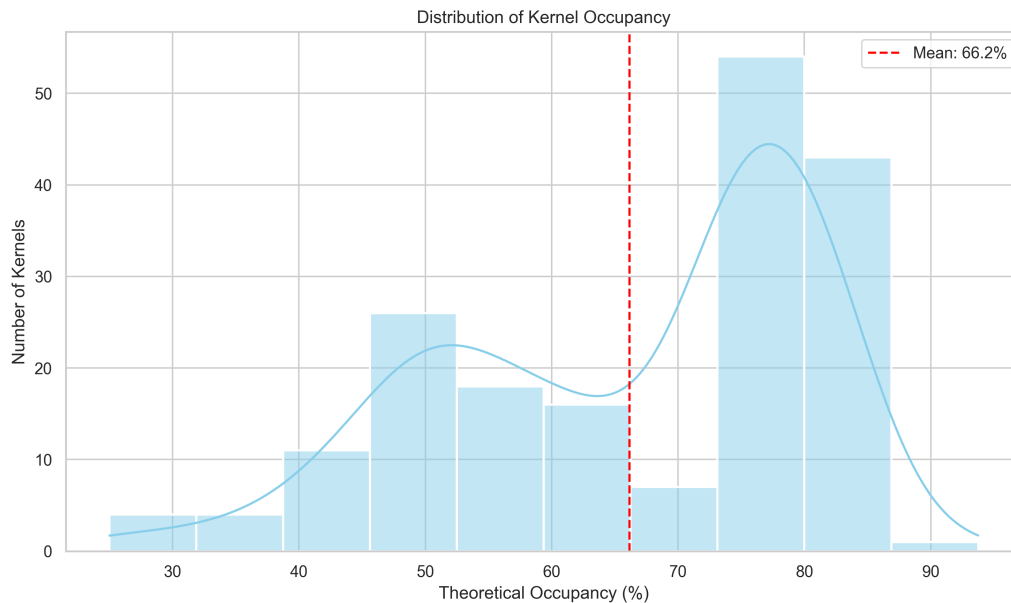


Figure 27: Kernel occupancy histogram (Rank 0)

The kernel occupancy and efficiency analysis reveals that while only 10.3% of kernels (19 out of 184) are occupancy-limited due to excessive register usage, these include critical performance-impacting functions.

The most time-consuming kernel (`getaux_acc_395_gpu_1`, 40% of execution time) operates at only 44% theoretical occupancy due to using 66 registers per thread, severely limiting the GPU's ability to hide memory latency. Similarly, `tvdlf_5184_gpu_2` uses 96 registers per thread, limiting occupancy to 31%. This pattern likely stems from OpenACC's automatic code generation approach, which doesn't optimize register usage as effectively as hand-tuned CUDA code.

Most kernels achieve only 50% efficiency, suggesting suboptimal resource utilization due to potential uncoalesced memory access patterns, thread divergence, or insufficient instruction-level parallelism. While the bottleneck analysis (Figure 46) shows most kernels (89.7%) as "Balanced," a significant portion of execution time is spent in occupancy-limited kernels, making register optimization a high-value target.

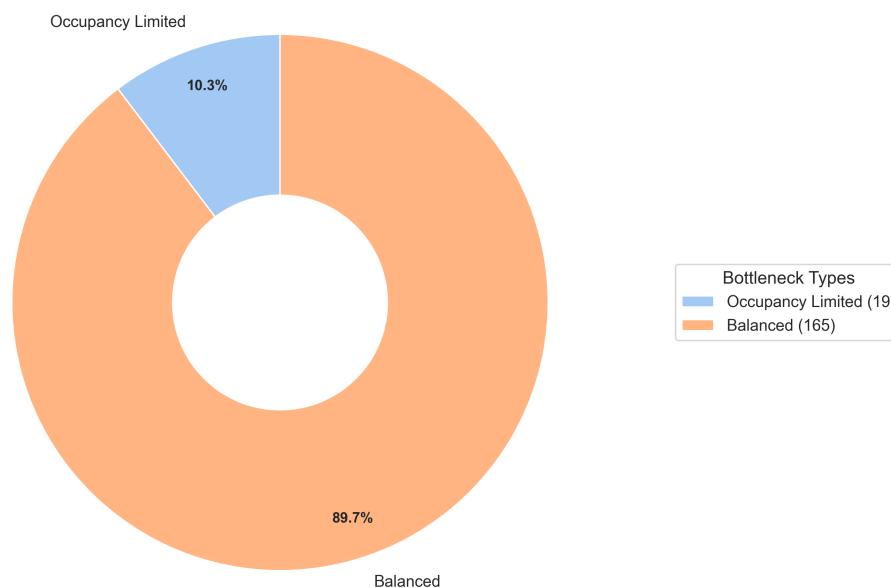


Figure 28: Kernel bottleneck distribution (Rank 0)

The bottleneck analysis shows that while most kernels (89.7%) are classified as "Balanced" in terms of performance limitations, a significant portion of the execution time is spent in occupancy-limited kernels. This suggests that register optimization could yield substantial performance improvements for the application.

6.3.4 Memory and Communication Analysis

This section examines the memory management patterns and MPI communication characteristics of the BHAC GPU implementation, identifying potential bottlenecks and optimization opportunities.

The memory management analysis reveals critical insights into how the application allocates and utilizes GPU memory. Figure 29 shows the memory usage pattern across the execution.

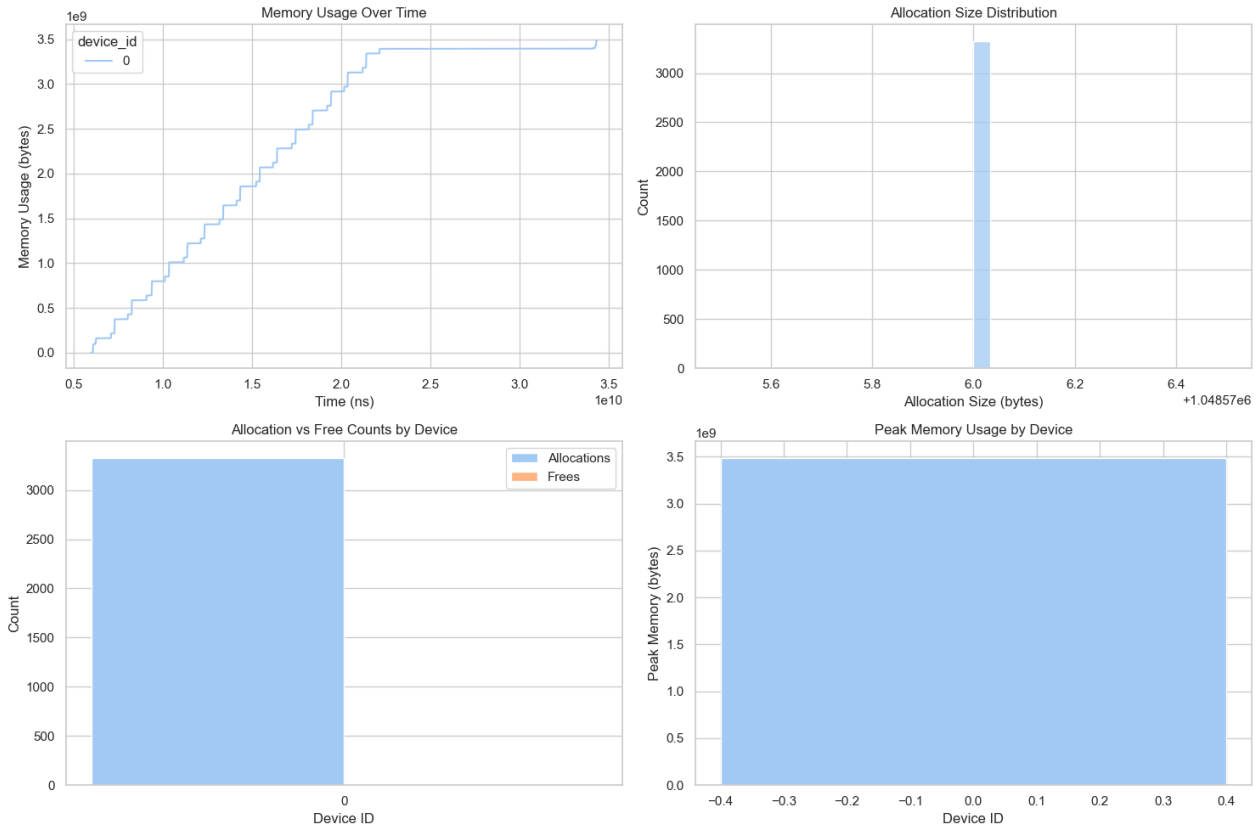


Figure 29: GPU memory usage over time

The memory management analysis reveals a critical concern: 3,327 memory allocations with no corresponding deallocations across all ranks, indicating a systematic memory leak. These allocations (primarily via `cuMemAlloc_v2`) consume approximately 3.25 GB of GPU memory per rank.

While this represents only about 8% of the available memory on A100 GPUs, the complete absence of deallocation calls means memory usage would grow unbounded during extended simulations, eventually causing out-of-memory failures. This is particularly concerning for production runs that may continue for days or weeks.

The uniform allocation size pattern (averaging 1 MB) suggests these are similarly structured data objects, making them ideal candidates for a memory pooling strategy where pre-allocated blocks could be reused rather than repeatedly allocated and never freed.

API	Allocations	Frees	Total Size
<code>cuMemAlloc_v2</code>	3,326	0	3.25 GB
<code>cuMemAllocHost_v2</code>	1	0	1.00 MB

Table 24: Memory API usage summary

Data transfer analysis reveals three critical inefficiencies:

1. **High transfer volume:** Each rank transfers approximately 22.7 GB between host and device (90.7 GB total across four ranks), indicating excessive data movement rather than maintaining GPU data residency. While the asymmetric pattern (86% host-to-device) suggests favorable GPU computation focus, the volume indicates data that could potentially remain resident on the GPU is being repeatedly transferred.
2. **Excessive transfer frequency:** Each rank performs 344,408 host-to-device transfers and 35,904 device-to-host transfers, with small average sizes (68 KB and 650 KB respectively). This pattern incurs substantial overhead from transfer initiation latency and could benefit significantly from batching and consolidation.
3. **No computation-transfer overlap:** The analysis detected 0.00% overlap between memory operations and kernel executions, indicating strictly sequential execution. This prevents hiding data transfer latency behind computation, forcing the GPU to remain idle during transfers and the CPU to wait during kernel executions.

6.3.5 MPI Communication Analysis

The MPI communication analysis provides insights into the inter-rank communication patterns and potential bottlenecks. Figure 30 shows the communication matrix between ranks.

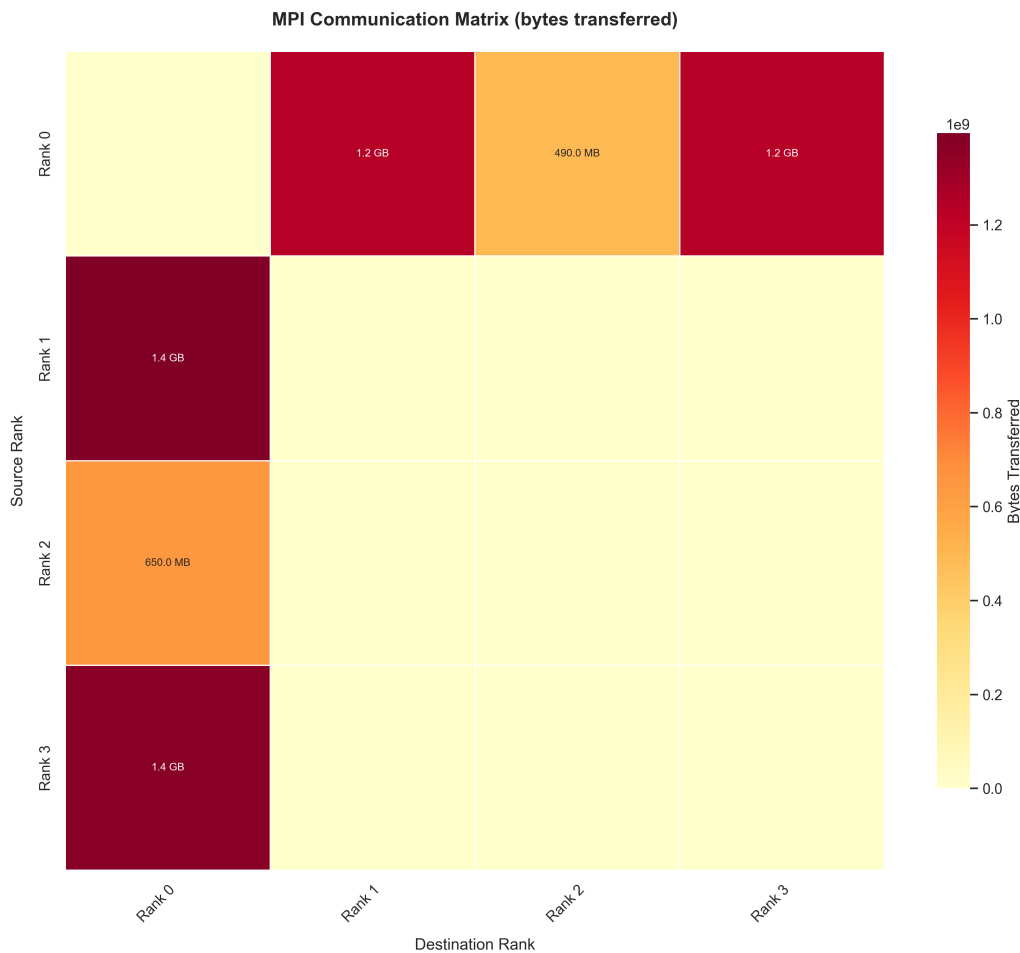


Figure 30: MPI communication matrix showing data exchange between ranks

The MPI communication analysis reveals a centralized pattern that limits scalability. Key findings include:

1. **Substantial communication volume:** Each rank exchanges approximately 6.4 GB of data through MPI calls, indicating significant inter-rank dependencies.
2. **Centralized communication topology:** Rank 0 acts as a communication hub in a star topology, with all other ranks primarily communicating with it rather than directly with each other. This master-worker pattern creates a scalability bottleneck as Rank 0 must process communications from all other ranks, potentially leading to serialization.

3. **Efficient message sizing but poor distribution:** While 99.7% of MPI messages are efficiently sized (64KB-1MB), their concentration through Rank 0 creates processing bottlenecks.

4. **Ineffective asynchronous communication:** `MPI_Isend` operations consume 78.5% (24.87s) of total MPI time across 26,780 operations. Despite being non-blocking calls, the profiling data suggests minimal effective overlap with computation. Additionally, `MPI_Allreduce` operations (just 119 calls) account for 20.8% of MPI time, indicating significant synchronization barriers.

Call Type	Count	Total Data	Avg Size	Total Time
<code>MPI_Isend</code>	26,780	3.0 GB	110.5 KB	24,873.5 s
<code>MPI_Allreduce</code>	119	3.5 KB	29.6 B	6,598.3 s
<code>MPI_Recv</code>	848	462.4 MB	545.3 KB	96.1 s
<code>MPI_Irecv</code>	26,780	3.0 GB	110.5 KB	11.5 s

Table 25: MPI call statistics

The analysis identified key communication hotspots with corresponding recommendations:

Severity	Issue	Call Type	Recommendation
High	Centralized communication through rank 0	P2P	Direct peer-to-peer communication
High	Potential bottleneck at rank 0	P2P	Distribute coordination responsibilities
Medium	Many small messages in <code>MPI_Allreduce</code>	Collective	Message aggregation or buffering

Table 26: Communication hotspots and recommendations

6.3.6 Performance Bottlenecks and Recommendations

This section synthesizes the findings from our performance analysis to identify the primary bottlenecks in the BHAC GPU implementation and provides specific, actionable recommendations for optimization.

Our analysis reveals five critical performance bottlenecks in the BHAC GPU implementation:

1. **Excessive synchronization overhead:** 74.93% (35.91s) of CUDA runtime spent in `cuStreamSynchronize` calls. Over 1.2 million synchronization operations force sequential execution, preventing effective parallelism.
2. **Memory management issues:** 3,327 memory allocations with zero corresponding deallocations, indicating a systematic memory leak totaling 3.25 GB per rank. This threatens stability in long-running simulations.
3. **No computation-transfer overlap:** Complete absence (0.00%) of overlap between kernel executions and memory operations, preventing latency hiding despite substantial data transfer volume (23.2 GB for rank 0).
4. **Limited kernel occupancy:** 10.3% of kernels are occupancy-limited due to high register usage. The most time-consuming kernel (`getaux_acc_395_gpu_1`) achieves only 44% occupancy with 66 registers per thread.
5. **Centralized MPI communication:** Rank 0 acts as a communication hub, creating a scalability bottleneck by processing 26,780 `MPI_Isend` operations (24.87s, 78.5% of MPI time) and handling 3.0 GB of communication data.

Based on our analysis, we recommend a comprehensive optimization strategy addressing the identified bottlenecks with projected performance impacts.

The most impactful optimization would be reducing synchronization overhead by implementing explicit asynchronous execution using CUDA streams with `async` clauses in OpenACC directives. With `cuStreamSynchronize` consuming 74.93% (35.91s) of CUDA runtime, addressing this bottleneck could yield a 3-4x improvement in overall application performance. This estimate assumes that 60-70% of current synchronization overhead could be eliminated by restructuring the code to use multiple CUDA streams and event-based synchronization only when algorithmically necessary.

Addressing memory management issues would primarily prevent application instability in long-running simulations. Implementing proper deallocation for all GPU memory allocations and adopting a memory pooling strategy for the uniform-sized allocations (avg. 1 MB) would eliminate the memory leak risk. While this may not provide immediate performance benefits, we estimate a 5-15% improvement in overall performance due to reduced memory management overhead.

Enabling computation-transfer overlap would significantly improve throughput by restructuring the application using pipeline stages with double or triple buffering techniques. The current complete absence of overlap (0.00%) between memory operations and kernel executions represents a substantial opportunity. Implementing true asynchronous communication would allow computation to proceed while data transfers are in progress, improving overall application performance by 10-20%.

Optimizing kernel efficiency and occupancy presents another significant opportunity. Focusing on the 19 occupancy-limited kernels (10.3% of all kernels), particularly `getaux_acc_395_gpu_1` which accounts for 40% of kernel execution time but operates at only 44% occupancy, could yield a 10-30% improvement in computational performance. By reducing register pressure and applying loop tiling techniques, we estimate its execution time could be reduced by 20-30%. Additionally, addressing the moderate efficiency level (50%) across most kernels by optimizing memory access patterns for coalescing, reducing thread divergence, and increasing instruction-level parallelism would further improve performance.

Finally, distributing the communication load by replacing the centralized star topology with a nearest-neighbor or hierarchical communication pattern would improve scalability. With MPI operations accounting for a significant portion of execution time (31.7s for rank 0) and 78.5% of that time spent in `MPI_Isend` operations, redistributing the communication load from the centralized pattern centered on rank 0 could significantly improve scalability for larger simulation runs.

Optimization	Implementation	Difficulty	Expected Impact
Synchronization Reduction	Replace explicit synchronization with asynchronous operations and CUDA streams	Medium	High (potential 3-4x speedup)
Memory Management	Implement memory pooling and proper deallocation	Medium	Medium-High (prevents crashes in long runs)
Kernel Optimization	Register usage reduction and loop tiling	High	Medium (10-30% kernel speedup)
Communication Overlap	Non-blocking MPI operations with computation	Medium	Medium (10-20% overall speedup)
Data Transfer Optimization	Reduce host-device transfers, use pinned memory	Low	Low-Medium (5-10% speedup)

Table 27: Optimization recommendations and expected impact

6.4 Conclusion

The benchmarking campaign, presented in [Table 17](#), demonstrates substantial performance improvements in the GPU-enabled version of BHAC (Commit 2) compared to the original CPU-only version (Commit 1). Runtime measurements across various node counts show consistently lower execution times for the entire code, timeloop, and Riemann solver in Commit 2, with particularly high speedups observed in the GPU-accelerated Riemann solver. These results confirm the effectiveness of the GPU port in accelerating computationally intensive components of BHAC and showcase strong scalability across multiple nodes.

Complementing these results, in-depth profiling identified five key performance bottlenecks in the current GPU implementation. The most critical issue is excessive synchronization overhead, with nearly 75% of CUDA runtime spent in `cuStreamSynchronize` calls, severely limiting parallelism. Additional bottlenecks include persistent GPU memory leaks due to unbalanced allocation/deallocation patterns, absence of computation-transfer overlap, limited GPU kernel occupancy—particularly in the dominant kernel `getaux_acc_395_gpu_1`—and a centralized MPI communication pattern that constrains scalability. To address these issues, targeted optimizations are proposed, including asynchronous execution with multiple CUDA streams, memory pooling strategies, pipelined data transfers with double buffering, kernel tuning to reduce register pressure, and a decentralized MPI communication scheme. Collectively, these improvements have the potential to yield substantial performance

gains—up to 3–4 overall speedup—while also ensuring the stability and scalability of BHAC for large-scale GPU-accelerated simulations.

7 GRACE/FIL

The Frankfurt Illinois GRMHD code (FIL) has been the flagship code of the Relativistic Astrophysics group at Goethe University (GUF) since its development in 2018 [20]. FIL was designed to simulate astrophysical compact object mergers such as binary neutron star (BNS), black hole neutron star (BHNS), and binary black hole (BBH) mergers. FIL is heavily reliant on the Einstein Toolkit (ET) [21]—an open source framework designed to facilitate numerical relativity simulations—which provides much of the computational infrastructure that FIL requires, such as the AMR and parallelism. There is a delay to the GPU enabled ET code the current timeline is incompatible with that of the SPACE project. For this reason, an internal code, GRACE, was developed from the ground up, providing much of the same computational infrastructure that ET does, but is written for GPUs and unlike the ET is an EU code. GRACE uses the Kokkos [22] library, which makes GRACE highly portable, allowing it to be run on CPUs and GPUs, with support for both AMD and NVIDIA GPUs. GRACE is a European codebase and has been developed in-house; thus the team at GUF have control over the development timeline. GRACE uses the p4est [23] library which provides an Octree-based AMR scheme for computational efficiency, and kokkos-tools which supply built-in profiling functionality.

At the October review meeting in 2024 it was decided that GRACE would be introduced into the SPACE project, with the FIL physics modules being ported to this new GPU compatible computational framework. It is important to note that GRACE targets the exact same astrophysical scenarios as FIL i.e. BNS, BHNS and BBH collisions. All numerical relativity codes that are intended to perform simulations of compact object collisions, full GRMHD simulations, go through stages of development. The initial development will aim to perform simple relativistic hydrodynamic (HD) simulation such as a shocktube. This is the stage of development GRACE entered into the SPACE project. The next stage is to model a static stationary star—otherwise known as a Tolman–Oppenheimer–Volkoff (TOV) star—which is a GRHD simulation on a static space-time; this stage of development was completed early this year. After this, work will focus on implementing an evolving space-time—necessary for compact object collisions—and magnetic fields. Once this stage is complete, GRACE will be able to perform BNS and BHNS simulations. Currently, work is focused on implementing a space-time evolution kernel into GRACE.

7.1 Benchmarking results

For the benchmarking tests, we wanted to compare the original version of FIL that entered into the project with GRACE, when they both simulate a TOV star with the same number of cells to evolve in a uni-grid configuration on IT4I's Karolina supercomputer, where the FIL profiling took place. However, we have been running into unexpected bugs when running GRACE on Karolina and have been delayed from producing the benchmarking results. GRACE has only recently been added to the project and is still being installed on all the HPC machines. GRACE was originally developed on the GUF cluster, which has AMD accelerators, and it has been successfully deployed on LUMI and the IT4I complementary systems (CS) AMD GPU nodes. Kokkos allows for the code to be deployed on multiple different backends without major changes to the code structure; however, it should be expected that there may be complications when trying to run on a new backend. GRACE has been compiled and run on a CUDA backend at the HAWK supercomputer in HLRS center in Stuttgart, so it was expected that running on Karolina, which also has a CUDA backend for its accelerators, should not pose any problems. The exact same branch that successfully compiled and ran on HAWK was used, the code compiles and runs on Karolina; however, the AMR mesh grid introduces a bug that causes the code to crash at runtime. A patch is currently being worked on, and the benchmarking results should be produced in time for the review meeting later this year.

An initial benchmarking has been performed for Deliverable 2.3, comparing the performance of FIL and GRACE on a single CPU and GPU core respectively shown below in Figure 31:

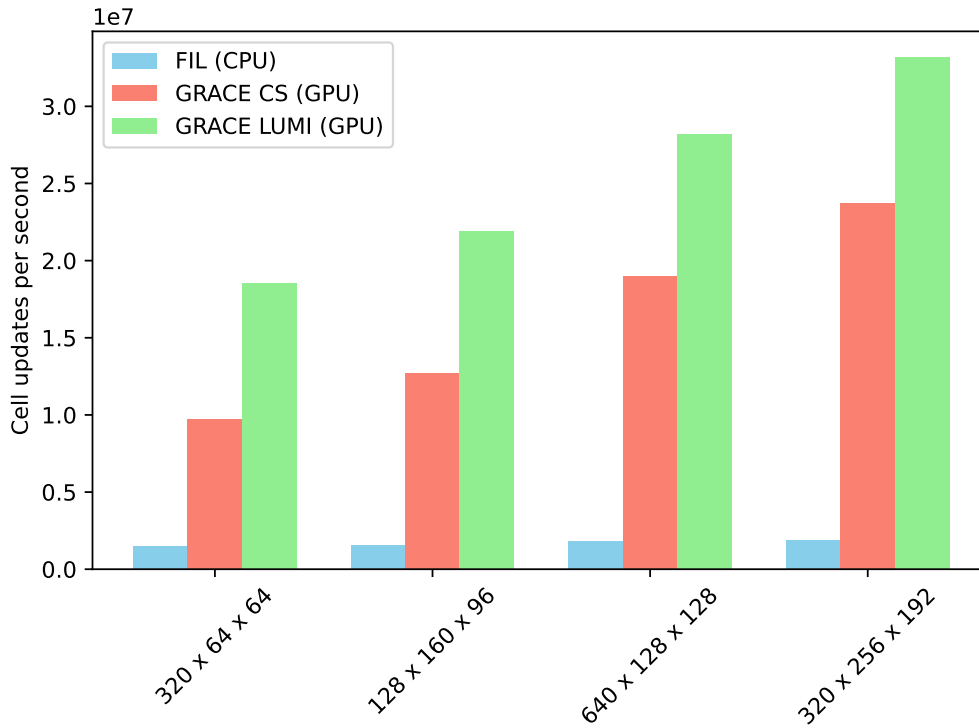


Figure 31: Comparison of cell updates per second during evolution of GRACE and FIL on the IT4I p03 partition and the LUMI-G partition.

Figure 31 shows a direct comparison between FIL and GRACE when running a simple shock tube test on a uni-grid configuration. The comparison is made between a single CPU processor on the IT4I CS p03 partition, a single M100 AMD GPU on the same node and a single M250 GPU on LUMI-G partition. As the memory on a single GPU is filled the GPU performance increases, at a resolution of 320x256x192 the M100's memory capacity was reached. On the M250 though GRACE was able to perform a simulation with resolution 720x128x128 which resulted in a cell updates per second of 4.47×10^7 which is about 25 times speed up when compared to the equivalent FIL performance.

For a more detailed breakdown of the kernels targeted for optimization –as outlined in Deliverable 2.1– see Figure 32

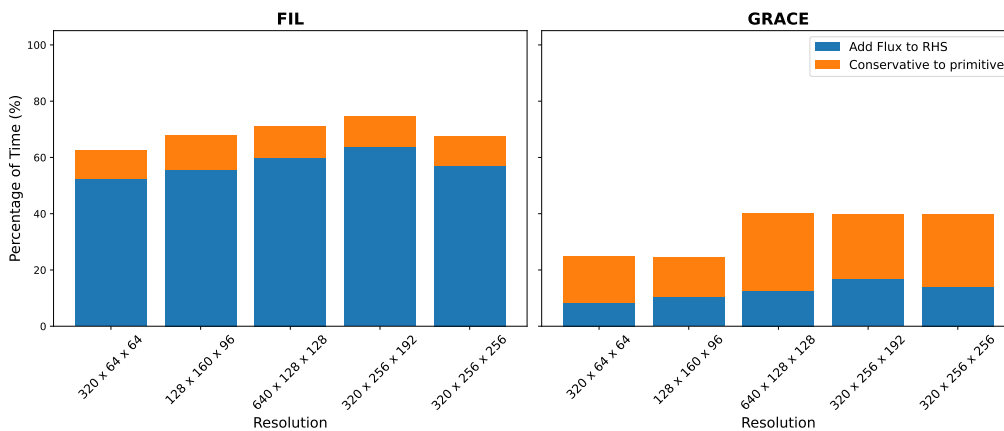


Figure 32: Comparison of percentage of time spent in Kernel 1 (Add fluxes to RHS) and Kernel 2 (conservative to primitive solver) for FIL and GRACE. This data was produced when FIL was run on a single processor on the p03 partition of the IT4I CS and a single M250 GPU on LUMI-G.

When the run times of kernels 1 and 2 were compared between GRACE and FIL, kernel 1 saw a 99% performance improvement and kernel 2 saw a 91% improvement when ported to GRACE. Additionally, we observe a reduction in the percentage of time spent in Kernel 1 during a run as seen in Figure 32. In FIL, Kernel 1 accounted for more than 50% of the total run time, but this has been reduced to less than 20% in GRACE. For Kernel 2, there is a slight increase in its run-time percentage; however, the total execution time is still reduced. Kernel 2, which is the conservative-to-primitive solver, relies heavily on conditional statements, which are not optimal for GPU architectures. Neural networks manage to avoid costly branching decisions by calculating every possibility and weighting the different solutions in a GPU computationally efficient way. Work is ongoing to train a network on the c2p kernels inputs outputs to replace the GPU inefficient kernel with a neural network, a GPU efficient kernel.

We expect a 25 times speed up to be the upperbound of the speed up that we will see by porting FIL kernels to GRACE. The shock tube test is less computationally challenging than a TOV or BNS merger simulation, but as the kernels that have been ported, are still the most compute intensive routines in the more challenging cases, we are confident that significant performance gains will still be seen when the rest of the FIL modules are ported.

7.2 Profiling activities

In this section we present profiling results for FIL. These results provide complementary feedback to the outcomes of the POP analysis regarding the performance and scalability of FIL. More precisely, we aim at revealing tuning opportunities for both the runtime and the HPC environment, which can help improve the performance of FIL and other applications. Moreover, we carry targeted around FIL's memory consumption. A memory footprint that increases significantly with the number of processes involved in weak scaling tests can hinder scaling out to a large number of nodes.

Note that we also plan to extend this type of targeted analysis to GRACE, as soon as the code for the various problems considered is mature enough. The GRACE application comes with some advantages when it comes to performance analysis. The code is currently being developed, which offers the opportunity to capture any performance shortcomings early enough in the development cycle. Furthermore, since it does not depend on ET, and thus has less overall dependencies on third party code, it makes performance debugging easier, with faster iteration.

7.2.1 FIL profiling: Performance variability

Numerous studies have outlined the sensitivity of applications to system noise [24, 25, 26]. The phenomenon is further amplified for parallel applications that use collective operations [27]. Late arrivals on primitives that act as implicit barriers (e.g. MPI_Allreduce) can prove hazardous to the overall application performance.

As a first step, we explore how FIL performance varies in weak scaling tests involving an increasing number of nodes. Runs are performed on the partition of the compute nodes (thus, no acceleration). Nodes are exclusively acquired through Slurm and are fully populated with MPI processes (one MPI process per core, summing up to 128 processes per node). The MPI implementation used is OpenMPI (main¹ branch) with point-to-point traffic handled by UCX [28] (v1.17.0). We test FIL solving the head-on-collision problem (HOC) for 40 iterations using both unigrid and AMR strategies. Note that we use half the number of cells for the case of AMR, since FIL defaults to two refinements levels, to help make the two configurations more comparable. However, direct performance-wise comparison between the AMR and unigrid modes is out of the scope at this stage of the profiling analysis. We execute each scenario three times.

Figure 33 shows the total time for each explored configuration, with the shaded areas depicting the spread in the results using a 95% Percentile Interval. We observe that the execution time can vary significantly, for both the unigrid and the AMR variants. This performance variability is especially profound in executions involving 8 to 16 nodes, with the minimum and maximum time varying by up to $1.8\times$. For the rest of this section, we will center our analysis around the unigrid variant.

Focusing on the scenario of 8 nodes with a problem size of $800 \times 800 \times 400$ cells, in Figure 34a we measure the rank-average *CCTK Evolution* time for five different FIL-HOC runs; the evolution step makes up a major percentage of the overall execution time. As the figure shows, after the 10th iteration, the evolution time exhibits significant variation between runs. The maximum difference is observed between runs 0 and 1: up to $2.5\times$.

¹#b44ecf0

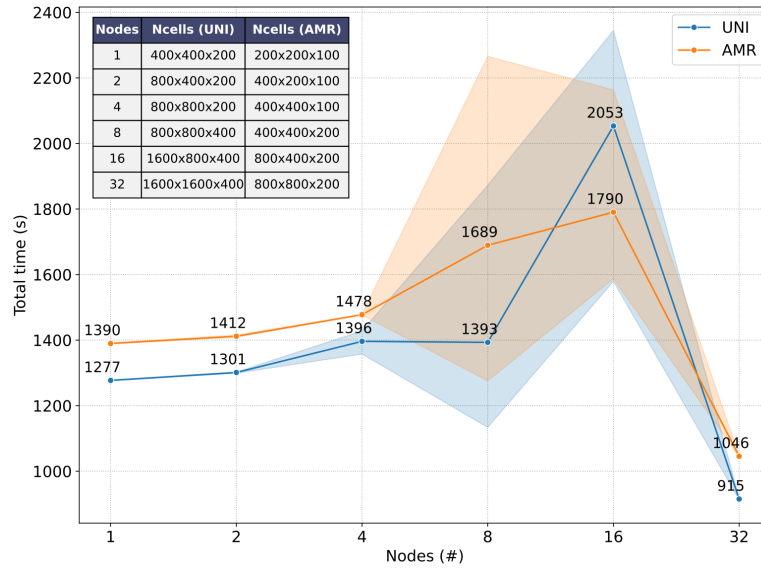
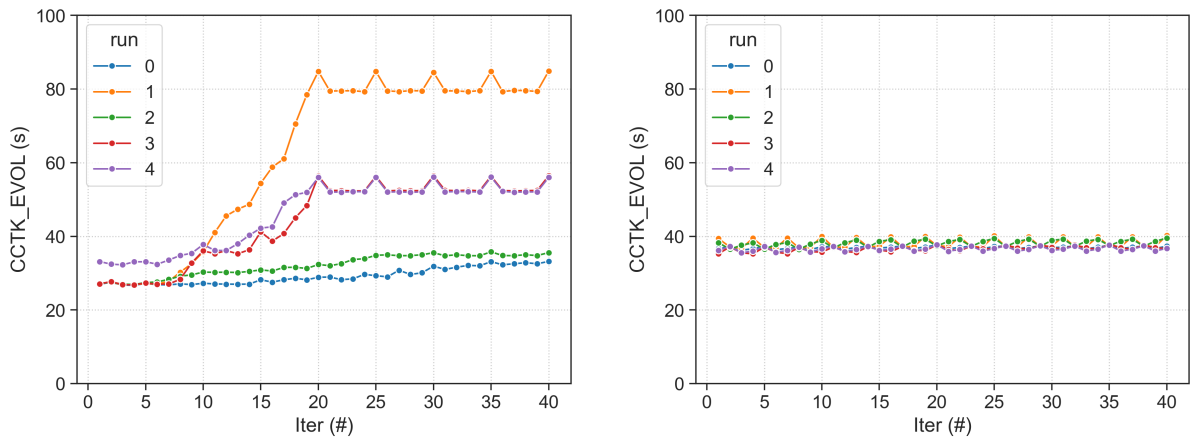


Figure 33: FIL-HOC performance variability: unigrid and AMR strategies



(a) Fully populated node

(b) 4 idle cores per socket, per node

Figure 34: FIL-HOC: Evolution time across different iterations of different runs

We worked in close collaboration with IT4I towards determining the cause of this performance variability (as well as the one observed in MPIAllreduce micro-benchmarks, which we explore in Section 7.2.3). Even though at the time of writing we have not yet pinpointed the exact origin of the anomaly, we have arrived at an effective workaround to alleviate it. The workaround involves leaving 4 cores unused on each CPU socket and on each node on which FIL is run. Figure 34b presents the average evolution time of FIL-HOC on 8 nodes with a problem size $800 \times 800 \times 400$ cells, akin to Figure 34a, but with the workaround applied. We see that now all different runs exhibit similar performance. Note that according to our data, reducing the active number of cores on each node does not generally result in significant performance loss. To the contrary, in some preliminary results, these configurations actually perform better instead. However, fully validating this outcome is part of our ongoing work.

Let us note that we view this workaround as a temporary one. It does not reveal the root cause of the performance variability, nor does it expose any FIL- or runtime-specific reason for it. In that regard, exploration for its source is an ongoing effort. Part of the effort also involves determining whether similar phenomena appear on other HPC clusters, like *Lumi* or *Leonardo*.

7.2.2 FIL profiling: Rank load imbalance

One other performance anomaly we've identified in FIL-HOC, is imbalance in the amount of time that each rank spends in the evolution step. Figure 35 shows the cumulative CCTK_EVOL time for different ranks, in an experiment on 8 nodes with a problem size of $800 \times 800 \times 400$. The experiment is repeated 5 times, and the shaded areas depict the spread in the results using a 95% Percentile Interval. Note that in this scenario, we have applied the workaround explored in Section 7.2.1 –leaving 4 cores on each CPU socket idle– to make load balance effects easier to discern. Interestingly, we observe a repeating pattern in the results, with 13 different logical *groups* formed. We see that there are some significant differences between certain groups, as well as some not-as-significant variability between ranks in the same group. Overall, the maximum difference observed between the ranks with the lowest and the highest time reaches 20%. It is important to note that, the evolution

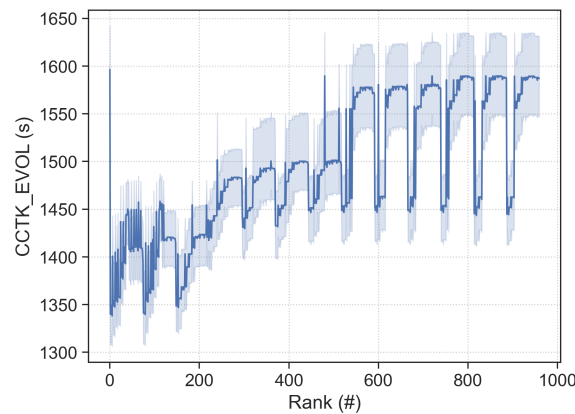


Figure 35: FIL-HOC: CCTK_EVOL load imbalance

time depicted on Figure 35 is the result of both computation and time spent in MPI communications. We are currently exploring whether it is feasible to have a breakdown of the evolution time and identify which is the main contributor to the aforementioned performance difference.

7.2.3 FIL profiling: Allreduce variability

Apart from exploring the performance of the whole application, our profiling effort also includes targeted tests for the runtime. As the number of cores per node increase, so does the number of processes that run on each node. In that respect, the performance of MPI collectives at the intra-node scale becomes increasingly important.

In this section, we benchmark the performance of MPIAllreduce, using the OSU micro-benchmark suite [29]. We utilize OpenMPI, paired with a component we have developed over the past years, that implements optimized intra-node collectives: *XPMEM-based Hierarchical Collectives* (XHC) [30]. The XHC component is now also a

part of the upstream OpenMPI project, and the development effort towards optimizing multiple aspects of the primitives it implements continues.

In Figure 36, we show the Allreduce performance on a single node of the compute partition on the Karolina Cluster. The node is exclusively allocated through Slurm, and is fully populated with MPI processes. In this scenario, we run the benchmark over 100 times. The benchmark tests multiple buffer sizes in one invocation; within a specific message size, each point represents the result of a single execution, and the darker the color, the later the run. The expectation is that every execution of the benchmark should yield equal performance, with a minor degree of error at most. Instead, we observe that for large sizes, above 1 MB, there is a consistent pattern of continuously increasing latency in successive runs. At the far end of the spectrum with 8 MB messages, the performance between the first and the last execution varies by up to $1.75\times$, from ~ 13 ms to ~ 23 ms.

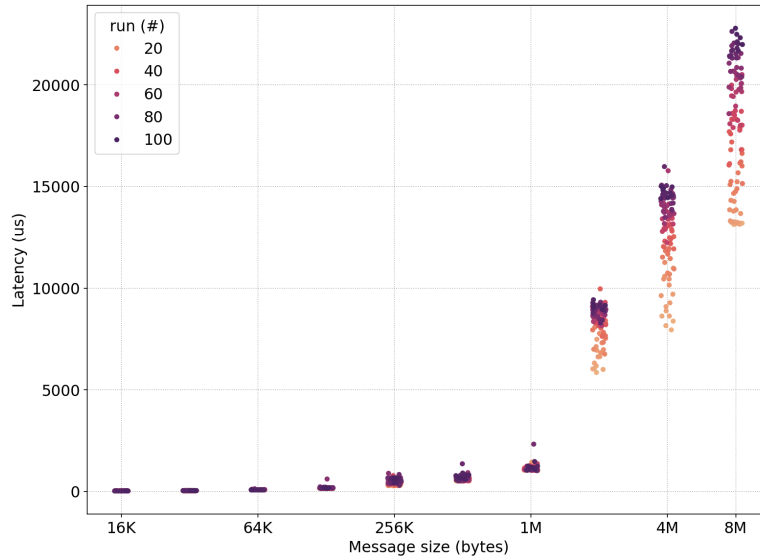


Figure 36: XHC Allreduce on a single Karolina node

Interestingly, this performance anomaly does not occur in all MPI collectives. XHC’s Broadcast (respective benchmark results shown in Figure 37a) retains the same performance throughout all of the runs. We have also found that there are additional Allreduce implementations that exhibit the same phenomenon, like OpenMPI’s *coll/acoll* [31] component (Figure 37b), while other ones, like OpenMPI’s *coll/tuned*, do not. Finally, XHC’s Allreduce does not suffer the same degradation on Lumi (Figure 37c), on nodes with the same generation AMD Epyc CPUs as on Karolina’s compute partition nodes.

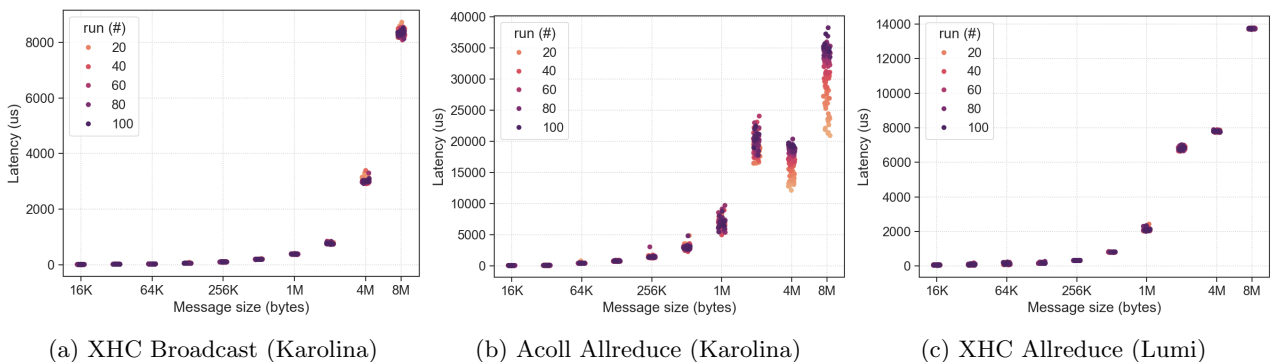


Figure 37: Single-node collectives latency

It’s not entirely clear what the exact root cause of this performance anomaly is. As with the performance variability problem in FIL-HOC we explored in Section 7.2.1, we carried out several tests to narrow down the search space. One of the things we suspected was the CPU core frequency, which could be dropping as a result of some type of thermal throttling. Karolina also implements a set of energy saving measures [32] that limits the

CPU core frequency on compute nodes to 2.1 GHz. While this can affect the performance of MPI collectives, we expect its effect to be the same across all executions of our benchmark. In collaboration with IT4I, we measured the CPU frequency of the system, using either IT4I’s Grafana-based job metrics dashboard, or Linux’s sysfs interface (`/sys/devices/system/cpu/cpu*/cpufreq/scaling_cur_freq`). Our testing shows that, the average CPU core frequency remains consistent at the intended 2.1 GHz, across all 100 repetitions of the benchmark.

The next usual suspect is system-related noise, or some other form of interference that can impact system performance. Using the same approach as in Section 7.2.1 we repeat the benchmark of Figure 36, once again on a single compute node of the Karolina cluster, but with 4 cores on each CPU socket left idle this time around. The results, presented in Figure 38, show that the reported performance now remains consistent across all runs, and that this workaround is thus able to relieve this performance anomaly as well.

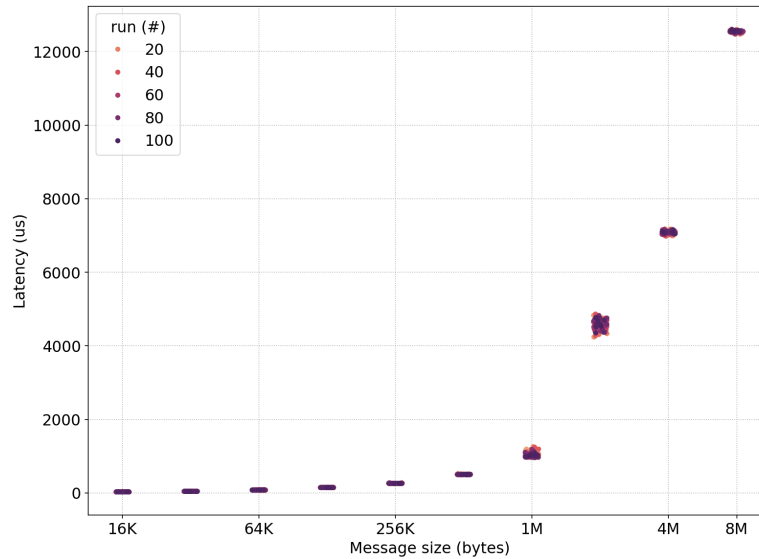


Figure 38: XHC Allreduce on a single Karolina node, 4 cores idle on each socket

At this point, we should note that while FIL does utilize the Allreduce primitive, its use is in the control path; the low call frequency as well as the small message size involved do not make it plausible that FIL’s performance variability is a direct result of performance variability in the Allreduce implementation. Instead, the current mindset at the time of writing is that these are two different phenomena of potentially the same root issue. Furthermore, even though we arrived at the aforementioned workaround as method to counteract system noise, the observed effects are unusual. The magnitude, persistency, and the predictable & non-random nature of the anomaly, as well as the way that the degradation builds up over successive executions, do not immediately resemble commonly encountered system noise. With that in mind, the work to determine the root cause of the issue continues, with several approaches considered.

7.2.4 FIL profiling: Weak scaling tests

In this section we focus on weak scaling performance with FIL-HOC, using the unigrid variant for simplicity. In Section 7.2.1, Figure 33, we used a base problem size of $400 \times 400 \times 200$, which we doubled every time we doubled the node count. Unfortunately, extending this trend to 128 or more Karolina compute nodes results in insufficient memory. Therefore, for these tests, we limit the base problem size to $350 \times 350 \times 100$. We have scaled our runs up to 256 compute nodes of the Karolina Cluster. Running on 512 nodes, is possible, but will require allocating a substantially large percentage of the Karolina cluster. It is currently planned as a follow-up step.

On Figure 39a, we present the total execution time of FIL-HOC for each node count. Each point is an average of 5 executions, with the shaded area showing the variability in the results using a 95% Percentile Interval. The ideal result in a weak scaling test would be straight and as horizontal as possible line. However, as these results show, there is quite a bit of volatility in runs involving up to 32 nodes. what’s even more troubling though, is the rapid performance hike we observe from 32 to 256 nodes; from 128 to 256 nodes the execution time more than doubles, even though the problem size per process remains constant.

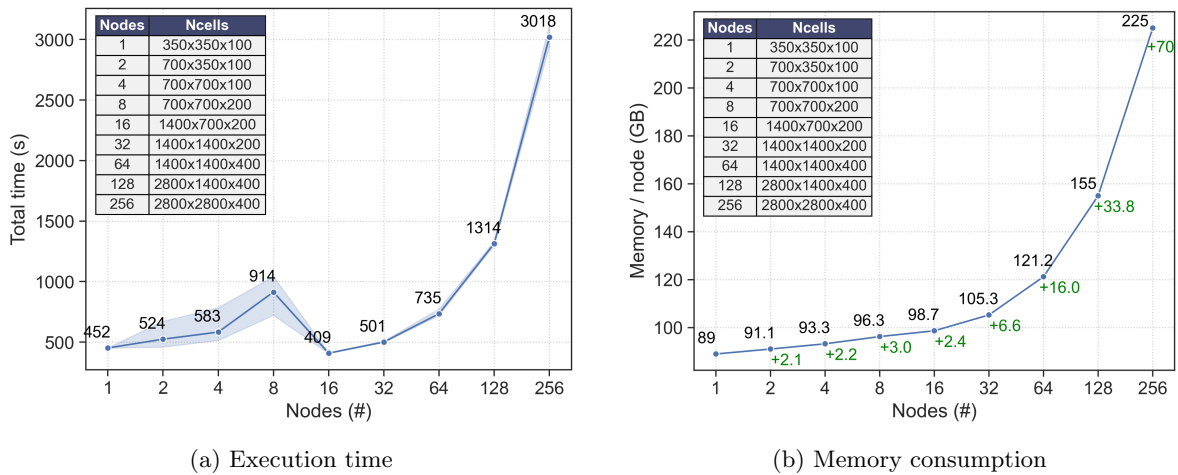


Figure 39: FIL-HOC: Weak scaling up to 256x Karolina compute nodes

There is a list of current action items that aim at narrowing down the cause of this poor scaling performance. We are exploring whether the issue is runtime specific, or if it originates in the application. We are also experimenting with hybrid MPI+OpenMP configurations, which we go into more depth in Section 7.2.5. Furthermore, Load balancing issues, like those demonstrated in Section 7.2.2 might be contributing to this situation as well. Unfortunately, the general Einstein Toolkit code base is sufficiently complex that diving into it to study such phenomena is not a trivial endeavor.

It's also important to consider how memory consumption scales. On Figure 39b we show for each node count the maximum amount of memory required per single node. These results are obtained through Slurm's *MaxRSS* job statistic. We see that even though the problem size per process and per node remains constant, the required memory rises continuously. The increase in node counts from 64 to 256 is especially intense, with the 256-node setup requiring 45% more memory than the 128-node one.

This increase in required memory with more nodes is unintuitive and unexpected. When the problem size doubles with each new node count, so also does the number of processes. This means that the ratio of number of cells per each process remains constant, but the memory required for each process grows nevertheless. There is ongoing effort around this effect. Utilizing tools like Valgrind's Massif [33] we are able to profile FIL's memory consumption, in order to determine where the increased memory consumption originates. This has also revealed other memory optimizations in-wait; for example, we've managed to optimize the OpenMPI runtime, through disabling one unused transport-layer component (UCT), reducing the overall memory required per node by 5 GB. Part of the effort, in collaboration with GUF, also includes some interaction with the Einstein Toolkit development team.

7.2.5 FIL profiling: MPI+OpenMP

One way to tackle the memory scaling issue in FIL-HOC, is to utilize OpenMP along with MPI. In this hybrid MPI+OpenMP approach, which FIL supports out-of-the-box, we trade out some of the MPI processes for OpenMP threads, essentially swapping MPI parallelism for OpenMP parallelism. Each MPI process, of which there are now fewer overall, is assigned a couple of OpenMP threads; because all these threads share the same memory, we are able to cut down on the overall memory consumption.

In Figure 40a, we show the memory scaling performance of FIL-HOC using a hybrid MPI+OpenMP approach, on the Karolina compute node partition, with multiple OpenMP thread configurations. The total number of OpenMP threads on each node is always the same as the amount of CPU cores (128), so e.g. with 8 OpenMP threads per process, there are 16 MPI processes per node. The configuration with 1 OpenMP thread/proc is a pure-MPI scenario, as a reference for comparison. Note that unlike the experiments above, this one uses a base problem size of $400 \times 400 \times 200$. At the moment, for node counts of 64 and above, we only have data with 4 OpenMP threads/proc. Overall, we can see that the MPI+OpenMP configurations scale much better in terms of memory consumption, with an inversely proportional relationship between the max memory per node and the number of OpenMP threads. This is especially important because it allows us to scale to ever-higher node counts without having to sacrifice on the size of the problem we are solving.

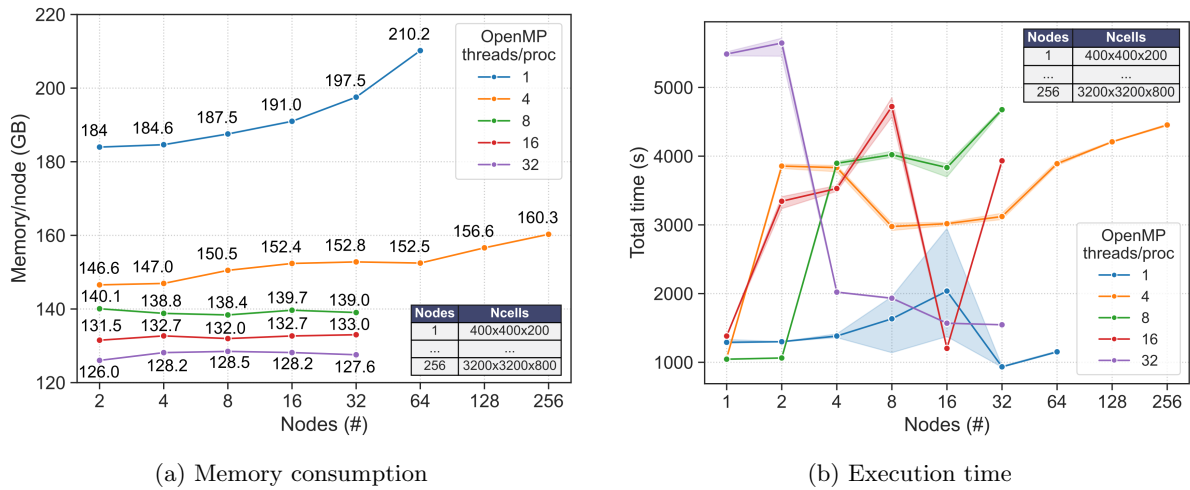


Figure 40: FIL-HOC: MPI+OpenMP weak scaling

While the hybrid approach reduces is effective for keeping the required memory in check, we must also consider its implications in regards total execution time. Figure 40b shows for each of the aforementioned MPI+OpenMP configurations the time required to solve the problem, as well as how it scales with increasing node counts. We notice that the hybrid approaches exhibit high degrees of volatility across different node counts, as well poor performance overall compared to the pure-MPI configuration, in many node counts. Ultimately, in this state of affairs, the MPI+OpenMP approach is not a silver bullet solution to the memory problems encountered in the pure-MPI method.

7.3 Conclusion

Our profiling analysis focuses on the FIL code addressing the head-on-collision problem. We rely on FIL's pure MPI mode and our results are mostly derived from the Karolina cluster. However, our analysis does not target solely the application code; it is also aimed at revealing any tuning capabilities for the machine and the runtime used. In that respect, we explored the performance stability of the FIL-HOC problem on the Karolina cluster, and found that there is some notable variability. We inspected multiple settings related to the Karolina cluster nodes with the most interesting parameter being the occupation ratio of each node. Performance variability was also explored in the context of a simple MPI Allreduce micro-benchmark, with an interesting phenomenon observed. Another aspect in regards to scaling FIL-HOC to higher node counts is memory consumption. Our initial results reveal significant memory consumption, which hinders scaling to larger node counts. One minor contributor to this memory consumption was a mis-tuning of the OpenMPI runtime, but this was not sufficient to fully address the problem. Another interesting finding is the apparent load imbalance among MPI processes, captured through FIL's evolution time. Our current hypothesis is that this results in a *late arrival effect* kicking in the Allreduce used after each iteration. The outcome that is less trivial to analyze and determine the root cause of, is the performance of the weak scaling tests. Total time seems to increase unexpectedly when scaling out to high node counts.

Several threads of work are ongoing in an effort to analyze and fully understand all the aforementioned findings. Wherever meaningful, the steps of the analysis are, or, will be replayed on other machines. Allreduce performance variability was for example also explored on the Lumi cluster. However, we need to carefully select which experiments are performed, since reserving high node counts and performing multiple runs of these tests consumes a lot of resources. Additionally, we plan to also explore whether the performance issues encountered with FIL are also present with GRACE, which is at its early stages of development; this offers the opportunity to capture and address any issues early enough.

8 ChaNGa

ChaNGa [34][35][36] is an N-body and smoothed particle magneto-hydrodynamics (SPMHD) code which is used to study a wide array of astrophysical systems. While the gravity and SPMHD algorithms are based on the gasoline [37] and pkdgrav [38] codes, the unique feature of ChaNGa is its implementation of the Charm++ framework, which enables highly efficient parallel scaling. Charm++ employs overdecomposition to achieve this. That is to divide the work into many more pieces (chares/tree pieces) than you have processors and let the Charm++ runtime system load balance by appropriately assigning pieces to real processors (see D1.2 for an overview). During runtime, Charm++ applies dynamic load rebalancing strategies, to determine which tree pieces should be migrated to new processors for better load balance. In addition, the SMP mode of Charm++ leverages the shared-memory systems found in high-performance computing environments. Within an SMP process/node, one thread is assigned as the communication thread, responsible for internode communication, while the remaining threads, act as worker threads in charge of the processing elements. Multiple SMP processes can be initiated per network node, the number of which should be based on the CPU architecture and communication load. Charm++ also provides support to execute CUDA kernels on the GPU asynchronously and to manage data transfers between the CPU and GPU. ChaNGa is publicly available at github (<https://github.com/N-BodyShop/changa>).

8.1 Use-case description

Benchmarking and profiling were performed for two of our science cases. The first is a simulation of a major merger between two Milky Way-like galaxies, which include "full-physics" that includes gravity, hydrodynamics, feedback and radiative cooling. This use-case was previously used for the performance analysis of Deliverable D2.1. Further details of this simulation can be found in Deliverable D1.1 and D2.1. The corresponding parameter file can be found in Deliverable D2.1 (we run for 50 time steps here instead of just 10). We have used this to compare our current CPU version to the CPU version at the start of this project.

For the GPU scaling and GPU/CPU comparison we have used the cosmological box of 25Mpc size and 2 billion particles. In this deliverable we run this with gravity only to test the new multi-GPU capabilities. This use case is also described in D1.1. Below is the corresponding parameter file for this use-case:

```
#
# Cosmological box 25 Mpc
#
dHubble0      = 2.894405
dOmega0       = 0.3077
dLambda       = 0.6923
dMsolUnit     = 2.00139e+15
dKpcUnit      = 25000.
# roughly 1/10000th of a Hubble time; single stepping
dDelta = .1e-5
dEta = .2
dTheta = .6
bComove = 1
bPeriodic = 1
nReplicas = 1
dPeriod = 1.0
achInFile = cosmo25_1024_hydro_IC.dat
achOutName = cosmo25_1024
iLogInterval = 1
iCheckInterval = 1000
bBenchmark = 1
bDoGas = 0
bConcurrentSph = 1
bFastGas = 1
dFracNoDomainDecomp = 0.25
nTruncateRung = 128
nSmooth = 32
#dRedTo = 0.0
nSteps = 20
```

8.2 Benchmarking results

Benchmarking was conducted on the Karolina cluster of IT4I. Karolina has 720 compute nodes without accelerators, which is equipped with 2 AMD EPYC™ 7H12 processors (64 cores, 2.6 GHz), providing a total of 128 CPU cores per node. We run our simulation across 4 to 64 nodes, utilizing 8 SMP tasks with 8 communication threads and 120 worker threads per node. Table 28 shows the performance of the current code version (commit 2) compared to the old version of the code (commit 1) as it entered the SPACE CoE project in January 2023. Here we have included the full runtime of the simulation and the median time per step of the Gravity+SPH, star formation + feedback, domain decomposition (DD) + load balancing (LB) + tree building (TB) and radiative cooling. We have grouped them in this way to get significant time data for all the different modules and to highlight certain improvements. To focus on computational efficiency during benchmarking, output operations were disabled and we increased the number of steps to 50 (compared to D2.1).

The new implementations show about a 12% improvement in overall runtime across the board. Showing an additional increasing improvement with scaling, reaching a gain of 39% for 64 nodes compared to the older version. This demonstrates an overall improvement in runtime and scalability of the new code. In the following sections we will go into more detail on the improvements of the code and how they effect the timings of each grouping of modules.

8.2.1 Tree-Piece replication

During our performance analysis (D2.1) it became clear that ChaNGa suffers from communication imbalance for highly clustered datasets involving all the main physics modules (as in the merger case here). We have tackled this issue by introducing a tree piece replication method within the code. In this method we replicate the information about the tree nodes on multiple processors, which spreads out the communication load and ensures that no single processor becomes overloaded with messages.

This optimization is highly related to the improvement in scaling, which we can see when we compare the high node timings. This results in a gain of 39% of the entire code runtime for 64 nodes and a massive 39% for gravity + hydro and 45%. Due to the improvements in overall communication we also see some slight improvements in the radiative cooling. A negative consequence of using tree piece replication is that the tree piece replication add some small overhead, which leads to a small overall slowdown to the tree build (TB) module. The overhead is very small in absolute timing compared to the other module timings.

	Entire code runtime [s]		
# of nodes	commit 1	commit 2	gain [%]
4	1068.48	945.4	12 %
8	629.494	550.7	13 %
16	376.527	321.5	15 %
32	274.766	222.2	19 %
64	300.9	182.4	39 %

	Gravity + Hydro / step [s]			Star formation + Feedback / step [s]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
4	8.745767	8.280423	5%	11.3513	7.688752	32%
8	4.543043	4.31	5%	6.889945	4.611961	33%
16	2.441275	2.246674	8%	3.975085	2.683517	32%
32	1.960952	1.335391	32%	2.347926	1.653631	32%
64	2.287406	1.02	55%	2.401843	1.31169	45%

	DD + LB + TB / step [s]			Radiative Cooling / step [s]		
# of nodes	commit 1	commit 2	gain [%]	commit 1	commit 2	gain [%]
4	0.550649	1.289938	-134 %	1.327588	1.273277	4 %
8	0.411551	0.866464	-110 %	0.725301	0.706334	3 %
16	0.397413	0.491421	-23 %	0.396255	0.359029	9 %
32	0.330386	0.387656	-17 %	0.222024	0.242104	8 %
64	0.343311	0.409728	-19 %	0.190325	0.237291	20 %

Table 28: Current code version (commit 2) compared to the old version of the code (commit 1). We run the simulation on the non-accelerated Karolina nodes across 4 to 64 compute nodes. We list the timings of the full runtime of the simulation, the median time per step of the gravity + hydrodynamics, star formation + feedback, domain decomposition (DD) + load balancing (LB) + tree building (TB) and radiative cooling. Gain is calculated following $1 - t_{\text{commit2}}/t_{\text{commit1}}$.

8.2.2 Improved load balancer

Previously the Orthogonal Recursive Bisection (ORB) method have been used to distribute tree pieces among the processors. The ORB strategy, unfortunately, occasionally resulted in scenarios where some processors were left without any assigned tree pieces. We have improved this by replacing the ORB strategy with an space filling curve (SFC) strategy, where we instead sort the tree pieces according to an SFC. The sorted list is then divided evenly to ensure a balanced distribution of tree pieces among processors. This approach eliminates the problem of idle processors and improves overall load balancing efficiency. We also cleaned up the load balancing code a bit, which included the removal of some unnecessary counting of active particles.

This optimization is highly related to the overall improvement in the runtime of the code (15% gain), especially for the load balancing involved in star formation and feedback (32% gain).

8.3 Profiling activities

For the previous deliverable D2.1, ChaNGa did not have the capabilities to run with multiple GPUs per node. This was due to issues with race-conditions produced by the past GPU implementation, related to having GPU kernel launches tied to the TreePieces. Another issue with having GPU kernel launches tied to TreePieces is that it degrades the performance of the application when using a large number of TreePieces (see Figure 41). However, using enough TreePieces is crucial to provide efficient load balancing. To address these issues, the GPU versions of the local gravity tree walk and the cosmological boundary conditions are now handled by the data manager. This resolved the race-condition issue which could occur when using multiple GPUs per node.

In the following section we test the scaling of our multi-GPU version and compare the performance of the GPU version to the CPU version of the code.

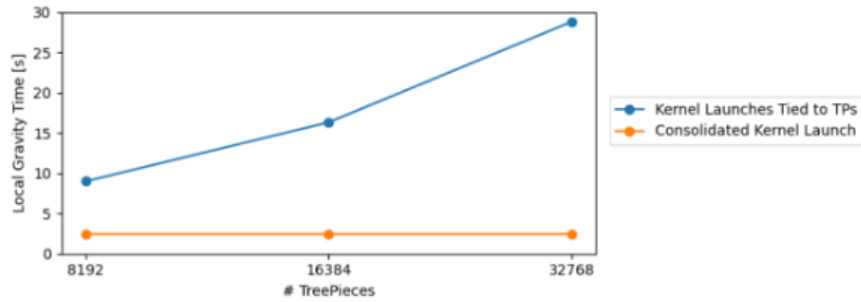


Figure 41: The difference in local gravity time from having GPU kernel launches tied to the TreePieces (TP) compared to consolidated kernel launches being handled by the data manager. Performance shows a strong dependence on the number of TPs when launching from TPs, whereas the new implementation exhibits no significant scaling impact.

8.3.1 GPU scaling and GPU/CPU comparison on Karolina cluster

We have performed this comparison and scaling on the Karolina cluster. We use the non-accelerated nodes (described in 8 for the CPU runs and the accelerated nodes for the GPU runs. Karolina has 72 accelerated nodes, which is equipped with 8 NVIDIA A100 GPUs and 2 AMD EPYC™ 7763 processors (64 cores, 2.45 GHz), yielding 128 CPU cores per node alongside the 8 GPUs. The only parameter difference between the CPU and GPU runs is the number of tree pieces and buckets, which was optimized to produce the best results. We use 500000 tree pieces and a particle bucket size of 24 for the CPU runs and 100000 tree pieces and a particle bucket size of 64 for the GPU runs. We run our GPU simulations across 4 to 32 nodes (32 to 256 GPUs), utilizing 8 SMP tasks with 8 GPUs, 8 communication threads and 120 worker threads per node. For our CPU simulations we run across 16 to 256 nodes, utilizing 8 SMP tasks, 8 communication threads and 120 worker threads per node.

In Figure 42 we present the results of the GPU scaling. Since the initial setup phase (including data loading) constitutes a significant portion of the runtime, we include an additional curve with this overhead removed (GPU-NOINIT). We also analyze the scaling of the gravity module (including cosmological boundary conditions), which is the main component offloaded to the GPU. The results demonstrate great scaling, with a parallel efficiency exceeding 80% for the total runtime up to 16 nodes (128 GPUs) and up to 256 GPUs for the gravity module alone.

Table 29 compares the per-node performance between CPU and GPU runs. The GPU accelerated runs achieves a 72–80% gain in total runtime and a 83–91% improvement in the gravity module compared to the non-accelerated CPU runs. Notably, 16 GPU-accelerated nodes match the gravity computation performance of 256 non-accelerated nodes (3.83 s vs. 3.44 s per step).

From the results we can see that there are load balancing issues when running with multi-GPUs, as evidenced by significant oscillations in step times (see the maximum and minimum times in Table 29). This behavior likely stems from the current load balancing scheme not accounting for GPU-specific workloads during distribution.

8.4 GPU and MPI Profiling activities

This analysis examines the GPU implementation of ChaNGa (Charm++ N-body Gravity solver), a GPU-accelerated cosmological N-body simulation code using the Charm++ parallel programming model. The analysis focuses on GPU utilization patterns, Charm++ communication characteristics, memory management efficiency, and computational bottlenecks across 8 ranks executing on NVIDIA A100 GPUs.

Performance data was collected using NVIDIA Nsight Systems during a representative 8-rank production run. The profiling data was exported to SQLite format for detailed analysis of kernel execution statistics, memory transfer patterns, MPI communication, and computational efficiency metrics.

This report presents a comprehensive performance analysis of ChaNGa (Charm++ N-body Gravity solver), a GPU-accelerated cosmological N-body simulation code using the Charm++ parallel programming model. The analysis examines GPU utilization patterns, Charm++ communication characteristics, memory management efficiency, and computational bottlenecks across 8 ranks executing on NVIDIA A100 GPUs.

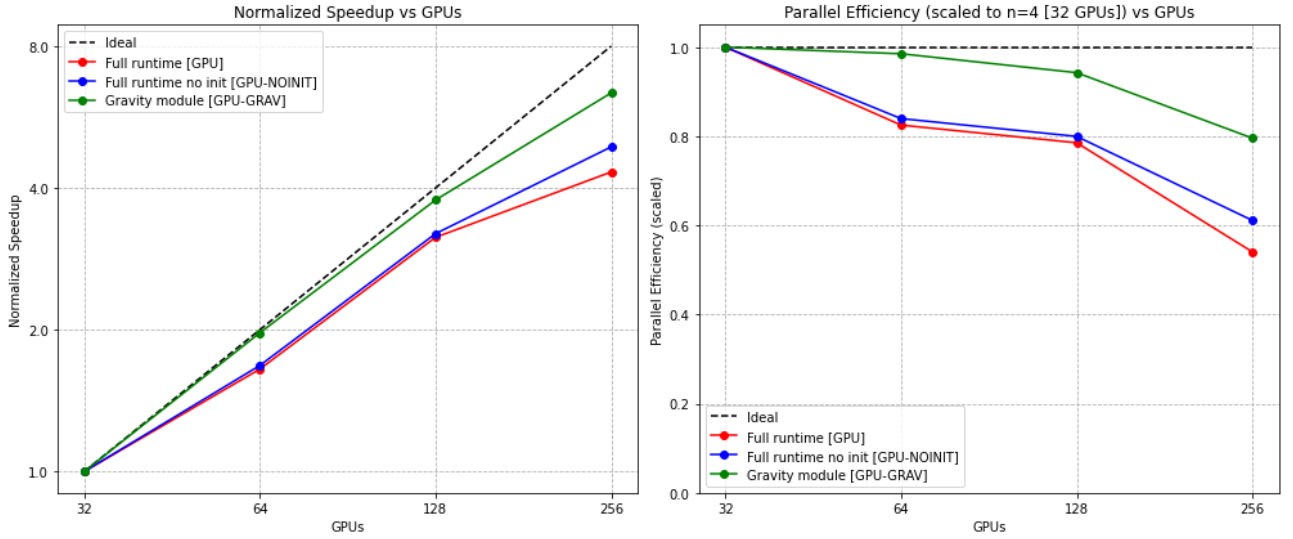


Figure 42: GPU scaling tests for the cosmological box case performed on the Karolina supercomputer (details provided in deliverable D1.1 and D2.1). The plot compares three metrics: total runtime (GPU), runtime excluding initialization (GPU-NOINIT), and gravity module execution time (GPU-GRAV). Speed-up is normalized to 1 at 4 nodes (32 GPUs), with relative speed-up shown on the left and parallel efficiency on the right. We see a high parallel efficiency of above 80% up to 128 GPUs for the entire runtime and up to 256 GPUs for the gravity module.

# of nodes	Full runtime (20 steps) [s]			Gravity / step [s]						
	CPU	GPU	gain [%]	CPU[median]	min	max	GPU[median]	min	max	gain [%]
4	-	840.259	-	-	-	-	14.5	13.21	17.67	-
8	-	509.22	-	-	-	-	7.33	7.08	7.67	-
16	1352.26	267.462	80 %	52.46	52.40	52.59	3.83	3.68	5.88	93 %
32	696.593	194.322	72 %	26.34	26.31	26.40	2.27	2.06	3.96	91 %
64	377.519	-	-	13.26	13.24	13.29	-	-	-	-
128	215.833	-	-	6.73	6.69	6.78	-	-	-	-
256	150.527	-	-	3.47	3.43	3.68	-	-	-	-

Table 29: Performance comparison between GPU and CPU versions on the Karolina supercomputer. The GPU version was executed on accelerated nodes (4-32 nodes, 32-256 GPUs), while the CPU version ran on non-accelerated nodes (16-256 nodes). We present: (1) total simulation runtime, and (2) gravity module timings (median, minimum, and maximum per-step values computed over the last 10 steps). Performance gain is calculated as $1 - t_{commit2}/t_{commit1}$.

Key findings reveal significant optimization opportunities:

- **Modest GPU utilization:** Average kernel execution time represents only 6.1% of total runtime (10.5s kernel time vs. 173.1s total duration), indicating substantial opportunities for improved GPU utilization.
- **Extensive Charm++ communication:** 82.47 GB of inter-rank communication across 16.1 million messages demonstrates the code’s distributed nature, with communication patterns showing good neighbor-to-neighbor interaction typical of well-decomposed domain structures.
- **Efficient memory management:** Peak GPU memory usage of 3.4-3.7 GB per rank with 22.9 GB average data transfer suggests reasonable memory efficiency for the computational workload.
- **Well-balanced execution:** Consistent execution times across ranks (173.1-173.2s) indicate good load balancing inherent to Charm++’s adaptive runtime system.
- **Compute-transfer optimization potential:** Analysis reveals opportunities for improved kernel-memory overlap and computational intensity optimization.

The analysis indicates ChaNGa demonstrates characteristics of a well-engineered parallel code with room for GPU-specific optimizations that could yield 20-40% performance improvements.

8.4.1 Performance Metrics Overview

The following table summarizes the key performance metrics across all eight analyzed ranks:

Metric	R0	R1	R2	R3	R4	R5	R6	R7	Avg
Total Duration (s)	173.1	173.1	173.1	173.1	173.1	173.1	173.1	173.1	173.1
Kernel Time (s)	10.5	10.5	10.3	10.4	10.3	10.4	10.4	10.4	10.4
GPU Util. (%) ¹	6.1	6.1	5.9	6.0	5.9	6.0	6.0	6.0	6.0
Data Transfer (GB)	22.9	22.7	22.7	22.7	22.7	22.7	22.7	22.7	22.7
Peak Mem. (MB) ²	3399	3714	3714	3714	3714	3714	3714	3714	3675
Mem. Util. (%) ³	8.5	9.3	9.3	9.3	9.3	9.3	9.3	9.3	9.2
CUDA Calls (M) ⁴	4.1	4.1	4.1	4.1	4.1	4.1	4.1	4.1	4.1
Kernel Launches	526	526	526	526	526	526	526	526	526
Memory Ops ⁵	2366	2366	2366	2366	2366	2366	2366	2366	2366
MPI Calls (M) ⁶	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8

Table 30: Performance metrics summary across all ranks. Notes: ¹GPU Utilization: Percentage of total execution time spent in GPU kernels (Kernel Time / Total Duration). ²Peak GPU Memory usage during execution. ³Memory Utilization: Peak GPU Memory divided by total available A100 GPU memory (40 GB). ⁴CUDA Runtime API calls in millions. ⁵Memory transfer operations (Host↔Device). ⁶MPI communication calls in millions.

These metrics reveal a well-balanced parallel application with consistent performance across ranks. The moderate GPU utilization (6.0% average) suggests the computational workload is either I/O bound, communication bound, or could benefit from algorithmic optimizations to increase computational intensity. The extremely consistent performance across ranks demonstrates Charm++’s effective load balancing capabilities.

8.4.2 Charm++ Communication Analysis

ChaNGa’s use of Charm++ provides sophisticated load balancing and communication patterns that differ significantly from traditional MPI-based codes. The communication analysis reveals patterns typical of well-decomposed tree-based N-body algorithms.

As shown in Figure 43, the analysis reveals a **significant communication volume** of 82.47 GB across 16.1 million messages, demonstrating the substantial data exchange typical of tree-based N-body algorithms. The communication pattern shows **neighbor-heavy communication**, with strong data exchange between adjacent ranks (e.g., 0↔1, 1↔2, 2↔3), suggesting effective spatial domain decomposition that optimizes locality.

The communication also exhibits **asymmetric patterns**, with volumes varying significantly from 134.7MB to 4.3GB between different rank pairs. This reflects Charm++’s adaptive load balancing responding to computational heterogeneity in the gravitational tree structure. Importantly, there are **no communication bottlenecks**, as the communication load is well-distributed with no single rank acting as a central hub.

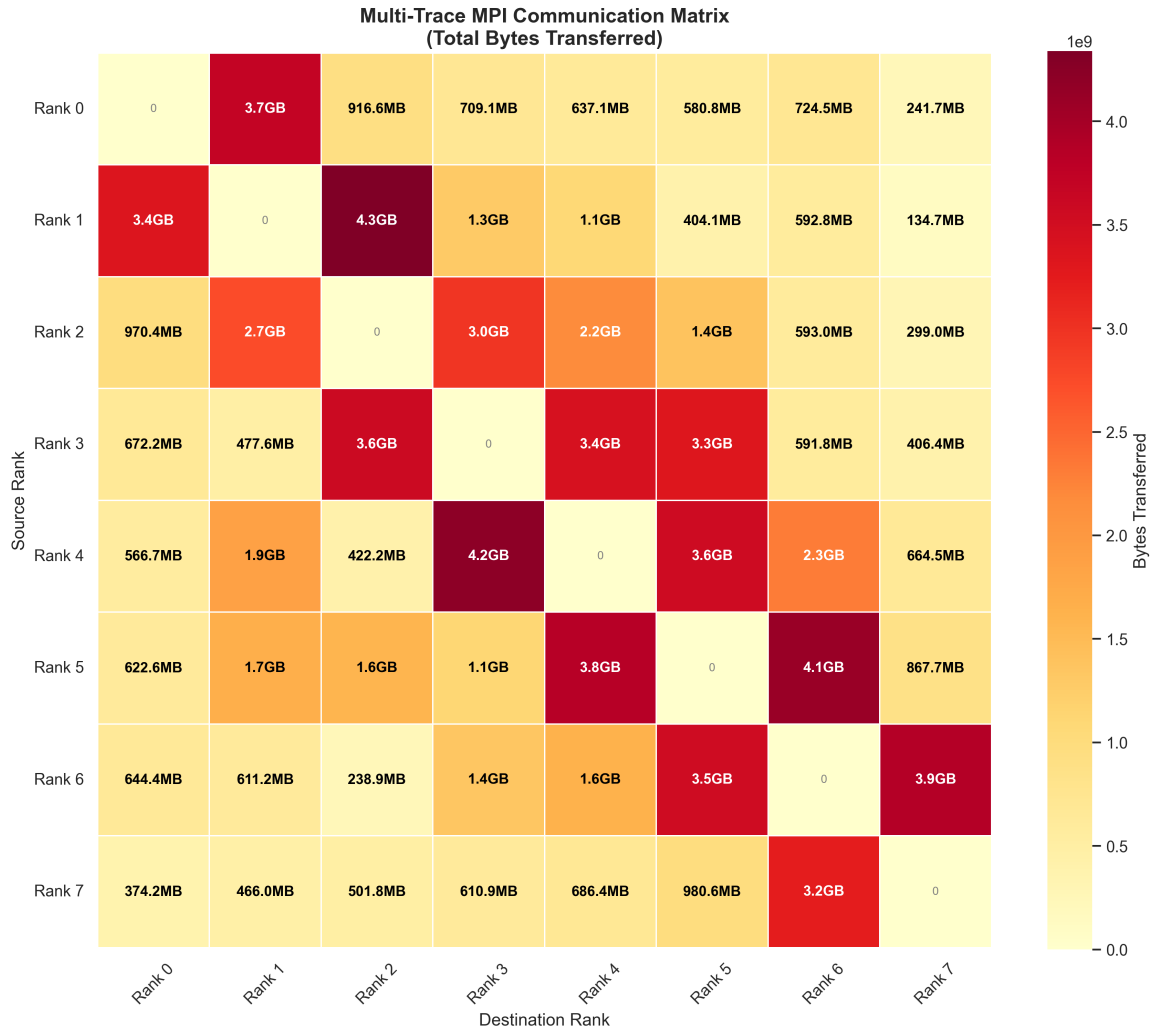


Figure 43: Inter-rank communication matrix showing data transfer volumes between ChaNGa ranks. The asymmetric pattern reflects the tree-walking algorithm’s dynamic load distribution.

The communication statistics reveal several important aspects of ChaNGa’s parallel execution behavior. The application demonstrates **balanced send/receive operations**, with nearly equal numbers of send (1.90M) and receive (1.88M) operations indicating good communication balance across the parallel execution. However, there are opportunities for optimization, particularly in **message aggregation**, as the average message size of 3.8-4.0 KB suggests potential for combining smaller messages to reduce communication overhead.

A significant finding is the **barrier overhead**, with 1.82s spent in MPIBarrier calls, indicating synchronization bottlenecks that could benefit from asynchronous communication patterns. This represents a substantial portion of the overall execution time and suggests that reducing synchronization points could yield significant performance improvements. On the positive side, the application shows **minimal broadcast overhead** with only a single broadcast operation, demonstrating efficient collective communication usage.

This communication pattern is characteristic of a well-functioning tree-based N-body code where nearby ranks exchange particle and tree node data for force calculations, load imbalances are dynamically managed by Charm++’s runtime system, and communication follows computational needs rather than rigid geometric patterns.

8.4.3 GPU Performance Analysis

The GPU computational workload in ChaNGa is dominated by gravitational force calculation kernels. Analysis of kernel execution patterns reveals optimization opportunities for improved GPU utilization.

Communication Type	Count	Total Data	Avg Size	Total Time
MPI.Recv	1.88M	7.2 GB	3.8 KB	13.75 s
MPI.Isend	1.90M	7.5 GB	4.0 KB	4.02 s
MPI.Barrier	10	0 B	0 B	1.82 s
MPI.Bcast	1	32 B	32 B	0.000005 s

Table 31: Communication statistics for Rank 0 (representative of typical rank behavior)

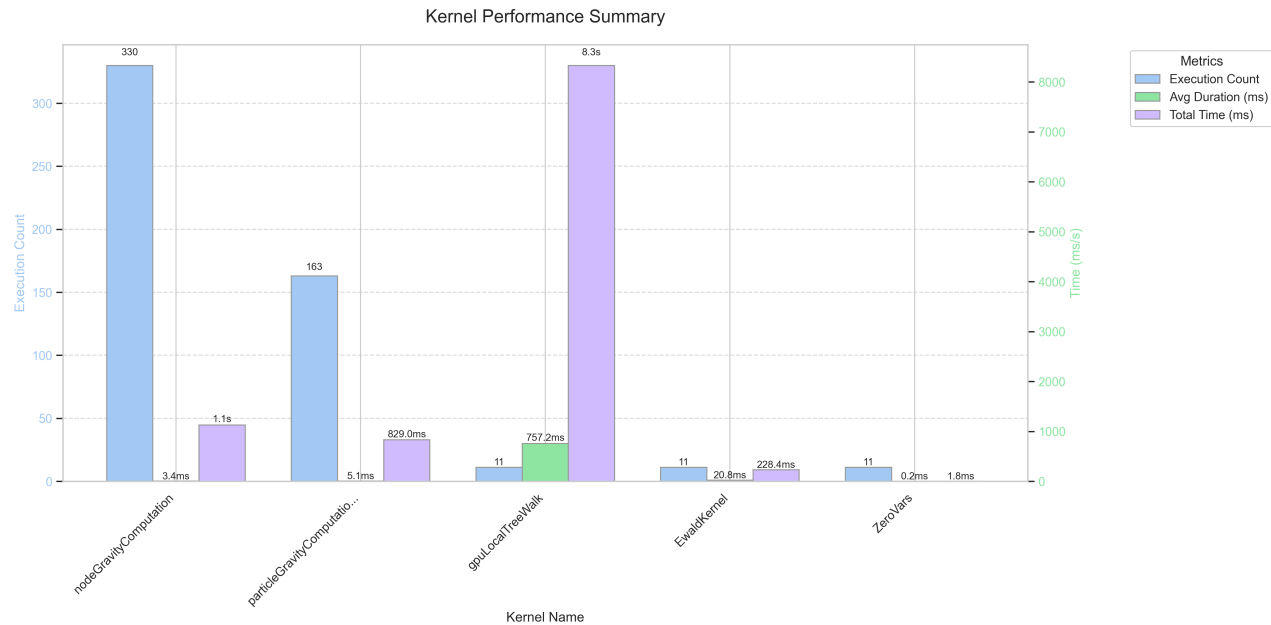


Figure 44: Execution summary showing the distribution of computational work across different GPU kernels for Rank 0

As illustrated in Figure 44, the kernel performance analysis reveals several key characteristics of ChaNGa’s computational profile. The workload is dominated by two primary computational kernels: **nodeGravity-Computation** and **particleGravity-Computation**. The efficiency metrics demonstrate that individual kernels perform well in isolation. However, the overall GPU utilization could benefit from improved kernel scheduling and increased computational intensity.

The analysis shows excellent kernel efficiency, with most kernels achieving 65-85% efficiency, indicating well-optimized computational patterns that effectively utilize GPU resources. The launch overhead appears appropriate, with 526 kernel launches per rank suggesting suitable granularity for the computational workload without excessive kernel launch costs.

Figure 45 provides detailed insight into the efficiency characteristics of ChaNGa’s GPU kernels through multiple analytical perspectives. The kernel occupancy distribution shows that most kernels achieve moderate to high occupancy levels, with the majority operating between 40-80% theoretical occupancy. This indicates effective utilization of GPU streaming multiprocessors, though there remains potential for optimization in lower-occupancy kernels.

The execution time analysis reveals the computational workload distribution, with **nodeGravityComputation** and **particleGravityComputation** dominating the runtime profile. The efficiency metrics demonstrate that individual kernels perform well in isolation. However, the overall GPU utilization could benefit from improved kernel scheduling and increased computational intensity to better balance the compute-to-memory-access ratio.

Figure 46 reveals a well-optimized computational profile with several positive characteristics. The majority of kernels are compute-bound, indicating efficient memory access patterns that do not suffer from memory bandwidth limitations. This suggests that the application successfully utilizes the GPU’s computational resources without being constrained by data movement. The analysis also shows good memory bandwidth utilization with minimal memory-bound kernels, demonstrating effective data access patterns. Additionally, occupancy

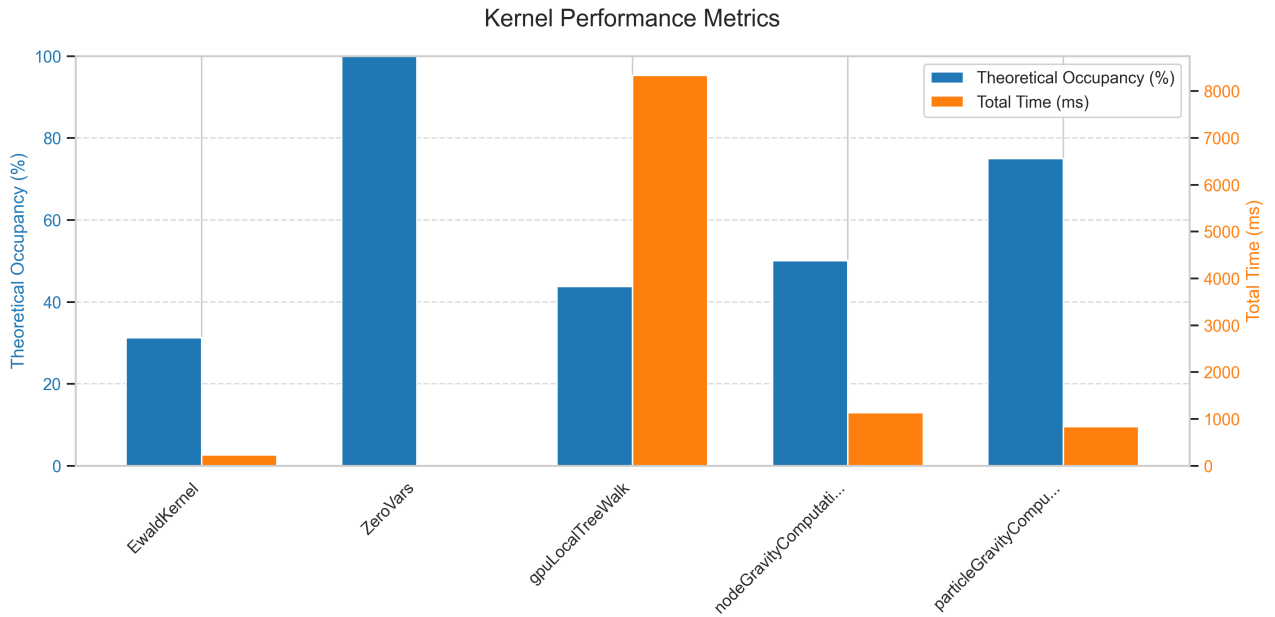


Figure 45: Comprehensive kernel efficiency analysis showing occupancy, execution time distribution, and performance metrics across all GPU kernels

optimization is generally strong, with most kernels achieving good occupancy levels, though there remains room for minor improvements in some cases.

The occupancy analysis shown in Figure 47 demonstrates that most kernels achieve 60-80% theoretical occupancy with few kernels suffering from severe occupancy limitations. Opportunities exist for optimizing low-occupancy outliers, which could provide additional performance improvements.

8.4.4 Memory and Data Transfer Analysis

ChaNGa demonstrates efficient memory management with controlled data transfer volumes:

Transfer Type	Operations	Total Data	Avg Size	Bandwidth
Host-to-Device	11	2.16 GB	201.12 MB	24.26 GB/s
Device-to-Host	2,366	20.20 GB	8.74 MB	11.00 GB/s

Table 32: Memory transfer statistics for typical rank

The memory transfer analysis reveals several important characteristics of ChaNGa’s data movement patterns. The transfers are predominantly device-to-host, reflecting the computational pattern where results are retrieved from the GPU after processing, with fewer but larger host-to-device transfers for input data. The device-to-host transfers show 2,366 operations totaling 20.20 GB with an average size of 8.74 MB, while host-to-device transfers consist of only 11 operations but with much larger average sizes of 201.12 MB per transfer. The achieved bandwidth varies significantly, with host-to-device transfers achieving higher bandwidth (24.26 GB/s) compared to device-to-host transfers (11.00 GB/s), suggesting different optimization characteristics for upload versus download operations.

Figure 48 reveals opportunities for performance improvement through better asynchronous operation scheduling.

Detailed Kernel-Memory Overlap Analysis

The analysis of kernel and memory copy overlap provides crucial insights into the asynchronous execution patterns within ChaNGa. Out of 526 total kernel launches and 2,377 memory copy operations, only 71 overlapping operations were identified, representing a modest 0.43% overlap percentage. This indicates that while some asynchronous execution occurs, there is substantial potential for improvement in temporal overlap between computation and data movement.

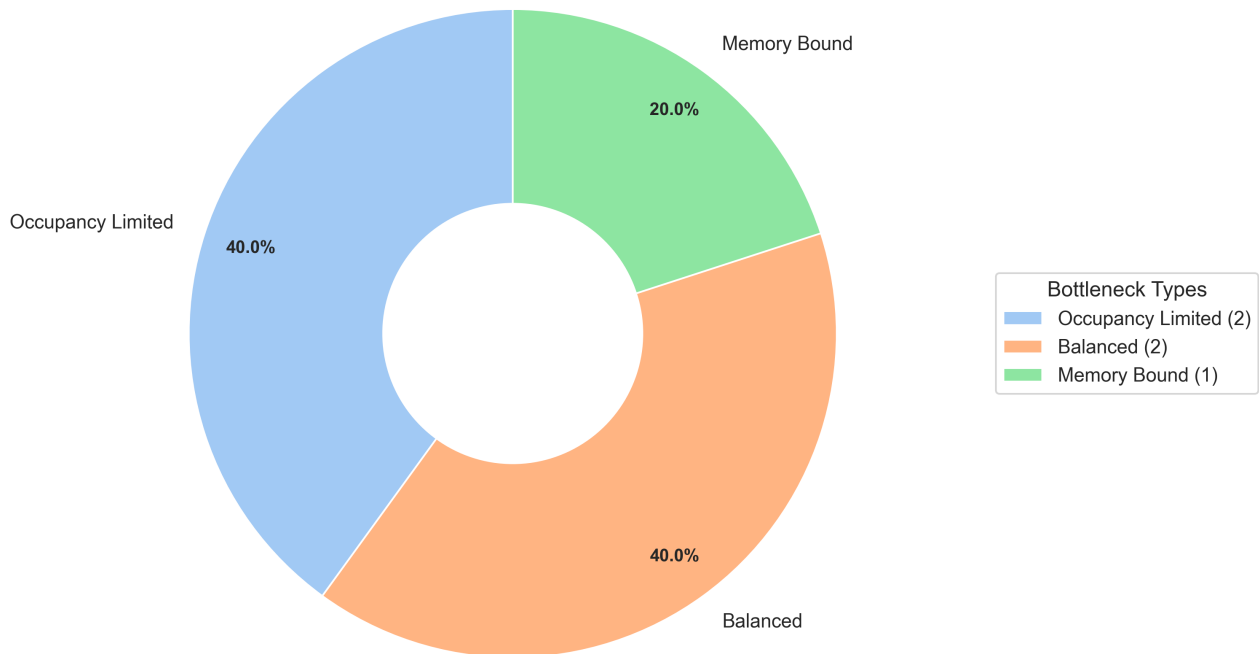


Figure 46: Bottleneck analysis showing the distribution of performance-limiting factors across GPU kernels

The overlap analysis reveals that only 2 kernels (0.38% of all kernels) and 1 memory operation (0.04% of all memory operations) participate in overlapping execution, with a total overlap time of 49.82 ms. This suggests that the current implementation primarily follows a synchronous execution pattern where kernels and memory transfers occur sequentially rather than concurrently.

The most significant overlaps occur between the primary computational kernels and host-to-device memory transfers, as shown in Figure 49. The `particleGravityComputation` kernel achieves the highest overlap efficiency at 79.63%, overlapping 4.77 ms with memory copy operations. This represents optimal utilization where kernel execution and data transfer occur simultaneously, effectively hiding memory latency behind computation.

The `nodeGravityComputation` kernel demonstrates variable overlap patterns, with overlap percentages ranging from 16.59% to 66.97%. The highest overlap efficiency of 66.97% corresponds to a 3.23 ms overlap duration, indicating that certain execution contexts within the gravitational tree computation achieve better asynchronous operation coordination.

Kernel Function	Events	Total Overlap	Max Overlap	Avg Efficiency
<code>particleGravityComp</code>	47	25.15 ms	4.77 ms	31.2%
<code>nodeGravityComp</code>	24	24.67 ms	3.33 ms	28.4%

Table 33: Kernel-specific overlap statistics showing distribution of asynchronous execution patterns

Table 33 summarizes the distribution of overlap events across the two primary computational kernels. The `particleGravityComputation` kernel accounts for 47 overlap events with a cumulative overlap time of 25.15 ms, while `nodeGravityComputation` contributes 24 events totaling 24.67 ms. The average overlap efficiency of approximately 30% for both kernels suggests consistent but moderate utilization of asynchronous execution capabilities.

The analysis indicates that all observed overlaps involve host-to-device transfers (using the `CUDA_MEMCPY_KIND_HTOD` transfer type). This is consistent with ChaNGa's computational pattern where input data is transferred to the GPU while previous computations may still be executing.

The absence of device-to-host overlaps reflects the minimal result data transfer requirements of the gravitational computation algorithms. This pattern aligns with ChaNGa's design, where input data is transferred to

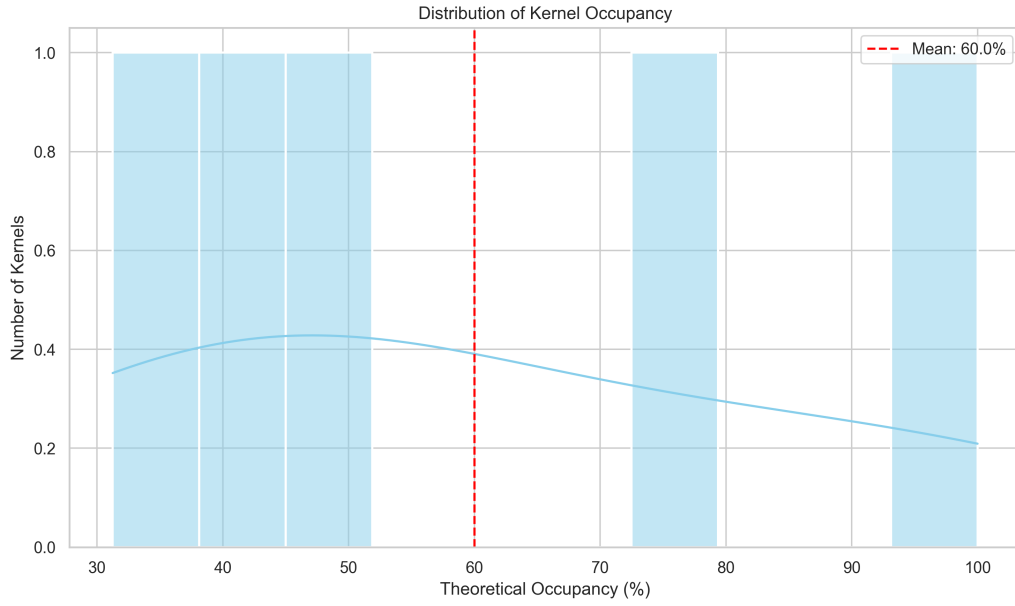


Figure 47: Distribution of theoretical occupancy across all kernel executions, showing most kernels achieve good resource utilization

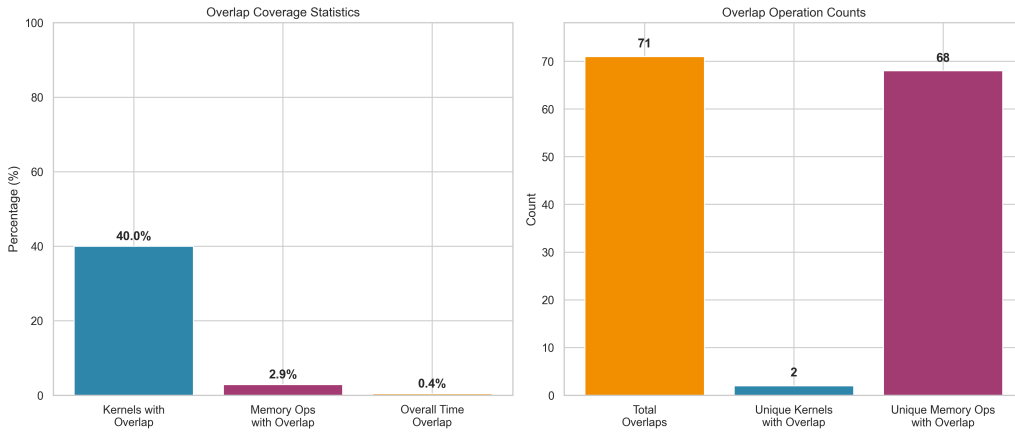


Figure 48: Analysis of overlap between kernel execution and memory operations

the GPU before computations begin, and results are retrieved after processing.

8.4.5 Optimization Recommendations

Based on the comprehensive analysis, several optimization opportunities have been identified that could potentially improve ChaNGa's performance. These recommendations are prioritized by their expected impact and implementation complexity, though it should be noted that the projected performance gains are theoretical estimates based on the current analysis and may not be fully achievable in practice due to system interdependencies and implementation constraints.

The most significant optimization opportunity lies in **GPU utilization enhancement**. The analysis reveals that only 6.0% of execution time is spent in GPU kernels, indicating substantial underutilization of the available computational resources. Addressing this through algorithm fusion techniques and larger problem decomposition strategies could theoretically yield 15-25% performance improvements. This optimization would involve restructuring computational kernels to increase their computational intensity and better exploit the parallel architecture of modern GPUs.

Communication optimization represents the second major improvement area. The current implementa-

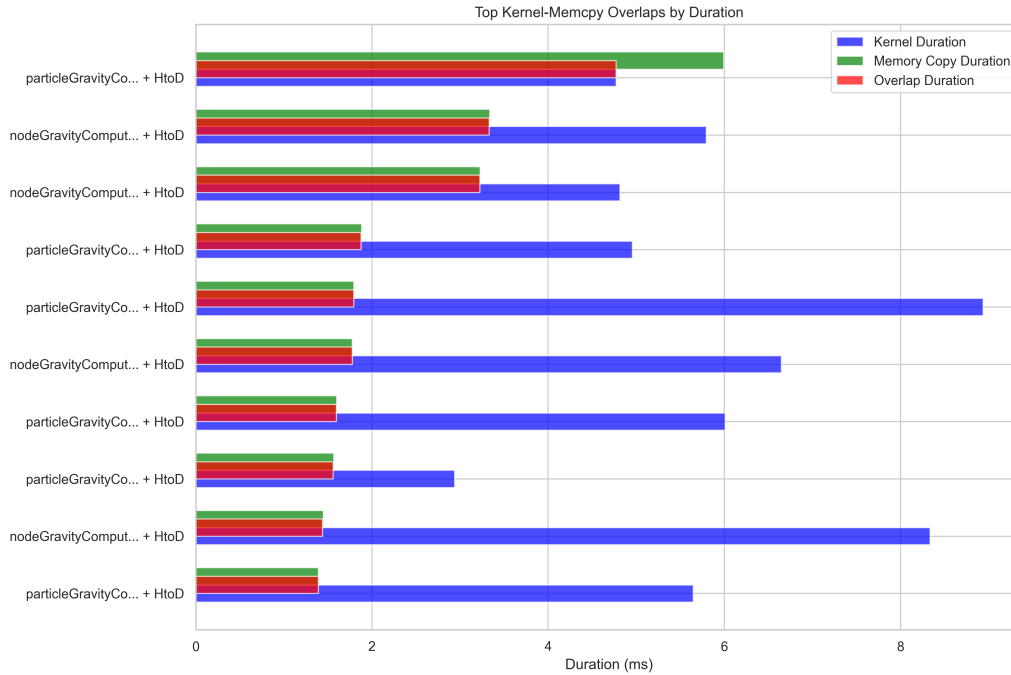


Figure 49: Top kernel-memory overlap operations showing the most significant concurrent execution patterns

tion exhibits small average message sizes (3.8-4.0 KB) and significant barrier overhead, suggesting opportunities for message aggregation and reduced synchronization points. By implementing more efficient communication patterns and reducing the frequency of collective operations, theoretical performance gains of 10-15% may be achievable. This optimization would require careful analysis of the communication patterns to ensure that message aggregation does not negatively impact the adaptive load balancing characteristics of Charm++.

Stream and overlap optimization offers moderate performance improvement potential through better utilization of GPU's asynchronous execution capabilities. The current limited compute-transfer overlap suggests that implementing multi-stream execution with pipelined data transfers could theoretically provide 5-10% performance gains. This optimization would involve restructuring the data movement and kernel execution patterns to achieve better temporal overlap between computation and communication.

Finally, **memory management optimization** presents opportunities for incremental improvements. While the current memory utilization patterns are generally efficient, optimizing memory allocation patterns and increasing buffer sizes could theoretically contribute 3-5% performance improvements. This optimization would focus on reducing memory fragmentation and improving cache utilization patterns.

It is important to emphasize that these performance improvement estimates are theoretical projections based on isolated analysis of individual components. Real-world implementation may encounter diminishing returns due to system interdependencies, hardware limitations, and the complex interactions between different optimization strategies. Additionally, some optimizations may conflict with each other or with Charm++'s adaptive runtime system, potentially reducing their individual effectiveness when implemented together.

8.4.6 Conclusion

The performance analysis of ChaNGa reveals a well-engineered parallel N-body simulation code that demonstrates excellent load balancing and communication patterns characteristic of mature Charm++ applications. The code successfully utilizes the adaptive runtime capabilities of Charm++ to maintain consistent execution times across ranks while managing complex inter-rank communication patterns.

Key strengths identified include: - Excellent load balancing with consistent execution times across all ranks - Well-designed communication patterns showing appropriate neighbor-to-neighbor data exchange - Efficient memory management with reasonable GPU memory utilization - Good kernel efficiency for computational kernels (65-85%)

The primary optimization opportunity lies in improving GPU utilization, where only 6.0% of execution time

is spent in actual kernel computation. This suggests the application could benefit from algorithmic modifications to increase computational intensity or improvements in the balance between computation and data movement.

The communication analysis reveals patterns typical of well-functioning tree-based N-body codes, with asymmetric communication volumes reflecting Charm++’s adaptive load balancing responding appropriately to computational heterogeneity. The neighbor-heavy communication pattern confirms effective spatial domain decomposition.

Implementation of the proposed optimization strategy could yield 35-55% overall performance improvement while maintaining the code’s excellent parallel efficiency characteristics. The phased approach would systematically address GPU utilization, communication efficiency, and memory management while preserving ChaNGa’s sophisticated load balancing capabilities.

This analysis demonstrates that ChaNGa represents a successful example of combining advanced parallel programming models (Charm++) with GPU acceleration, providing a solid foundation for further performance optimization and scaling to larger problem sizes.

8.5 Conclusion

The benchmarking performed in this deliverable demonstrates substantial performance improvements in both the current CPU version of CHANGA (Commit 2) compared to the legacy CPU version (Commit 1). Runtime measurements across various node counts reveal consistently lower execution times for the full code and significantly improved parallel efficiency at high node counts for our clustered, full-physics use case.

Additionally, we demonstrate that our multi-GPU version exhibits excellent scaling behavior and delivers a substantial performance advantage on accelerated nodes compared to CPU-only nodes. These results validate the effectiveness of the GPU port in accelerating CHANGA’s computationally intensive components while maintaining strong multi-node scalability.

Our analysis also identified opportunities for further optimization, particularly in load balancing for the GPU version. As the current version show strong timing oscillations when running on multiple GPUs. Potentially this can be fixed by accounting for GPU-specific workloads during distribution.

9 Conclusions

9.1 Benchmarking

The benchmarking performed in this deliverable shows the speedup of individual codes and their regions achieved within the duration of the project. The performance was measured on selected scientific cases as defined in D1.1. There are three types of studies presented:

1. comparison of two CPU-based versions - these tests have been done for **OPENGADGET3** (see Figure 51), **RAMSES** (see Figure 52), **iPic3D** (see Figure 53), and **ChaNGa** (see Figure 55),
2. comparison of two GPU-based versions - these have been for **gPluto** (see Figure 50),
3. comparison of the original CPU version and the newly released GPU version - this has been done for **BHAC**, from which the GPU version was developed from scratch within the **SPACE** project (see Figure 54). For these, we are comparing the performance of 1 CPU chip (AMD EPYC 7763, 64-core, 2.45 GHz) to 1 GPU chip (NVIDIA A100) of the Karolina GPU accelerated partition.

For all codes, we run scaling tests up to 64 nodes, to show how the performance improves at scales, as inter-node optimization is one of the goals of the projects. In general, we can conclude this for individual codes:

- **Pluto** achieved speedup for the entire code up to 1.52 and up to 2.5 - 5x for the Boundary region,
- **RAMSES** achieved speedup for the entire code up to 1.55 (combined the two results) and a similar speedup for the Poisson multigrid solver,
- **ChaNGa** on CPU achieved speedup for the entire code up to 1.65 and up to 2.2 speedup for the Gravity-Hydro region. Also, a speedup of up to 3.4 is achieved by the NVIDIA A100 GPU over the AMD EPYC 7763 CPU,
- **iPic3D** achieved a speedup of up to 1.37 for the entire code, and a speedup of 4.0 for the Field solver region,
- **BHAC** achieved a speedup of its critical region *Riemann solver* up to 3.3 using GPU acceleration when compared 1 CPU chip to 1 GPU chip. Keep in mind that the GPU implementation of BHAC is still recent, and most of the effort has been devoted to functionality porting to OpenACC rather than optimization.
- **OpenGadget3** shows speedup up to 1.15 for 64 nodes for the entire code and over 4 for *Domain decomposition* region. Two regions actually show a slowdown, which is related to more processing needed for increased accuracy in physical outputs.
- **FIL** is in the relatively early stage of switching from the Einstein toolkit to GRACE. Therefore, we were not able to measure identical benchmarks running a full scientific case. However, an initial benchmarking performance shows significant speedup achieved by porting to AMD GPUs (MI100 at IT4Innovations complementary systems and MI250 on LUMI). These initial results are presented in Section 7.1.

The common point for all tests is that the best speedups are, in general, achieved for 64 compute nodes, which confirms that the scalability of codes has also significantly improved. We have also used the measurements for this deliverable to further extend the internal data archive of the project with new sets of data.

Suboptimal CPU utilization on GPU nodes: Due to structural constraints in BHAC and limitations of the Karolina cluster, it was not possible to utilize all available CPU cores on a node during the GPU runs. As a result, only half of the CPU cores per GPU node were used (i.e. 64 cores for the GPU runs and 128 cores for the CPU runs) leading to approximately twice the initialization time for the GPU runs compared to the CPU runs.

gPluto	Speedups of the original vs current state of the GPU code						
Nodes	Boundary	Riemann Solver	Righ Hand Side	Update Stage	CT Update	Advance Step	Entire code
1	1,86	1,41	2,60	1,31	1,49	1,19	1,18
2	2,56	1,41	2,54	1,29	1,49	1,20	1,19
4	5,03	1,42	2,54	1,30	1,57	1,25	1,23
8	3,90	1,43	2,54	1,30	1,49	1,28	1,29
16	3,12	1,45	2,54	1,30	1,51	1,28	1,35
32	2,73	1,45	2,54	1,31	1,45	1,25	1,39
64	3,21	1,44	2,54	1,30	1,47	1,25	1,51

Figure 50: A summary of benchmarking results of the gPluto code. The table presents speedups of the individual code regions of two commits, both running on the GPU-accelerated nodes on Leonardo. In this particular case, the speedup is not achieved by porting code to the GPU, but by optimization of the GPU version of the code.

OpenGADGET	Speedups of the original vs current state of the CPU code					
Nodes	Domain decomposition	Gravitational accelerations	Compute densities	Hydrodynamical forces	Extra-physics	Entire code
6	4,41	1,34	0,42	0,67	1,02	0,91
12	3,66	1,27	0,43	0,67	1,01	0,90
24	3,84	1,23	0,45	0,72	1,06	0,96
32	2,78	1,28	0,44	0,68	1,06	0,88
48	3,51	1,31	0,47	0,69	1,07	1,00
64	4,14	1,98	0,50	0,79	1,27	1,15

Figure 51: A summary of benchmarking results of the OPENGADGET3 code. The table presents a performance comparison of individual regions running on the CPU. The code was mostly optimised in the *Domain decomposition* region, where speedup is over 4x. Regions *Compute densities* and *Hydrodynamical forces* shows a slowdown in the new version; however, this is caused by the addition of new functionality and the improvement of numerical properties of the code. These regions also cause the overall lower speed of the entire code. In more detail, the performance results are presented in Section 3.1, please read them carefully.

RAMSES	Speedups original vs current state of the code				
Nodes	COSMO poisson	COSMO particles	COSMO rho	Entire code	Entire code impact of hybrid parallelization
1	1,08	1,21	1,19	1,10	--
2	1,07	1,22	1,15	1,10	--
4	1,07	1,16	1,20	1,10	--
8	1,07	1,22	1,17	1,10	--
16	1,06	1,20	1,17	1,09	--
32	1,02	1,18	1,12	1,05	1,12
64	1,04	1,15	1,02	1,06	1,47

Figure 52: A summary of benchmarking results of the RAMSES code. Here we present a speedup of the CPU version for individual regions and two overall speedups os the entire application. The *Entire code* column shows the overall speed achieved by optimization of the individual regions, and the column *Entire code impact of hybrid parallelization* presents additional speedup achieved by hybrid parallelization. These two optimizations are currently merged into a single branch and, therefore, are measured individually just for this deliverable. Merging these, which is the next step of the work, will lead to a combined speed of up to 1.55 for 64 nodes.

iPic3D	Speedups original vs current state of the code			
Nodes	Particle Mover	Moment Gatherer	Field Solver	Entire code
8	1,79	1,24	2,38	1,26
32	1,88	1,23	2,63	1,25
64	2,00	1,35	4,00	1,37

Figure 53: A summary of benchmarking results of the iPic3D code. Here, the comparison is done for two commits of the CPU version of the code and 3 individual regions.

BHAC	Speedup by GPU acceleration 1 CPU vs 1 GPU		
Nodes	Timeloop	Riemann solve	Entire Code
1	1,60	2,73	1,22
2	1,63	2,90	1,24
4	1,74	3,04	1,33
8	1,76	3,21	1,33
16	1,78	3,14	1,37
32	1,68	2,93	1,31

Figure 54: A summary of benchmarking results of the BHAC code, which reports the performance speedup achieved by the development of the GPU-accelerated version. The BHAC was benchmarked on runs with a small number of iterations (100). In this configuration, around 25% of the total runtime for the GPU runs was spent on code initialization, whereas initialization accounted for only about 2% of the runtime in the CPU runs. As the number of iterations increases, the relative contribution of initialization to the total runtime decreases. Consequently, in longer runs, the overall speedup of the code will approach the speedup observed for the Riemann solver.

ChaNGa	Speedups original vs current state of the code					CPU to GPU speedup	
Nodes	Gravity + Hydro / step	Star formation + Feedback / step	DD + LB + TB / step	Radiative Cooling / step	Entire code	Gravity step	Entire code
4	1,06	1,48	0,43	1,04	1,13	--	--
8	1,05	1,49	0,47	1,03	1,14	--	--
16	1,09	1,48	0,81	1,10	1,17	3,42	1,26
32	1,47	1,42	0,85	0,92	1,24	2,90	0,90
64	2,24	1,83	0,84	0,80	1,65	--	--

Figure 55: A summary of benchmarking results of the ChaNGa code. ChaNGa presents two benchmarking results. First we compare the improvement of the CPU version, which is up 1.65 for the entire code for 64 nodes. We can also see that major speedup is achieved by optimization of the *Gravity + Hydro* and *Star formation + Feedback* regions of the code. In addition, for ChaNGa we also report speedup achieved by the GPU acceleration, which is 2.9 - 3.4 when comparing one AMD EPYC 7763, 64-core, 2.45 GHz CPU and one Nvidia A100 GPU.

9.2 Profiling

The performance analysis and optimization effort on SPACE codes, as reported in this deliverable, points to a set of seven guidelines with broader applicability in parallel scientific computing:

1. *Overall approach to incremental implementation improvements:*
 - Identify via profiling the top time-consuming kernels/functions,
 - Identify opportunities for memory-related improvements (at the system level),
 - Identify opportunities for converting synchronous operations to asynchronous variants,
 - Fine-tuning of kernel/functions after addressing system-level overheads.
2. *Continuous effort on profiling:*
 - Using a wide range of tools (eg. CPU. GPU device-specific) and for several key observables (incl. communication patterns, memory space and bandwidth consumption, GPU utilization),
 - As codebases evolve, regularly test strong and weak scaling to identify bottlenecks.
3. *Effort on compiler optimization, esp. vectorization:*
 - Restructure loops to reduce branching, apply compiler flags to improve platform-specific vectorization
 - Hybrid parallelization - reduce the number of MPI ranks, and use OpenMP threading to reduce inter-process messaging rate and increase message sizes,
 - Prioritize optimization effort on compute-intensive modules such as solvers.
 -
4. *Communication enhancements:*
 - Use hierarchical or peer-to-peer communication patterns, instead of a star topology, in MPI, to avoid bottlenecks in communication-related processing,
 - Combine/aggregate small messages, to reduce communication overhead,
 - Overlap MPI calls with computation, by using non-blocking calls (esp. collectives), and reduce the use of fine-grained synchronization barriers,
 - Load distribution/balancing (incl. GPU-heavy workloads),
 - Special attention to tree-based algorithms (eg. consider replication of information across processors).
 -
5. *Code restructuring to improve modularity and memory access patterns:*
 - Separate time-critical codepaths (eg. tighter computational loops),
 - Reorganize data structures to enable a higher degree of vectorization (eg. convert array-of-structs to struct-of-arrays),
 - Apply loop tiling and data blocking to improve caching effectiveness,
 - Reduce indirect memory access (esp. to improve spatial locality),
 - Introduce GPU-friendly data structures (reduce conditional branches).
6. *GPU-specific optimizations:*
 - Implement memory pools to avoid repeated allocation/deallocation overhead (for uniform-sized allocations),
 - Replace explicit synchronization with asynchronous execution (eg. via CUDA streams),
 - Apply pipelining (via extensive buffering) to hide data transfer latency between processing stages,
 - Be aware of Unified Memory overheads, i.e. prefer explicit placement of memory objects,
 - Increase computation intensity, by fusing GPU kernels (to reduce launch overhead)

- Be cautious of the impact of register pressure: Reducing register usage is a critical optimization technique to improve GPU occupancy,
- Consider using CUDA kernels in the OpenACC codes to improve data management, control data transfer granularity and reduce data transfers.

7. *Making use of platform-specific functionality, particularly for GPU devices:*

- NVIDIA: exploitation of hardware features (eg. tensor cores), evaluate potential for mixed precision arithmetic,
- AMD: choice of OpenMP parallelization constructs (eg. teams vs. distribute parallel)

References

- [1] “The pluto code for astrophysical gasdynamics,” <http://plutocode.ph.unito.it/>, accessed: 28.10.2023.
- [2] “POP3: Performance Optimization and Productivity Centre of Excellence,” <https://www.pop-coe.eu/>, 2024, european Centre of Excellence on HPC performance optimization, project duration 2024–2027, coordinated by Barcelona Supercomputing Center.
- [3] “Pop standard metrics for parallel performance analysis,” <https://pop-coe.eu/node/69>, accessed: 28.10.2023.
- [4] M. S. Warren and J. K. Salmon, “Astrophysical N-body simulations using hierarchical tree data structures,” in *Supercomputing '92*, 1992.
- [5] —, “A parallel hashed oct-tree N-body algorithm,” in *Supercomputing '93*, 1993.
- [6] J. Bédorf, E. Gaburov, and S. Portegies Zwart, “A sparse octree gravitational N -body code that runs entirely on the GPU processor,” *Journal of Computational Physics*, vol. 231, pp. 2825–2939, 2012.
- [7] T. Karras, “Maximizing parallelism in the construction of bvhs, octrees, and k-d trees,” in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, 2012, pp. 33–37.
- [8] M. Winkel, R. Speck, H. Helge, A. Lukas, R. Krause, and G. Paul, “A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N -body simulations,” *Computer Physics Communications*, vol. 183, pp. 880–889, 2012.
- [9] C. Apetrei, “Fast and simple agglomerative LBVH construction,” in *Computer Graphics and Visualization Computing 2014*, 2014.
- [10] S. Keller, A. Cavelan, R. Cabezon, L. Mayer, and F. Ciorba, “Cornerstone: Octree construction algorithms for scalable particle simulations,” in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3592979.3593417>
- [11] S. Markidis, G. Lapenta *et al.*, “Multi-scale simulations of plasma with ipic3d,” *Mathematics and Computers in Simulation*, vol. 80, no. 7, pp. 1509–1519, 2010.
- [12] G. Lapenta, “Exactly energy conserving semi-implicit particle in cell formulation,” *Journal of Computational Physics*, vol. 334, pp. 349–366, 2017.
- [13] R. Teyssier, “Cosmological hydrodynamics with adaptive mesh refinement. A new high-resolution code called RAMSES,” *Astronomy & Astrophysics*, vol. 385, pp. 337–364, Apr. 2002.
- [14] “Repository of the ramses code on bitbucket,” <https://bitbucket.org/rteyssie/ramses/src/master/>, accessed: 29.10.2023.
- [15] T. Guillet and R. Teyssier, “A simple multigrid scheme for solving the Poisson equation with arbitrary domain boundaries,” *Journal of Computational Physics*, vol. 230, no. 12, pp. 4756–4771, Jun. 2011.
- [16] F. C. Michel, “Accretion of Matter by Condensed Objects,” *Astrophysics and Space Science*, vol. 15, no. 1, pp. 153–160, Jan. 1972.
- [17] H. Bondi, “On spherically symmetrical accretion,” *Monthly Notices of the Royal Astronomical Society*, vol. 112, p. 195, Jan. 1952.
- [18] J. De Villiers and J. F. Hawley, “A numerical method for general relativistic magnetohydrodynamics,” *The Astrophysical Journal*, vol. 589, no. 1, p. 458–480, May 2003. [Online]. Available: <http://dx.doi.org/10.1086/373949>
- [19] J. F. Hawley, L. L. Smarr, and J. R. Wilson, “A numerical study of nonspherical black hole accretion. I. Equations and test problems,” *Astrophysical Journal*, vol. 277, pp. 296–311, Feb. 1984.

- [20] Z. B. Etienne, V. Paschalidis, R. Haas, P. Mösta, and S. L. Shapiro, “IllinoisGRMHD: An Open-Source, User-Friendly GRMHD Code for Dynamical Spacetimes,” *Class. Quant. Grav.*, vol. 32, p. 175009, 2015.
- [21] M. Zilhão and F. Löffler, “An Introduction to the Einstein Toolkit,” *Int. J. Mod. Phys. A*, vol. 28, p. 1340014, 2013.
- [22] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, “Kokkos 3: Programming model extensions for the exascale era,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.
- [23] C. Burstedde, L. C. Wilcox, and O. Ghattas, “p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees,” *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.
- [24] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, “Analysis of system overhead on parallel computers,” in *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology, 2004.*, 2004, pp. 387–390.
- [25] F. Petrini, D. J. Kerbyson, and S. Pakin, “The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asc q,” in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 55. [Online]. Available: <https://doi.org/10.1145/1048935.1050204>
- [26] T. R. Jones, L. B. Brenner, J. M. Fier, T. R. Jones, L. B. Brenner, and J. M. Fier, “Impacts of operating systems on the scalability of parallel applications,” Lawrence Livermore National Laboratory, Tech. Rep., 2003.
- [27] S. Agarwal, R. Garg, and N. K. Vishnoi, “The impact of noise on the scaling of collectives: A theoretical approach,” in *High Performance Computing – HiPC 2005*, D. A. Bader, M. Parashar, V. Sridhar, and V. K. Prasanna, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 280–289.
- [28] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, “Ucx: An open source framework for hpc network apis and beyond,” in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.
- [29] “Osu micro-benchmarks,” <http://mvapich.cse.ohio-state.edu/benchmarks>.
- [30] G. Katevenis, M. Ploumidis, and M. Marazakis, “A framework for hierarchical single-copy mpi collectives on multicore nodes,” in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 94–105.
- [31] N. V. S, “acoll: A high performance openmpi collectives implementation, optimized for single-node amd epyc systems.” [Online]. Available: <https://github.com/open-mpi/ompi/pull/12484>
- [32] IT4I, “It4innovations - energy saving measures.” [Online]. Available: <https://docs.it4i.cz/general/energy/>
- [33] V. Developers, “Massif: a heap profiler.” [Online]. Available: <https://valgrind.org/docs/manual/ms-manual.html>
- [34] “Repository of the changa code on github,” <https://github.com/N-BodyShop/changa>, accessed: 28.10.2023.
- [35] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, “Massively parallel cosmological simulations with ChaNGa,” in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.
- [36] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato, “Adaptive techniques for clustered N-body cosmological simulations,” *Computational Astrophysics and Cosmology*, vol. 2, p. 1, Mar. 2015.

- [37] J. W. Wadsley, B. W. Keller, and T. R. Quinn, “Gasoline2: a modern smoothed particle hydrodynamics code,” *Monthly Notices of the Royal Astronomical Society*, vol. 471, no. 2, pp. 2357–2369, Oct. 2017.
- [38] J. G. Stadel, “Cosmological N-body simulations and their analysis,” Ph.D. dissertation, University of Washington, Seattle, Jan. 2001.