Master in High Performance Computing

Performance Analysis and GPU Scalability of OGSTM-BFM

Supervisors:
Stefano Campanella,
Giorgio Bolzon

Internal Supervisor:
Ivan Girotto

Candidate:
André Feitosa Benevides

Academic Year 2025–2026

Abstract: OGSTM-BFM is a coupled physical-biogeochemical model developed at the National Institute of Oceanography and Applied Geophysics (OGS) [1, 2, 3, 4] and is used for climate-related studies. Recent work within ESiWACE3 (Centre of Excellence in Simulation of Weather and Climate in Europe) reported substantial performance gains after porting the model to GPU architectures [5].

This thesis presents a reproducibility study of selected GPU performance results of OGSTM-BFM on the Leonardo supercomputer, as well as further investigations that were not included in previous ESiWACE3 reports. A significant acceleration can be achieved when using GPUs, provided that appropriate MPI rank-to-GPU mappings and Multi-Process Service (MPS) configurations are employed. Several configurations were tested. When we consider the GPU version running on two nodes, with four NVIDIA A100 GPUs per node and four MPI ranks per GPU, compared to two dual-socket nodes with Intel Sapphire Rapids CPUs, this leads to a speedup of 1.64. Once MPS is enabled, performance increases dramatically (speedup of 5.72). Different mappings of the ranks to the GPUs were tested. It was found that round-robin mapping combined with 50% MPS (each rank limited to $\sim 50\%$ GPU threads), further increases the speedup to 5.97. Lastly, after adding NUMA binding in the MPS launch path, we managed to achieve a speedup of 6.22. We note that a speedup of 7.41 was reported on the ESiWACE3 Technical Report [5]; however, in this study, we were not able to actually reach this result.

In addition, this work evaluates an alternative implementation of the vertical diffusion tridiagonal solver using NVIDIA's cuSPARSE batched routines. The original implementation solved a system with a tridiagonal matrix by a method that does not allow for full parallelism. The algorithm, however, is highly specialized to tridiagonal matrices. We developed benchmark tests both in isolation and integrated into OGSTM-BFM. The cuSPARSE-based variant in isolation is about 3.33 times slower than the specialized version. When integrated into OGSTM-BFM it leads to 8% slower runs than the baseline in the tested configuration.

These results emphasize that increased parallelism alone does not guarantee improved time-to-solution; they also show that launch-level tuning (MPS, mapping, NUMA placement) is as important as kernel-level optimization. The thesis provides practical insights into GPU reproducibility, solver integration, and performance engineering for large-scale scientific applications.

Contents

Chapter 1

Introduction

1.1 The OGSTM-BFM Model

OGSTM-BFM is a Fortran application, partially ported to GPU using OpenACC for Nvidia GPUs, developed at the National Institute of Oceanography and Applied Geophysics (OGS). It simulates a coupled physical–biogeochemical marine ecosystem model. It combines:

- The OGS Tracer Model - OGSTM [2], responsible for physical transport processes,
- The Biogeochemical Flux Model - BFM [6], responsible for nonlinear biological and chemical interactions.

The model solves a partial differential equation modeling the transport of substances due to movement (advection), spreading (diffusion), and production/consumption (reaction). More precisely, it solves advection-diffusion-reaction equations for more than fifty tracers over a three-dimensional domain with over one hundred vertical levels.

OGSTM-BFM is used operationally for Mediterranean Sea forecasting and for long-term climate studies, making time-to-solution a critical factor for scientific productivity.

Within the ESiWACE3 project, the model was ported to GPU architectures using OpenACC directives, and substantial speedups were reported [5]. One of the goals of this work is validation of the reported results and further analysis.

1.2 Objectives of This Thesis

This thesis pursues two complementary objectives.

The first objective is to validate selected GPU performance results of OGSTM-BFM on the Leonardo supercomputer. Reproducibility is essential in high-performance computing, where performance depends strongly on hardware architecture, runtime configuration, and decomposition strategy. Particular attention is given to MPI rank-to-GPU mapping and the use of NVIDIA Multi-Process Service (MPS).

The second objective is to investigate the performance of the vertical diffusion solver in the routine `trczdf`. This routine solves a large number of tridiagonal linear systems at each timestep and represents a significant portion of the GPU runtime. An alternative implementation based on NVIDIA's cuSPARSE batched tridiagonal solver is developed and evaluated against the specialized baseline solver.

1.3 Research Questions

The work presented in this thesis addresses the following research questions:

1. Can we reproduce reported GPU performance results of OGSTM-BFM under controlled conditions?
2. How do MPI decomposition and rank-to-GPU mapping affect performance and runtime stability?
3. Does replacing the specialized tridiagonal solver with a cuSPARSE-based batched solver improve overall performance?
4. What insights can be drawn regarding algorithmic specialization and GPU library integration in large-scale scientific codes?

Chapter 2

The OGSTM-BFM Model System

2.1 Overview of the Coupled Model

OGSTM-BFM is a coupled physical-biogeochemical marine ecosystem model developed at the National Institute of Oceanography and Applied Geophysics (OGS). The system combines:

- **OGSTM (OGS Transport Model)**, responsible for the physical transport of tracers.
- **BFM (Biogeochemical Flux Model)**, responsible for nonlinear biological and chemical reaction processes.

The coupled system numerically solves the temporal evolution of multiple marine tracers, representing physical, chemical, and biological state variables. These tracers evolve according to advection-diffusion-reaction equations over a three-dimensional spatial domain.

The OGSTM model is an open-source code available on GitHub[1]. Meanwhile, the biogeochemical module based on the Biogeochemical Flux Model (BFM) is managed by the BFM consortium[2]. For the Mediterranean Sea application, model formulations and evaluations are documented in a set of BFM and ecosystem studies [1, 3, 4, 7].

The code is written in Fortran90 and parallelized using MPI. The domain decomposition is performed along the longitudinal and latitudinal directions. The reaction term BFM is called as an external F90 library.

In its operational configuration for the Mediterranean Sea, the model uses:

- A horizontal grid of 1085 (longitude) by 380 (latitude) points,
- 125 vertical levels,
- 51 biogeochemical tracers.

[1] https://github.com/inogs/ogstm
[2] https://www.bfm-community.eu/bfm-consortium/

This results in tens of millions of degrees of freedom per simulation timestep, making OGSTM-BFM a computationally demanding application well suited for high-performance computing systems.

In operational production on CINECA systems, a typical run uses 10 nodes with 48 CPUs each, yielding a per-MPI-task subdomain of approximately 125x35x35 compute points.

2.2 Governing Equations

For each tracer $c_i(x,t)$ (representing the biogeochemical concentration of the $i-th$ state variable), the governing equation can be expressed in the general form:

$$\frac{\partial c_i}{\partial t} = -\mathbf{U}\cdot\nabla c_i + \mathcal{D}_h(c_i) + \frac{\partial}{\partial z}\left(k_v\frac{\partial c_i}{\partial z}\right) + w_{s,i}\frac{\partial c_i}{\partial z} + R_i(c_1,\ldots,c_N), \tag{2.1}$$

where:

- $\mathbf{U}$ represents the sea water velocity field (advection),
- $\mathcal{D}_h$ represents horizontal (bi-Laplacian) diffusion,
- The vertical eddy diffusivity models vertical mixing, a fundamental process for the dynamics of marine ecosystems,
- $w_{s,i}$ represents the living plankton and detrital matter sinking velocity,
- R_i denotes nonlinear biological reactions computed by BFM.

The equation contains both explicitly integrated transport terms and an implicitly solved vertical diffusion term. The latter requires the solution of linear systems along vertical columns and plays a key role in the computational structure of the model.

2.3 Numerical Splitting and Time Integration

OGSTM-BFM adopts a source-splitting approach in which physical transport and biological reactions are integrated sequentially within each timestep.

At a high level, each timestep consists of:

1. Advection (Smolarkiewicz scheme),
2. Horizontal diffusion,
3. Vertical diffusion (implicit solve),
4. Biological reaction terms (BFM),
5. Boundary conditions and surface fluxes.

Advection, horizontal diffusion, and biological transformations are integrated using explicit time stepping. In contrast, the vertical diffusion operator is treated implicitly to ensure numerical stability for fine vertical resolution and strong mixing conditions.

The implicit vertical diffusion step requires solving a tridiagonal linear system for each vertical column and tracer. This leads to a large number of small, independent linear systems that must be solved at every timestep.

2.4 Parallelization Strategy

Since domain decomposition is performed in the horizontal dimensions (longitude and latitude), the vertical dimension remains local to each MPI rank.

- Each MPI rank handles a three-dimensional subdomain (representing a water column).
- Halo exchanges are required between neighboring ranks for horizontal operators.
- MPI communication accounts for a significant fraction of runtime.

When executed on GPU-enabled systems, depending on how coarse-grained the domain decomposition is, we may have multiple MPI ranks sharing a single GPU. This can improve occupancy, but it also introduces additional complexity in terms of resource sharing and synchronization.

Because the vertical dimension is not decomposed, each rank must independently solve all vertical tridiagonal systems within its subdomain. This design simplifies communication but influences solver parallelization strategies.

2.5 Computational Hotspots

Profiling results show that the main computational hotspots of OGSTM-BFM include:

- The advection routine (`trcadv`),
- The horizontal diffusion routine (`trchdf`),
- The vertical diffusion routine (`trczdf`),
- Several BFM biological kernels, including carbonate system and plankton dynamics.

Among these, the vertical diffusion step is particularly interesting from a numerical and architectural perspective. While advection and reaction terms exhibit substantial loop-level parallelism, the implicit vertical solver operates on relatively small tridiagonal systems (one per vertical column per tracer).

The original algorithm used to invert the tridiagonal matrices is exclusively designed for this type of matrices, however it cannot be fully parallelized. This motivates the investigation of alternative linear solver implementations presented later in this work.

2.6 The Vertical Tridiagonal Systems

The vertical diffusion term leads to linear systems of the form:

$$A\mathbf{x} = \mathbf{b}, \tag{2.2}$$

where A is tridiagonal and corresponds to the discretization of the second-order vertical diffusion operator.

For each vertical column and tracer, a separate system must be solved. Given:

- N_z vertical levels (~ 125),
- N_t tracers (~ 50),
- Multiple horizontal grid points per MPI rank,

the total number of tridiagonal solves per timestep is large.

The currently implemented solver uses the Thomas algorithm, a specialized, highly efficient form of Gaussian elimination used exclusively for tridiagonal systems. It reduces computational cost from $\mathcal{O}(n^3)$ to $\mathcal{O}(n)$.

However, the Thomas algorithm exhibits limited parallelism, which can restrict GPU efficiency. This raises the question of whether replacing it with a GPU-oriented batched sparse solver could improve performance, a question explored in Chapter 5.

Chapter 3

Computational Environment and Experimental Setup

3.1 Target HPC System

All experiments presented in this work were performed on the *Leonardo* supercomputer hosted at CINECA. In particular, the GPU-enabled *Booster* module was used for all accelerator-based benchmarks.

Each Booster node (BullSequana X2135 "Da Vinci") consists of:

- 1 Intel Xeon 8358 CPU (32 cores, 2.6 GHz),
- 512 GB DDR4 3200 MHz memory,
- 4 NVIDIA A100 GPUs (64 GB HBM2 memory each),
- 2 NVIDIA HDR 2x100 Gb/s network cards,
- Peak performance of 89.4 TFLOPs per node.

The presence of four GPUs per node allows multiple MPI ranks to share a single accelerator. This configuration is relevant for performance tuning, as GPU occupancy and kernel concurrency depend on the number of ranks mapped to each device.

The CPU-only reference runs were performed on Leonardo Data Centric nodes (BullSequana X2140). Each node consists of:

- 2 Intel Sapphire Rapids CPUs (56 cores each, 4.8 GHz),
- 512 GB DDR5 4800 MHz memory,
- 3 NVIDIA HDR 1x100 Gb/s network cards,
- 8 TB NVM.

When simulating the Mediterranean Sea, at least two nodes (eight GPUs total) are required to fit the full OGSTM-BFM workload on GPUs.

3.2 Software Stack

Two distinct toolchains were employed:

- **CPU Toolchain:**
  - Intel oneAPI compilers (ifort),
  - Intel MPI.
- **GPU Toolchain:**
  - NVIDIA HPC SDK (nvhpc),
  - OpenMPI,
  - CUDA Toolkit 12.x,
  - OpenACC directives for GPU offloading.

The GPU implementation of OGSTM-BFM uses OpenACC. Data regions are defined using unstructured directives (manual control when data is created, copied, and deleted) to keep arrays resident on the GPU throughout the execution phase, minimizing host-device transfers.

The ESiWACE3 deployment used the toolchain versions summarized in Table 3.1 [5].

Table 3.1: Software stack versions used in the ESiWACE3 deployment.

Component	Version	Toolchain
cmake	3.27.7	ifort, nvhpc
hdf5	1.14.3	ifort, nvhpc
parallel-netcdf	1.12.3	ifort, nvhpc
netcdf-c	4.9.2	ifort, nvhpc
netcdf-fortran	4.6.1	ifort, nvhpc
intel-oneapi-compilers	2023.2.1	ifort
intel-oneapi-mpi	2021.10.0	ifort
nvhpc	23.11	nvhpc
openmpi	4.1.6	nvhpc
cuda	12.1	nvhpc

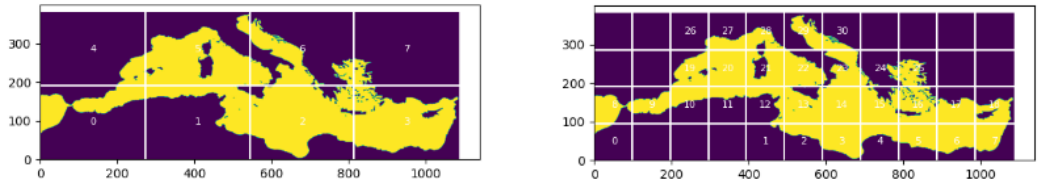
All runs were compiled with optimization flags enabled, and no debugging instrumentation was active during performance measurements.

3.3 Domain Decomposition and MPI Configuration

OGSTM-BFM employs horizontal domain decomposition using MPI. The spatial domain is partitioned in longitude and latitude, while the vertical dimension remains local to each MPI rank. Halo exchanges are required between neighboring ranks.

Even though OGSTM-BFM can be applied to arbitrary domains, we are interested in running and benchmarking the code using the Mediterranean Sea grid mesh. Once the domain is divided in smaller cells, not every cell contains water points. Therefore we often need to adjust to the nearest valid configuration.

Figure 3.1 illustrates two representative decompositions for 8 and 31 MPI rank counts.



(a) 8 ranks (1 per GPU)

(b) 31 ranks (4 per GPU)

Figure 3.1: Horizontal domain decomposition examples for two-node runs.

Notice that the coarser decomposition contains valid water points in all cells. On the other hand the finer decomposition has only 31 valid cells. To run this layout, we allocate 16 tasks per node and use only 31 ranks.

3.4 Multi-Process Service (MPS)

When multiple MPI ranks share a single GPU, NVIDIA Multi-Process Service (MPS) was enabled to allow concurrent kernel execution and better context sharing. In the main job file, each rank runs a wrapper script which performs three tasks:

1. Start a single MPS control daemon per node (only by local rank 0) with job-specific directories.
2. Bind each local rank to one logical GPU according to the target occupancy.
3. Launch the application binary (*ogstm.xx*) with the selected GPU exported through CUDA/OpenACC variables.

Activation is done by running the mps daemon in the background:

```
nvidia-cuda-mps-control -d
```

We also set the mapping of the ranks to the GPUs on the same script.

Benchmarks were performed both with MPS enabled and disabled in order to evaluate the effect of kernel concurrency and the impact of GPU context switching. Surprisingly, disabling MPS in configurations with four MPI ranks per GPU resulted in drastic performance degradation. We obtained $5.72x$ slowdown for this configuration when MPS is disabled.

3.5 Benchmark Methodology

To ensure reproducibility and comparability with previously reported results, the following benchmarking procedure was adopted:

- All measurements exclude the first timestep to avoid initialization overhead.
- I/O dumping was disabled in all reported runs.
- Wall-clock time was measured for a fixed three-day simulation window.
- Each experiment was repeated to confirm stability and consistency.

I/O-enabled measurements were not included in this thesis because enabling output triggered runtime errors in the current I/O library on the tested software stack.

For solver comparisons, a first standalone program was developed. We used identical numerical configurations to ensure that differences in performance were attributable to algorithmic changes only. Afterwards, the cuSPARSE library used in the external program was integrated into the vertical tracer component. Its use was enabled by a compiler flag.

3.6 Runtime Stability

During reproduction experiments, certain MPI configurations triggered GPU `cudaErrorIllegalAddr` runtime failures localized to the `pelagiccsys` routine.

These failures were reproducible for specific domain decompositions (notably 8 and 23 total MPI ranks), independent of the rank-to-GPU mappings. Other configurations completed successfully.

Given that stable configurations were available and reproducible, performance comparisons were conducted only on successful execution cases. The observed error cases are reported in Chapter 4.

Chapter 4

Reproduction and Extension of GPU Performance Results

4.1 Reproduction Methodology

This chapter presents the reproduction of selected GPU performance results for OGSTM-BFM on the Leonardo supercomputer. The objective is to validate previously reported scalability trends under controlled and reproducible conditions.

All benchmarks were performed using:

- A three-day simulation window,
- Two Leonardo Booster nodes (eight GPUs total),
- I/O dumping disabled

4.2 Multi-GPU Performance Results

Table 4.1 summarizes the benchmark results for different MPI rank configurations. Configurations that resulted in runtime failures (illegal GPU memory access) are reported but excluded from performance analysis.

The best GPU performance was obtained using 31 MPI ranks (four ranks per GPU) with round-robin mapping and MPS thread limit at 50%, combined with NUMA binding, achieving a total runtime of 337.4 seconds.

4.2.1 Effect of MPI Rank Density

Increasing the number of MPI ranks per GPU improves occupancy and reduces idle time, provided that kernel concurrency is enabled via NVIDIA Multi-Process Service (MPS).

Disabling MPS in the 31-rank configuration resulted in an average runtime of 1276 seconds, corresponding to a performance degradation of approximately a factor of 3.5. This comparison is summarized in Figure 4.1.

Table 4.1: Three-day simulation on two nodes (I/O disabled).

MPI Ranks (Total)	Ranks per GPU	MPS / Policy	Time (s)
8	1	Yes	Error
16	2	Yes	438
23	3	Yes	Error
31	4	Yes (default launch)	366.0
31	4	Yes (rr, 50%)	351.3
31	4	Yes (rr, 50% + NUMA)	**337.4**
31	4	No	1276
31	CPU-only	No	2097

4.2.2 Impact of MPS in Multi-Rank GPU Execution

For the same 31-rank decomposition, enabling MPS in this launch context reduces runtime from 1276 s to 366.0 s, i.e. a reduction of 910.0 s (about 71.5%). This is one major runtime effect.

This behavior is consistent with the execution model where several MPI processes share each GPU. Without MPS, work submitted from different processes incur substantial context-management overhead and reduced overlap, especially with four ranks per GPU. With MPS enabled, process-level GPU work submission is coordinated through the MPS server, which improves effective concurrency for this type of workload.

The present study does not include low-level CUDA context traces, so this interpretation is treated as a plausible mechanism rather than a formally isolated proof.

4.2.3 MPI Rank Mapping and MPS Thread Limit

We tested 31-rank runs with three mapping strategies and different MPS thread limits to isolate scheduler effects.

The mapping policies were:

1. **block**: ranks are grouped and assigned to one GPU via

$$\text{GPU} = \left\lfloor \frac{\text{LOCAL\_RANK}}{\text{PROCS\_PER\_GPU}} \right\rfloor$$

2. **rr**: ranks are distributed round-robin:

$$\text{GPU} = \text{LOCAL\_RANK} \bmod \text{GPUS\_PER\_NODE}$$

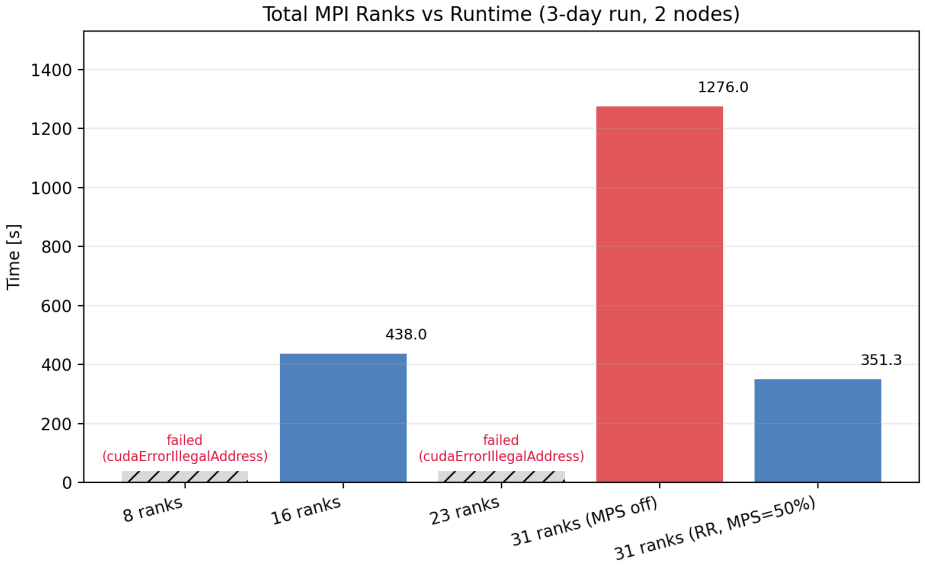


Figure 4.1: MPI rank sensitivity for two-node, three-day runs with known failures (8 and 23 ranks) and 31-rank comparison with and without MPS.

3. **block_shift**: block mapping with a circular shift:

$$\text{GPU} = \left(\left\lfloor \frac{\text{LOCAL\_RANK}}{\text{PROCS\_PER\_GPU}} \right\rfloor + \text{SHIFT} \right) \bmod \text{GPUS\_PER\_NODE}$$

For this workload the best configuration was round-robin mapping with four ranks per GPU and `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=50%` (351.3 s).

For block-based mappings, enabling 50% MPS was already effective; however, block-shift variations (shift 0–3) gave similar runtime values (376.3–378.0 s), indicating that the shift itself did not materially affect performance.

The same thread-limit sweep on round-robin mapping found that 50% performed best (351.3 s), with 75% and 100% both at 360.0 s. Compared with the previous best block+MPS baseline (384.3 s), this is about 9% faster.

4.2.4 NUMA Binding for MPS Client Launch

An additional refinement was to bind ranks to local NUMA domains when launching MPS clients. The idea is to make ranks that use a certain GPU run on the CPU NUMA node closest to that GPU, and allocate memory on that NUMA node.

With round-robin, 50% MPS and this affinity-aware launch path, runtime improved from 351.3 s to 337.4 s, i.e. a reduction of 13.9 s (about 3.95%).

4.2.5 Speedup Compared to CPU Execution

The CPU-only configuration required 2097 seconds for the same simulation window.

The speedup achieved by the optimal GPU configuration is therefore:

$$S = \frac{2097}{337.4} \approx 6.21.$$

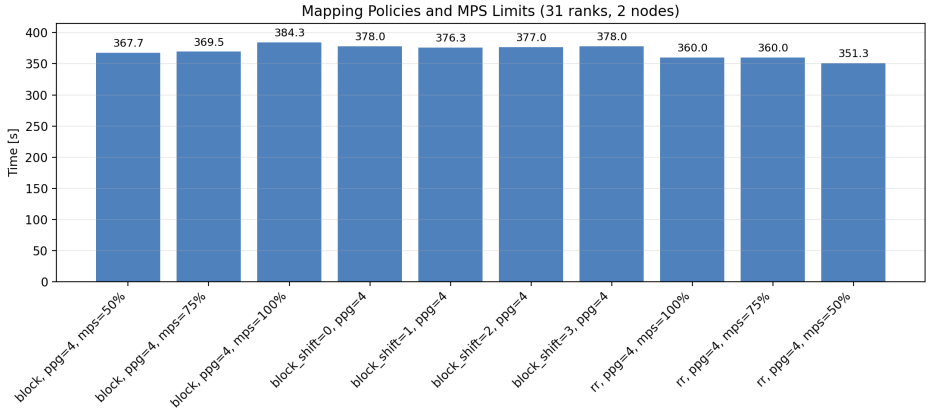


Figure 4.2: Impact of MPI rank-to-GPU mapping and MPS thread limit for the 31-rank setup.

4.2.6 Combined Ranking of 31-Rank Configurations

Table 4.1 and Figure 4.3 show the progression of improvements for 31-rank tests:

- No-MPS launch: 1276 s,
- MPS enabled (default launch): 366.0 s,
- RR + 50% MPS: 351.3 s,
- RR + 50% MPS + NUMA binding: 337.4 s.

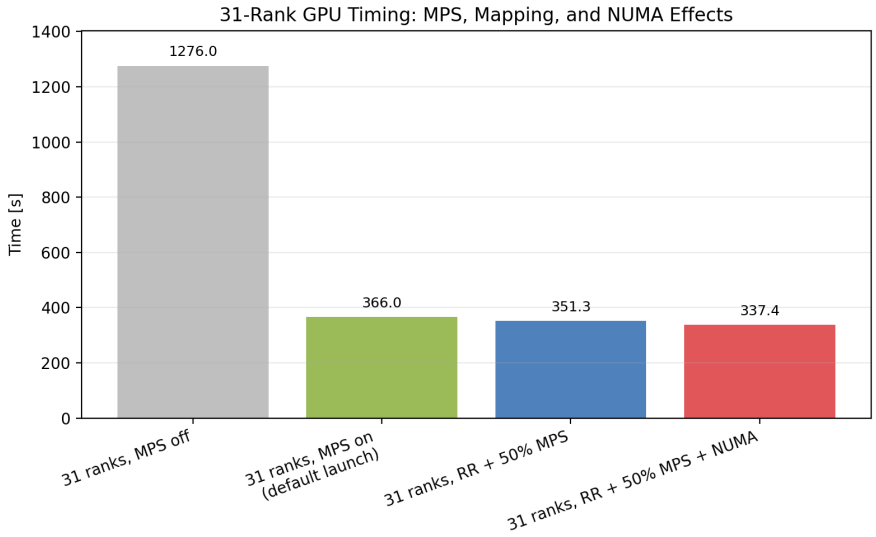


Figure 4.3: 31-rank timing progression: no-MPS, MPS default launch, RR with 50% MPS, and RR with 50% MPS plus NUMA binding.

This result confirms substantial acceleration when using GPUs.

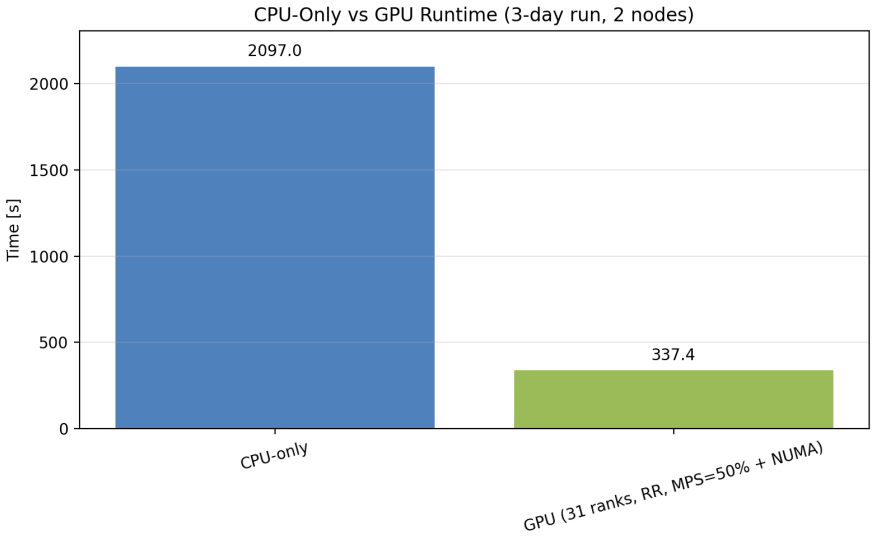


Figure 4.4: CPU-only versus GPU runtime for the baseline stable configuration.

4.2.7 Reference Results Reported in ESiWACE3

The ESiWACE3 technical report identifies the best two-node, no-I/O configuration as 31 MPI ranks (four ranks per GPU, with one GPU hosting three ranks due to decomposition constraints) [5]. The report also shows a speedup of approximately 7.41 for a three-day simulation when comparing two GPU nodes against two CPU nodes, and reports a performance degradation of about a factor of five when MPS is disabled in the four-ranks-per-GPU configuration. Unfortunately no further details are explicitly provided on how this speed-up was achieved.

In this thesis, the same best rank layout is used and combined with further improvements, with a final speedup of 6.21 and MPS-off penalty 3.78x relative to no-MPS at 31 ranks. There is still a significant difference between the previously reported speedup and the value we were able to obtain. However without access to the details of the report, it is not clear what the root cause for this discrepancy is.

4.3 Configuration-Dependent GPU Failures

The configurations using 8 and 23 MPI ranks consistently produced GPU runtime failures of the form:

- `cudaErrorIllegalAddress`,
- Failure during `cuStreamSynchronize`,
- Error localized to the `pelagiccsys` routine.

For the tested software stack and decomposition settings, these failures were reproducible for those rank counts, while other configurations completed successfully. Root-cause analysis is outside the scope of this work.

4.4 Runtime Breakdown Analysis

To better understand performance characteristics, profiling data was collected for the main routines contributing to total runtime.

Table 4.2 reports the elapsed time (in seconds) of selected routines for three representative MPI ranks (minimum, median, maximum communication load).

Table 4.2: Runtime contribution (seconds) of major routines.

Routine	**Rank A**	**Rank B**	**Rank C**
bfm_PhytoDynamics	22.81	37.79	33.48
forcing_phys	5.61	18.68	12.11
trcadv	66.29	84.80	77.33
trcbio	84.70	121.93	113.66
trcsms	86.64	124.54	115.91
trcstp	275.23	294.06	287.42
trczdf	24.94	45.71	35.02
step_total	300.16	311.67	307.87

4.4.1 Observations

Several important trends can be identified: The vertical diffusion routine `trczdf` consumes a significantly larger fraction of runtime on GPU compared to CPU. Also, we observe noticeable imbalance in routine timing across MPI ranks.

Notably, `trczdf` accounts for a significant share of GPU execution time relative to its CPU contribution. This behavior is consistent with previous observation that the algorithm implemented here cannot be fully parallelized. This motivated the investigation of alternative solver strategies presented in Chapter 5.

Chapter 5

Alternative GPU Solver for `trczdf`: cuSPARSE Batched Tridiagonal Solve

5.1 Motivation

Profiling results discussed in Chapter 4 show that the vertical diffusion routine `trczdf` contributes a disproportionately large fraction of the total runtime on GPU compared to CPU[1]. This behavior is consistent with the algorithmic structure of the implicit vertical diffusion step: for each horizontal grid point and tracer, the model solves an independent tridiagonal linear system along the vertical dimension, with an algorithm that is sequential per column.

The baseline solver is highly specialized for tridiagonal matrices and provides excellent performance on CPUs. However, it raises the question of whether we could outperform the sequential algorithm by using the built-in NVIDIA matrix inversion library for the GPU. This chapter investigates whether a more GPU-oriented approach based on NVIDIA's cuSPARSE batched tridiagonal solvers can improve time-to-solution.

5.2 Baseline: Tridiagonal Solve in `trczdf`

In OGSTM-BFM, the implicit vertical diffusion step constructs the tridiagonal system:

$$A\mathbf{x} = \mathbf{b}, \tag{5.1}$$

where A is tridiagonal. In the code, the three diagonals are stored in the arrays:

- `zwi`: sub-diagonal,
- `zwd`: main diagonal,

[1] That does not mean it takes longer execution time on the GPU. It means that, relative to the total runtime, it represents a larger share on GPU than on CPU for the same benchmark.

- `zws`: super-diagonal,

and the right-hand side is stored in `zwy`. The solution is produced in `zwx`.

The existing implementation follows a Thomas-like forward elimination and backward substitution scheme. While the solve is performed for many independent columns (indexed by `jv`), each individual system contains loop-carried dependencies in the vertical index `jk`. As a result, parallelization is primarily available across columns, while the inner solve remains effectively sequential per column.

This characteristic makes `trczdf` a challenging kernel for GPU parallelization: the work per system is relatively small (on the order of the number of vertical levels), the algorithm for each column has dependencies that prevent parallelism.

5.3 cuSPARSE Batched Tridiagonal Solver

As an attempt to increase GPU utilization, an alternative solver based on cuSPARSE was implemented using the routine:

- `cusparseDgtsvInterleavedBatch`

This function solves a batch of independent tridiagonal systems in a single library call. The interleaved format expected by cuSPARSE stores the batch as a two-dimensional layout where each system is contiguous in the leading dimension. In this work, the arrays were stored as `(batch, m)` in Fortran, where:

- `m` corresponds to the system size (equal to `jpkm1`),
- `batch` corresponds to the number of vertical columns (`dimen_jvzdf`).

The cuSPARSE solver supports different internal algorithms. In the experiments presented later, the LU factorization with pivoting was selected.

5.3.1 CUDA Wrapper

cuSPARSE was called through a lightweight CUDA C++ wrapper that we implemented, exposing a C function callable from Fortran. The wrapper performs:

- cuSPARSE handle creation,
- temporary buffer size query,
- temporary buffer allocation,
- batched solve call,
- cleanup (buffer free and handle destruction).

This design minimizes integration complexity but introduces per-call overhead due to repeated handle creation and memory allocation. The performance implications of this are discussed in Chapter 6.

5.3.2 Fortran Interface and OpenACC Interoperability

The Fortran interface module (`gtsv_batched_interface.F90`) uses `iso_c_binding` and OpenACC interoperability to pass device pointers to the CUDA wrapper without explicit host-device transfers.

The key mechanism is:

- `$acc host_data use_device(...)`

which makes the device addresses of OpenACC-managed arrays available to the C function.

5.4 Integration into `trczdf`

The cuSPARSE solver was integrated into `trczdf` using conditional compilation. When `USE_ZDF_GPU` is enabled, the routine:

1. Packs the diagonals (`zwi, zwd, zws`) and RHS (`zwy`) into persistent interleaved buffers: `dl_ib, d_ib, du_ib, x_ib`.

2. Enforces boundary constraints required by the library:

$$dl(:,1) = 0, \quad du(:,m) = 0.$$

3. Calls the batched solver on the GPU.

4. Unpacks the solution from `x_ib` into `zwx` to preserve compatibility with the downstream code.

This approach allows the remainder of `trczdf` to remain unchanged.

5.4.1 Data Residency

To avoid repeated GPU allocations, the interleaved buffers are allocated only once and persist across calls. They are created with OpenACC data directives so they remain resident on device memory for the duration of the run.

5.4.2 Packing and Unpacking Overhead

Unlike the baseline implementation, which operates directly on the solver arrays, the cuSPARSE implementation requires explicit packing into the interleaved layout and unpacking of the solution. Although these operations are parallelizable and performed on the GPU, they introduce additional memory traffic that may impact performance.

5.5 Correctness Validation with a Standalone Benchmark

To validate the cuSPARSE integration independently of the full OGSTM-BFM workflow, a standalone benchmark was developed. The benchmark generates batches of diagonally-dominant tridiagonal systems with a known solution $\mathbf{x}_{true}$, computes the corresponding right-hand side:

$$\mathbf{b} = A\mathbf{x}_{true},$$

and then solves the system using:

- cuSPARSE `DgtsvInterleavedBatch`,
- a custom GPU Thomas implementation (`one thread per system`).

The computed solution $\hat{\mathbf{x}}$ is compared to $\mathbf{x}_{true}$ using a relative ℓ_2 error:

$$\varepsilon = \frac{\|\hat{\mathbf{x}} - \mathbf{x}_{true}\|_2}{\|\mathbf{x}_{true}\|_2}.$$

This benchmark provides confidence that the cuSPARSE implementation produces correct solutions and that the OpenACC interoperability mechanism passes valid device pointers.

5.6 Summary

This chapter presented an alternative GPU implementation of the vertical diffusion tridiagonal solver based on cuSPARSE batched routines. The integration relies on persistent interleaved buffers and OpenACC `host_data` to avoid explicit data transfers. A standalone benchmark was developed to validate correctness and compare performance against a custom GPU Thomas solver.

The next chapter evaluates the impact of this change on the full OGSTM-BFM application, including runtime comparisons and analysis of the cuSPARSE-based approach performance.

Chapter 6

Results and Discussion

6.1 Overview

This chapter evaluates the impact of replacing the baseline tridiagonal solver in `trczdf` with a cuSPARSE-based batched solver. The analysis is organized around two questions:

1. Does the cuSPARSE batched approach improve time-to-solution for OGSTM-BFM?
2. If not, what factors explain the observed performance behavior?

Results are presented both integrated into OGSTM-BFM on Leonardo (two nodes, three-day simulation, no I/O), as well as using a standalone program to validate correctness and compare raw solver performance.

6.2 Application-Level Performance

6.2.1 Benchmark Setup

All application-level benchmarks in this chapter use the same setup as Chapter 4:

- Two GPU nodes (eight GPUs total),
- Three-day simulation window,
- I/O dumping disabled,

The baseline GPU configuration used for comparison is: 31 MPI ranks with MPS enabled (no additional rank mapping or NUMA bindings were used in this test).

6.2.2 Baseline vs cuSPARSE Performance

Table 6.1 summarizes the application-level results comparing the baseline tridiagonal solver and the cuSPARSE-based solver.

In the experiments performed during this thesis, replacing the specialized tridiagonal solver with the cuSPARSE batched solver increased time-to-solution from

Table 6.1: Application-level runtime comparison (three-day simulation, two nodes, I/O disabled).

Solver Implementation	**MPI Configuration**	**Time (s)**
Baseline tridiagonal solver (Thomas algorithm)	31 ranks, MPS on	366
cuSPARSE batched solver (`DgtsvInterleavedBatch`)	31 ranks, MPS on	380

366 s to 380 s (about 3.8%). While cuSPARSE offers greater theoretical parallelism via batched execution, the specialized solver outperforms it, at least for the tested problem size.

6.3 Solver-Level Microbenchmark Results

6.3.1 Purpose of the Microbenchmark

To isolate solver performance from the complexity of OGSTM-BFM, a standalone benchmark was implemented. It generates batches of diagonally-dominant tridiagonal systems with a known solution and compares correctness and runtime between the cuSPARSE batched solver and a custom GPU Thomas implementation (one CUDA thread per system).

This benchmark validates correctness independently and provides insight into raw solver throughput under controlled conditions.

6.3.2 Correctness

Both solvers reproduce the known solution with small relative error. The error metric used is the relative ℓ_2 norm:

$$\varepsilon = \frac{\|\hat{\mathbf{x}} - \mathbf{x}_{true}\|_2}{\|\mathbf{x}_{true}\|_2}.$$

The cuSPARSE integration and OpenACC interoperability were validated successfully.

6.3.3 Microbenchmark Timing

The benchmark measures solver runtime for fixed system size m and batch count. The results are reported as wall-clock time for the solve portion, excluding initialization.

For the tested case (`m=512`, `batch=8192`), the standalone run produced the following results:

In this isolated test, cuSPARSE alone is approximately 3.3x slower than the custom Thomas-GPU implementation while delivering comparable numerical accuracy. However, when integrated at application level the total slowdown is more modest in absolute terms (about 3.8%).

Table 6.2: Standalone GPU solver benchmark (`m=512`, `batch=8192`).

Method	**Time (s)**	**Relative error**
cuSPARSE `DgtsvInterleavedBatch`	0.020	0.134×10^{-15}
Custom Thomas-GPU	0.006	0.125×10^{-15}

6.3.4 Per-call Library Overhead

The baseline solver is specialized for the exact structure and boundary conditions of the vertical diffusion operator. Such specialization can reduce memory movement and exploit known properties of the system.

cuSPARSE provides a general-purpose batched solver designed for broad applicability. Even though it is highly optimized, its generality can introduce additional operations and memory access patterns that are not necessary for the OGSTM-BFM case.

In the current wrapper implementation, each call to the cuSPARSE solver performs:

- `cusparseCreate()` and `cusparseDestroy()`,
- Buffer size query,
- `cudaMalloc()` and `cudaFree()` for the temporary workspace buffer.

The cuSPARSE API expects interleaved input arrays for the diagonals and the right-hand side. In OGSTM-BFM, the baseline solver operates directly on solver arrays such as `zwi`, `zwd`, `zws`, and `zwy`. The cuSPARSE implementation requires:

- Packing diagonals and RHS into interleaved buffers,
- Enforcing boundary constraints in separate kernels,
- Unpacking the solution back into `zwx`.

Although these steps are parallelized on the GPU, they increase memory traffic and kernel launch count.

Because `trczdf` is invoked repeatedly (for each timestep and tracer), these operations may introduce overhead. In contrast, the baseline solver operates entirely within the existing OpenACC region without calling external libraries or allocating temporary buffers on each invocation.

A potential optimization would be to use a persistent cuSPARSE handle and persistent workspace allocation across calls, but this was not implemented in the current experimental version.

6.4 Implications for `trczdf` and Future Optimizations

The results indicate that increasing solver parallelism through a generic batched library is not sufficient to improve application performance. For OGSTM-BFM, the key performance challenge in `trczdf` appears to be the balance between the inherent sequential structure of tridiagonal solves and overheads introduced by library integration.

Future work could explore alternative approaches such as:

- Persistent cuSPARSE handles and workspace buffers to reduce per-call overhead,
- Algorithmic alternatives to Thomas (e.g., cyclic reduction variants) that offer greater parallelism while remaining specialized,

6.5 Summary

This chapter evaluated a cuSPARSE-based batched tridiagonal solver as an alternative to the baseline specialized solver in `trczdf`. While correctness was validated and the approach offers higher theoretical GPU parallelism, the application-level benchmarks show that the cuSPARSE implementation increased time-to-solution (380 s versus 366 s). In an isolated solver benchmark, cuSPARSE was also slower (0.020 s versus 0.006 s for Thomas-GPU at `m=512`, `batch=8192`).

The observations are consistent with overhead from library calls and data-layout handling, but a dedicated ablation study would be required for definitive attribution. These findings reinforce a key lesson in HPC: greater parallelism alone does not guarantee improved performance.

In the next chapter, we conclude the thesis and summarize key lessons learned from reproducibility, benchmarking, and solver evaluation on modern GPU-based HPC systems.

Chapter 7

Conclusions

7.1 Summary of Contributions

This work investigated the GPU performance of the OGSTM-BFM coupled physical-biogeochemical model on the Leonardo supercomputer. The work was structured around two main objectives:

1. Reproducing and extending previously reported GPU performance results,
2. Evaluating an alternative GPU-based tridiagonal solver using cuSPARSE.

The reproduction study confirmed substantial speedup with GPUs when MPI rank-to-GPU mapping and NVIDIA Multi-Process Service (MPS) are configured appropriately. The best-performing tested configuration (31 MPI ranks across two nodes with MPS enabled and NUMA-aware launch) achieved 337.4 seconds versus 2097 seconds for the CPU-only reference, corresponding to a speedup of approximately 6.21. For the same 31-rank decomposition, disabling MPS increased runtime to 1276 seconds (3.78x slower), showing that MPS is a first-order performance factor for this workload.

Certain MPI configurations (8 and 23 ranks) consistently produced GPU runtime failures characterized by illegal memory access errors. These failures are reported as reproducible observations for the tested setup; root-causes are not clear.

The second objective focused on the vertical diffusion routine `trczdf`, identified as a potential GPU performance hotspot. A cuSPARSE-based batched tridiagonal solver was implemented and integrated using OpenACC interoperability mechanisms. A standalone benchmark was developed to validate correctness and compare solver-level performance against a custom GPU Thomas implementation.

Although the cuSPARSE solver has higher theoretical parallelism, application-level benchmarks showed a decrease in performance for the tested configuration (380 seconds versus 366 seconds for the baseline solver). A standalone solver benchmark (`m=512`, `batch=8192`) showed the same direction: Thomas-GPU took 0.006 s while cuSPARSE took 0.020 s, with comparable relative error.

7.2 Lessons Learned

Several practical lessons emerge from this work:

Reproducibility in HPC Is Non-trivial

Even with detailed technical documentation, reproducing performance results requires careful control of runtime configuration, MPI decomposition, GPU sharing strategy, and software environment. Although the best-performing configuration came close, it did not match the ESiWACE3 report peak speedup [5].

GPU Efficiency Depends on Algorithmic Structure

The vertical diffusion step involves many small tridiagonal systems with strong sequential dependencies. This limits fine-grained parallelism and makes performance sensitive to implementation details.

Solver substitutions that are attractive at kernel level may not improve full-application time-to-solution. End-to-end measurements remain the decisive criterion.

7.3 Final Remarks

This thesis documents a full reproducibility effort and a solver-integration study on a production HPC code. The results show clear GPU acceleration, practical constraints on valid decompositions, and the need for application-level validation of optimization ideas. These findings could provide concrete guidance for future OGSTM-BFM performance engineering.

This study also suggests further directions of exploration:

- Implementing persistent cuSPARSE handles and pre-allocated workspaces to reduce per-call overhead.
- Investigating alternative parallel tridiagonal algorithms (e.g., cyclic reduction variants) that preserve specialization while increasing concurrency.

Acknowledgements

I would like to thank the MHPC program and OGS, for the support, resources, and collaboration that made this thesis possible.

I am grateful to my supervisor Stefano Campanella, and to my internal supervisor Ivan Girotto, for their guidance throughout this work.

I also thank the my colleagues for the insightful discussions and constructive feedback during the courses and the development of this thesis.

Bibliography

[1] G. Cossarini, P. Lazzari, and C. Solidoro, "Spatiotemporal variability of alkalinity in the mediterranean sea," *Biogeosciences* **12** (Mar., 2015) 1647–1658.

[2] P. Lazzari, A. Teruzzi, S. Salon, S. Campagna, S. Calonaci, S. Colella, M. Tonani, and A. Crise, "Pre-operational short-term forecasts for mediterranean sea biogeochemistry," *Ocean Science* **6** (2010) 25.

[3] P. Lazzari, C. Solidoro, V. Ibello, S. Salon, A. Teruzzi, K. Béranger, S. Colella, and A. Crise, "Seasonal and inter-annual variability of plankton chlorophyll and primary production in the mediterranean sea: a modelling approach," *Biogeosciences* **9** (Jan., 2012) 217–233.

[4] P. Lazzari, C. Solidoro, S. Salon, and G. Bolzon, "Spatial variability of phosphate and nitrate in the mediterranean sea: A modeling approach," *Deep Sea Research Part I: Oceanographic Research Papers* **108** (Feb., 2016) 39–52.

[5] ESiWACE3, "Esiwace3 technical report - hpc project service on ogstm / bfm," technical report, ESiWACE3 / OGS, n.d. Internal report referenced for this thesis; local copy: `ESiWACE3_OGTSM_BFM_GPU.pdf`.

[6] M. Vichi, T. Lovato, P. Lazzari, G. Cossarini, E. Gutierrez-Mlot, G. Mattia, S. Masina, W. J. McKiver, N. Pinardi, C. Solidoro, L. Tedesco, and M. Zavatarelli, "The biogeochemical flux model (bfm): Equation description and user manual, bfm version 5.1," technical report, BFM Consortium, 2015.

[7] S. Salon, G. Cossarini, G. Bolzon, L. Feudale, P. Lazzari, A. Teruzzi, C. Solidoro, and A. Crise, "Novel metrics based on biogeochemical argo data to improve the model uncertainty evaluation of the cmems mediterranean marine ecosystem forecasts," *Ocean Science* **15** (Aug., 2019) 997–1022.