natESM

Sprint Documentation 16

PDAF2GPU sprint

Sergey Sukov[1] and Lars Nerger[2]

[1]Jülich Supercomputing Center, Forschungszentrum Jülich GmbH, Jülich, Germany
[2]Alfred-Wegener-Institut, Helmholtz-Zentrum für Polar- und Meeresforschung, Bremerhaven, Germany

Contact: info@nat-esm.de

Published on 10 July 2025 on https://www.nat-esm.de/services/accepted-sprints

1 Summary

The objective of the sprint was to address two technical challenges: evaluating and optimizing the performance of PDAF software modules, and developing a strategy for porting the code to GPUs. The key results achieved during the sprint are an evaluation of the parallel efficiency of a standalone application of the ICON-Ocean model with PDAF, code analysis, and the development of a strategy for GPU offloading. Preliminary GPU porting of core PDAF subroutines and subsequent performance testing were also carried out. However, due to the specific characteristics of the data assimilation algorithm, further restructuring and GPU-specific development are needed. As the PDAF developers currently lack sufficient GPU expertise, a follow-up sprint is recommended to continue this work.

2 General Information

Start and end date:	29.08.2024 - 31.3.2025
Intended period:	6 months
Responsible RSE:	Sergey Sukov (JSC)
Responsible scientist:	Lars Nerger (AWI)

The Parallel Data Assimilation Framework (PDAF) is a software framework to perform data assimilation, i.e. the quantitative combination of model information with observational data. PDAF provides model-agnostic data-assimilation functionality and is suited for application with complex high-dimensional models. Framework modules are coded in Fortran and support hybrid (MPI plus OpenMP) parallelization. PDAF can be directly coupled with a model, or it can be used as the base for a standalone parallel data assimilation program. It is combined with a model by implementing case-specific routines which are mainly called by PDAF as callback procedures that provide case-specific operations. The main purpose of PDAF is to perform ensemble data assimilation in which an ensemble of model states is used to estimate the model state and its uncertainty (as a covariance matrix or probability distribution). PDAF provides the possibility to

directly couple it into a model, so that the model is modified into an ensemble model with data assimilation functionality ("online coupling"). The alternative is "offline coupling" in which the data assimilation program is separate from the model program and data exchanges are performed through disk files. In this case the data assimilation program is called after running the model to integrate an ensemble of model states. It reads restart files of the model, computes the assimilation (analysis) step, and writes updated restart files. The online coupling leads to a more efficient program because model restarts are avoided. However, the direct coupling into a model is also more complex to implement than the file-based offline coupling.

PDAF consists of a software library plus tutorials and templates that help users to build a data assimilation system for the model they want to use. For optimal compatibility with models of Earth system components, which are commonly coded in Fortran, PDAF is also coded in Fortran and uses both MPI and OpenMP for parallelization. PDAF is designed to work with essentially any model and the developers at AWI, but also external users have coupled PDAF to a wide variety of models.

In preparation for the sprint, an offline-analysis program for the ICON-Ocean model (pdaf-icono program) was prepared to serve as a test case. This test case is based on that used by Pohlmann et al. (Climate Dynamics, 61(2023) 357–373, https://doi.org/10.1007/s00382-022-06558-w) and was updated to support the latest features of PDAF version 2.3. In this climate application, only temperature and salinity-model fields are considered, and profile observations of these variables are assimilated using the LESTKF method with an ensemble size of 40. The code supports both the MPI and OpenMP parallelization. The test case only needs a few minutes run time when using a full node on Levante. It has a sufficiently realistic complexity to assess the code performance and to develop a porting strategy to GPUs.

3 Sprint Objectives

The objective of the sprint was to address two technical problems:

- Performance evaluation and optimization of PDAF software modules using the offline-coupled (file-based) application provided for the sprint.
- Development of a strategy for offloading computations to GPUs and porting the code.

In the course of the work, the main focus was on the core subroutines of the PDAF library and the callback functions – that is, on modules used in both offline and online coupled simulations. At the same time, in accordance with the request for support, restrictions were imposed on the changes made to the code. Due to the large number of developer teams whose simulation codes use the data-assimilation framework, the structure of the PDAF code, the interface of its library subroutines, and the calling algorithm should be considered fixed.

4 Procedure and Insights

4.1 Technical Approach / Procedure

The actual sprint schedule included the following stages:

1. Evaluation of pdaf-icono parallel efficiency and PDAF-framework code-structure analysis.
2. Performance comparison between CPU and GPU implementations of the BLAS and LAPACK library subroutines using custom-designed benchmarks.
3. GPU porting of PDAF-core subroutines.

Parallel efficiency evaluation of pdaf-icono involved strong scaling tests on up to 4096 CPU cores and a runtime comparison of different launch configurations (pure MPI vs. hybrid MPI+OpenMP) on 128 CPU cores. The OpenMP parallel efficiency presented in the request for support was considerably underestimated because of incorrect launch settings. Therefore, a specially developed C/Fortran module was linked into the code to manage and configure OpenMP thread affinity to CPU cores. This solution was implemented to prevent similar issues in the future.

The PDAF subroutines included in the data analysis module make intensive use of BLAS and LAPACK library functions. Therefore, runs of simplified benchmarks preceded the code porting. The performance comparison of CPU and GPU implementations of linear algebra functions revealed that offloading computations to GPUs requires a heterogeneous execution model combining MPI, OpenMP, and OpenACC/CUDA with concurrent GPU kernel launches.

Porting and subsequent performance tests were carried out separately for the observation-search subroutine and the variable values-analysis subroutine. GPU executables were compiled using NVIDIA SDK 22.5, which includes GPU-accelerated linear-algebra-function libraries. Profiling was conducted using graphical data and reports generated by NVIDIA Nsight Systems tools.

While the code was being ported, efforts were also made to optimize it. The focus was on reducing the number of memory reallocation operations and avoiding repeated evaluations of boolean conditions.

Discussions of the sprint's technical details mainly took place through email correspondence and during weekly video conferences. One personal meeting occurred between the RSE and the responsible scientist during the natESM Community Workshop 2025 in Berlin.

4.2 General Insights

The pdaf-icono program operates on two datasets: grid nodes and observation points. Each dataset comprises arrays of coordinates and variable values. The assimilation algorithm evaluates and updates variables at grid nodes based on observation data. Since the processing of individual grid nodes is free from data dependencies, the computational loop over the nodes can be parallelized using a single OpenMP directive !$OMP PARALLEL DO, without requiring thread synchronization within the loop body. The most computationally intensive subroutines involved in processing a single grid node are “observation search” and “analysis.”

The observation-search subroutine generates a local list of observation points located within a specified radius around the node. The list is built through two complete passes over the observation dataset: the first pass counts the number of relevant observation points, while the second pass stores their indices in the list. Both passes call the same subroutine, PDAFomi_check_dist2_noniso_loop, but with different settings.

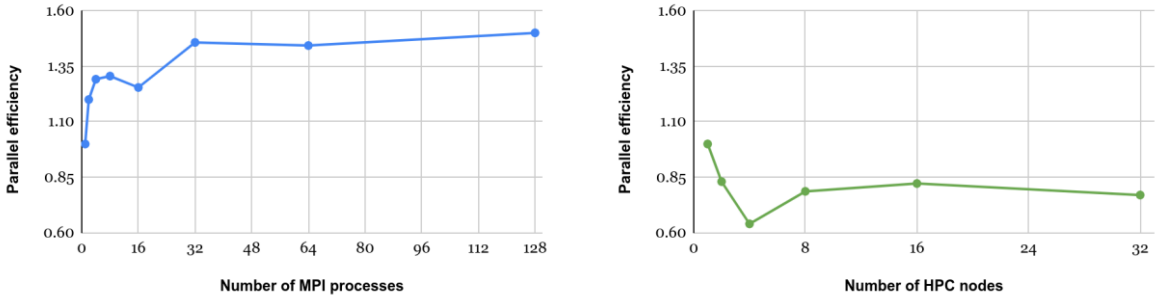
The analysis subroutine generates matrices and performs linear-algebra computations such as matrix-matrix and matrix-vector multiplications, solving systems of linear equations (SLEs), etc. These operations are implemented via functions provided by the BLAS and LAPACK libraries. Each call to the analysis subroutine involves allocating and deallocating memory space for numerous temporary arrays.

The ratio between the execution times of the two main procedures varies depending on the particular task specifics. Therefore, optimization and efficient GPU porting are equally important in both cases. For the sprint’s selected task, the search for observations subroutine consumed at least 85% of the total runtime.

Depending on the number of points in the local observation list, the matrix dimensions ranged from 40×40 to 200×200.

The parallel algorithm for distributed memory systems, implemented in pdaf-icono, is based on the domain decomposition method. Each MPI process first extracts the subset of observation points corresponding to its grid node domain, then runs the assimilation procedure without exchanging data with other processes. For pdaf-icono, the decomposition is done for the uppermost layer of the 3-dimensional model grid (thus, the ocean surface) and then the corresponding lower layers are added to the same domain. The partition of the ocean surface grid is performed by dividing the list of grid nodes into equal segments. In the case of linking PDAF with simulation codes (online coupling), the node distribution is received from the simulation driver.

Figure 1 presents strong scaling plots for pdaf-icono, generated from the results of experiments performed during the sprint: within a single node (fig. 1a) and across multiple nodes (fig. 1b). Hereafter, all programs and benchmarks were run on the Levante HPC system (DKRZ). When evaluating performance, parallel efficiency, and other metrics, only the execution time of the observation search and analysis subroutines was taken into account.



a) within a single node

b) across multiple nodes

Figure 1. Distributed-memory parallel efficiency.

The superlinear scaling in fig. 1a is attributed to the acceleration of the observation search procedure. The size of the observation dataset for a single MPI process's domain is less than that in the complete dataset. At the same time, the trivial decomposition algorithm does not account for the spatial distribution of grid nodes and observation points, nor their relative positions. Therefore, the scaling plot in fig. 1b does not demonstrate a clear relationship between parallel efficiency and the number of nodes.

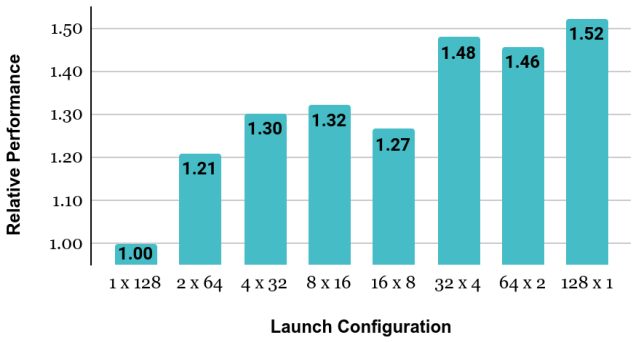


Figure 2. Relative performance comparison for different launch configurations (M MPI processes × N OpenMP threads each) using the hybrid parallelization model.

OpenMP threads operate on the same observation dataset as their parent MPI process. As a result, hybrid MPI+OpenMP parallelization is less efficient than pure MPI parallelization on the same number of CPU cores (fig. 2). Nevertheless, OpenMP threads demonstrated near-ideal scalability within a single MPI process domain.

The results discussed above were obtained using an executable built with the Intel compiler. After recompiling the source code with NVIDIA tools, the execution time increased by approximately 15%. The optimal performance in the MPI+OpenMP launch configuration was achieved with the setting OMP_SCHEDULE="dynamic,100".

GPU-accelerated versions of BLAS and LAPACK functions are provided in the cuBLAS, cuSOLVER, and cuSPARSE libraries. On average, the matrices processed by the analysis subroutine had dimensions not exceeding 64×64. It is well known that such a problem size is insufficient to effectively utilize a modern massively parallel accelerator. Therefore, as a first step in the code porting process, two custom-developed synthetic benchmarks were launched. These runs aimed to compare the execution times of relatively small kernels on the CPU and GPU using different launch configurations.

The benchmarks implemented loops that called the dgemm (cublasDgemm) or dgesv (cusolverDnDDgesv) subroutines on matrices grouped into a task array. CPU subroutines were called employing 32 OpenMP threads, while the optimal number of OpenMP threads (and CUDA streams) used for GPU kernels was determined experimentally. These settings correspond to a typical launch configuration on Levante: one node running 4 MPI processes, each MPI process utilizing 32 CPU cores and 1 GPU. The obtained results are summarized in Tables 1 and 2.

Problem size		Execution time (seconds)				
Matrix size	Number of tasks	Cuda BLAS subroutines		OpenACC self-implementation		CPU Blas subroutines
		Loop	Batched	Loop	Batched	
40	447392	0.81	< 0.05	1.02	0.04	0.62
64	176128	0.60		0.41	0.06	0.61
96	77824	0.15		0.19	0.10	0.60
128	43008	0.10		0.15	0.14	0.60
256	12288	0.04		0.32	0.41	0.73

Table 1. Matrix-matrix multiplication benchmark results.

The first matrix-matrix multiplication benchmark showed no performance gain from GPU offloading, even in the case of concurrent kernel execution. This is explained by the fact that the overhead of kernel launch management was comparable to the duration of the computations themselves. Notable performance gains were achieved only in batch mode (cublasDgemmBatched), in which tasks were distributed across thread blocks within a single large GPU kernel. However, PDAF cannot utilize the batched versions of cuBLAS subroutines, since they impose the restriction that all matrices must have identical dimensions. An OpenACC implementation of the algorithm for batched multiplication of variable-sized matrices, developed during the sprint, demonstrated comparable performance for single task size up to 64×64.

Problem size		Execution time (seconds)	
Matrix size	Number of tasks	CPU	GPU
40	447392	0.34	65.34
64	176128	0.34	31.51
96	77824	0.34	17.82
128	43008	0.34	11.85
256	12288	0.39	5.85
512	3072	0.50	3.66
1024	768	1.20	1.55
2048	192	1.91	0.79

Table 2. Results of the benchmark with the linear system solver subroutine.

The cuSolver library subroutines invoked in PDAF do not have batched versions. Most of them internally launch multiple GPU kernels, some of which are sequential (i.e., run on single-thread grids). Therefore, the performance of GPU computations observed in the second benchmark was entirely unsatisfactory. In this

case, GPU offloading is only justified if the data is already present in the device's memory, and the performance losses are compensated by acceleration in other parts of the code.

The benchmark results led to the conclusion that effective porting of the assimilation algorithm requires its conversion into a batch-processing format for grid nodes. However, such modifications would require substantial changes to the PDAF subroutine interfaces, which would be undesirable given the large user base. For this reason, it was decided to continue working with the current version of the code and to explore GPU performance without changing the program's original structure. In parallel with other efforts, the code was optimized to improve CPU performance.

The total size of the temporary workspace used by the analysis subroutine did not exceed 512 KB. Using custom-developed memory allocation functions based on the utilization of a preallocated heap had no impact on execution time, either on the CPU or GPU. Following a minor optimization of the PDAFomi_check_dist2_noniso_loop subroutine, its execution time was reduced by half. However, this performance improvement was limited to the executable produced by the Nvidia compiler, while the Intel-compiled version exhibited no performance gains.

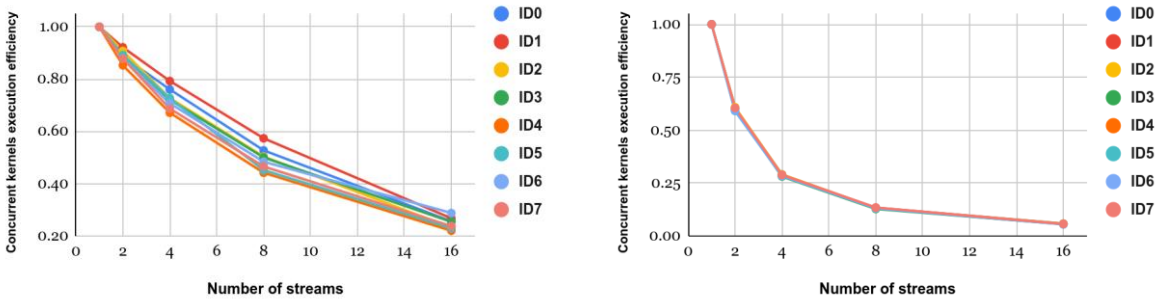
Preliminary GPU porting was performed based on the MPI+OpenMP code version. Each MPI process was assigned a dedicated GPU. OpenMP threads shared the device and launched OpenACC kernels alongside GPU-accelerated subroutines in their own CUDA streams. The code structure is illustrated schematically in fig. 3. Since the program infrastructure was not ported to the GPU, kernel launches alternated with host/device data transfers. The number of such operations was minimized.

```
! numOmpThreads – constant number of OpenMP threads
! idAccStream – OpenACC asynchronous stream ID

!$OMP PARALLEL NUM_THREADS(numOmpThreads) PRIVATE(idAccStream)
idAccStream = omp_get_thread_num() + 1

!$OMP DO
DO iGridNode=1, numGridNodes ! Main loop over grid nodes

! Adding first GPU kernel to the idAccStream queue
!$ACC PARALLEL ... ASYNC(idAccStream)
...
!$ACC END PARALLEL
...
! Adding last GPU kernel to the idAccStream queue
!$ACC PARALLEL ... ASYNC(idAccStream)
...
!$ACC END PARALLEL
...
```

```
! Waiting for all async kernels in idAccStream
! async queue to complete
!$ACC WAIT(idAccStream)
END DO
!$OMP END DO
!$OMP END PARALLEL
```

Figure 3. Code structure.

The GPU computation performance was evaluated separately for the observation search and analysis subroutines. The GPU executable was consistently launched on two Levante nodes (8 MPI processes), while varying the number of OpenMP threads per process from 1 to 16. Figure 4 presents graphs showing the efficiency of concurrent kernel execution separately for each MPI process.



a) analysis subroutine

b) observation search subroutine

Figure 4. Concurrent kernel execution efficiency.

In both cases, performance gains are only seen in the range of 2 to 4 threads (streams). The execution time of the software modules on the GPU is approximately equal to the performance of a single CPU core. These mediocre characteristics are partly due to delays in host-device data transfers, as well as GPU idle times caused by OpenMP thread desynchronization. However, based on the data gathered from benchmark runs, achieving computing performance comparable to the CPU - that is, accelerating the GPU executable by a factor of 50 to 100 - appears unrealistic. Therefore, to continue porting, it is necessary to switch to a batch format for processing grid nodes (fig. 5), and to develop subroutines for matrix-matrix and matrix-vector multiplication, adapted to arrays of tasks with varying sizes and PDAF data structures.

```
! sizeBatch – constant number of grid nodes in a batch
numBatches = numGridNodes / sizeBatch ! number of batches
DO iBatch=1, numBatches ! Main loop over grid node batches

sIndex = (iBatch - 1) * sizeBatch + 1 ! Index of the first grid node in the batch
```

```
fIndex = sIndex + sizeBatch ! Index of the last grid node in the batch
! GPU kernel for processing a batch of grid nodes
!$ACC PARALLEL LOOP ... ASYNC(1)
DO iGridNode=sIndex, fIndex
...
ENDDO
!$ACC END PARALLEL
!$ACC WAIT(1)
...
! Concurrent execution of GPU kernels to process individual grid nodes
!$OMP PARALLEL NUM_THREADS(numOmpThreadsX) PRIVATE(idAccStream)
idAccStream = omp_get_thread_num() + 1
!$OMP DO
DO iGridNode=sIndex, fIndex
! Adding GPU kernel to the idAccStream queue
!$ACC PARALLEL ... ASYNC(idAccStream)
...
!$ACC END PARALLEL
ENDDO
!$OMP END DO
!$ACC WAIT(idAccStream)
!$OMP END PARALLEL
...


ENDDO
```

Figure 5. Code structure for batch processing of grid nodes.

5 Results

The key results achieved during the sprint can be summarized as follows:

1. An evaluation of the parallel efficiency metrics of the offline-coupled pdaf-icono application and the performance of the core subroutines of the PDAF framework was performed.
2. Modifications to the CPU code have been prepared to check and set OpenMP thread affinity to CPU cores during program execution, as well as to improve the performance of the procedure for generating the local observations list.
3. The code was analyzed to identify bottlenecks in its GPU porting, and the advantages and disadvantages of various strategies for offloading computations to massively parallel accelerators were evaluated.

4. Preliminary porting of PDAF core subroutines to GPU has been performed without changes to the code structure or the framework subroutine call interface.

6 Conclusions and Outlook

The work in the sprint showed that the particularities of the data-assimilation algorithm, which leads to computations with small one- and two-dimensional arrays, in the example code often with sizes between 10 to 300 for vectors and 40 x 40 elements for matrices, leads to challenges for an efficient use of GPUs. Here further work is required. The sprint showed possible paths to proceed in the porting with possible code restructuring. One step in this direction was done with the release of PDAF V3.0 in May 2025. This separates more clearly the operations on observations from those on ensemble arrays. However, further analysis is required to see which further code changes might be required for performance GPU offloading. To this end, a further sprint request appears needed as the PDAF developers do not have sufficient experience with GPU coding.

7 References

PDAF Resources and References

Online resources

1. PDAF documentation. Available at: https://pdaf.awi.de

2. PDAF source code repository on GitHub. Available at: https://github.com/PDAF/PDAF

3. Citable version of PDAF with DOI on Zenodo. Available at: https://doi.org/10.5281/zenodo.7861812

4. PDAF entry in the Helmholtz Software Directory. Available at: https://helmholtz.software/software/pdaf

Publications

5. Nerger, L., Tang, Q., & Mu, L. (2020). Efficient ensemble data assimilation for coupled models with the Parallel Data Assimilation Framework: Example of AWI-CM. Geoscientific Model Development, 13, 4305–4321. https://doi.org/10.5194/gmd-13-4305-2020

6. Nerger, L., & Hiller, W. (2013). Software for Ensemble-based Data Assimilation Systems – Implementation Strategies and Scalability. Computers and Geosciences, 55, 110–118. https://doi.org/10.1016/j.cageo.2012.03.026

7.Nerger, L., Hiller, W., & Schröter, J. (2005). PDAF – The Parallel Data Assimilation Framework: Experiences with Kalman Filtering. In W. Zwieflhofer & G. Mozdzynski (Eds.), Use of High Performance Computing in Meteorology (pp. 63–83). Singapore: World Scientific. https://doi.org/10.1142/9789812701831_0006 [preprint]

NVIDIA CUDA Toolkit, Profiling Tools, and Associated Math Libraries

8. CUDA Toolkit – General Library and Programming Documentation: https://docs.nvidia.com/cuda/

9. cuBLAS – CUDA Basic Linear Algebra Subroutines Documentation: https://docs.nvidia.com/cuda/cublas/

10. cuSOLVER – CUDA Solver Library Documentation: https://docs.nvidia.com/cuda/cusolver/

11. Nvidia Nsight Systems: https://developer.nvidia.com/nsight-systems

12. OpenACC API: https://www.openacc.org/