MAchinE Learning for Scalable meTeoROlogy and climate

MAELSTROM

Report on software performance benchmarking for ML solutions from deliverable D1.4

Saleh Ashkboos

www.maelstrom-eurohpc.eu

MAELSTROM

D2.6 Report on software performance benchmarking for ML solutions from deliverable D1.4

Author(s):	Saleh Ashkboos (ETH)
Dissemination Level:	Public
Date:	25/03/2024
Version:	1.0
Contractual Delivery Date:	31/03/2024
Work Package/ Task:	WP2/ T2.2 T2.3 T2.4 T2.5 T2.6
Document Owner:	ETH
Contributors:	ECMWF, 4Cast
Status:	Final

MAELSTROM

Machine Learning for Scalable Meteorology and Climate

Research and Innovation Action (RIA)
H2020-JTI-EuroHPC-2019-1: Towards Extreme Scale Technologies and Applications

Project Coordinator:	Dr Peter Dueben (ECMWF)
Project Start Date:	01/04/2021
Project Duration:	36 months

Published by the MAELSTROM Consortium

Contact:
ECMWF, Shinfield Park, Reading, RG2 9AX, United Kingdom
Peter.Dueben@ecmwf.int

The MAELSTROM project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955513. The JU receives support from the European Union's Horizon 2020 research and innovation programme and United Kingdom, Germany, Italy, Luxembourg, Switzerland, Norway

Contents

1 Executive Summary

This delivery is a report on software performance benchmarking for machine learning solutions.

- Software benchmarking tools are on track and benchmarks have been completed.
- A comprehensive timing infrastructure for benchmarking deep learning applications has been developed.
- The machine learning solutions from Deliverable 1.4 have been benchmarked and the results are documented in this report.
- A new application (AP7) is presented (from ETH) which is exploring the use of machine learning for data assimilation in numerical weather prediction. The application is implemented in JAX and benchmarked using our infrastructure.

2 Introduction

2.1 About MAELSTROM

To develop Europe’s computer architecture of the future, MAELSTROM will co-design bespoke compute system designs for optimal application performance and energy efficiency, a software framework to optimise usability and training efficiency for machine learning at scale, and large-scale machine learning applications for the domain of weather and climate science.

The MAELSTROM compute system designs will benchmark the applications across a range of computing systems regarding energy consumption, time-to-solution, numerical precision and solution accuracy. Customized compute systems will be designed that are optimized for application needs to strengthen Europe’s high-performance computing portfolio and to pull recent hardware developments, driven by general machine learning applications, toward needs of weather and climate applications.

The MAELSTROM software framework will enable scientists to apply and compare machine learning tools and libraries efficiently across a wide range of computer systems. A user interface will link application developers with compute system designers, and automated benchmarking and error detection of machine learning solutions will be performed during the development phase. Tools will be published as open source.

The MAELSTROM machine learning applications will cover all important components of the workflow of weather and climate predictions including the processing of observations, the assimilation of observations to generate initial and reference conditions, model simulations, as well as post-processing of model data and the development of forecast products. For each application, benchmark datasets with up to 10 terabytes of data will be published online for training and machine learning tool-developments at the scale of the fastest supercomputers in the world. MAELSTROM machine learning solutions will serve as blueprint for a wide range of machine learning applications on supercomputers in the future.

2.2 Scope of this deliverable

2.2.1 Objectives of this deliverable

The objective of Deliverable 2.6 is to employ and enhance the benchmarking framework established within Work Package 2 for assessing the software performance of machine learning applications within the MAELSTROM project. The benchmarking process should be easy to incorporate with minimal interference in application performance, yet offer comprehensive insights for users to recognize bottlenecks or performance declines resulting from modifications. This framework is intended to empower MAELSTROM applications to offer actionable insights through routine performance evaluations.

2.2.2 Work performed in this deliverable

In the following, we present the tasks completed in this deliverable:

- An easy-to-use interface for timing (and benchmarking) the applications using Deep500 was developed.
- The software benchmarking of MAELSTROM applications using the introduced interface was performed.
- The software benchmarking results of Application 6 were included for the first time in this deliverable.
- A new application for data assimilation was introduced and benchmarked using our software benchmarking tools.

2.2.3 Deviations and counter measures

Software benchmark did not turn out to be as important for the MAELSTROM project as anticipated during the proposal-writing phase. This is mainly the case as the number of competitive software packages for machine learning in high performance computing has reduced significantly over the last years. Instead of a large zoo of libraries, the main environments that are used today are Pytorch, TensorFlow and JAX. After extensive consultations with the application teams, particularly during the last MAELSTROM General Assembly, it was determined to revise the primary objective of the current deliverable. While an additional round of benchmarking with the Deep500 framework was conducted for all applications in PyTorch and Tensorflow, a new application from ETH was developed by ETH, referred to as AP7 in this deliverable. For Deep500, we have devised a lightweight timing infrastructure that can seamlessly integrate into existing applications with minimal adjustments along with the development of an accessible API. Regardless of the approach, detailed software benchmarking results will continue to be generated.

3 Benchmarking Infrastructure by Work Package Tasks

We detail recent progress in the MAELSTROM workflow and software benchmarking packages. In the first part, we present a brief introduction to the new application for data assimilation. Then, we present the general idea of our benchmarking tools in this deliverable.

3.1 New Application AP7: Data assimilation for numerical weather predictions via machine learning

Modern weather forecasts heavily rely on numerical weather predictions. These predictions depend on data assimilation, which uses sparse observations to create initial conditions on a model grid. Errors in these initial conditions are a major reason for errors in forecasts. Data assimilation is also used to generate reanalysis datasets like ERA5, which recreate past weather in grid form. These datasets are crucial for weather and climate research and for training global weather forecast models that use machine learning.

Different techniques for data assimilation have been developed to handle various aspects of observation data and system dynamics. Among these, variational data assimilation and ensemble Kalman filter are the two most commonly used methods in operational data assimilation. Both approaches require considerable computational resources: the variational method demands multiple optimization iterations and the Kalman filter method performs numerous ensemble simulations. While a number of recent papers have shown the successful use of sophisticated machine learning models to build entire weather forecast models that are competitive with state-of-the-art conventional models (e.g. Graphcast, Fourcastnet, PanguWeather and Gencast), we present one of the first approaches to use machine learning models for data assimilation.

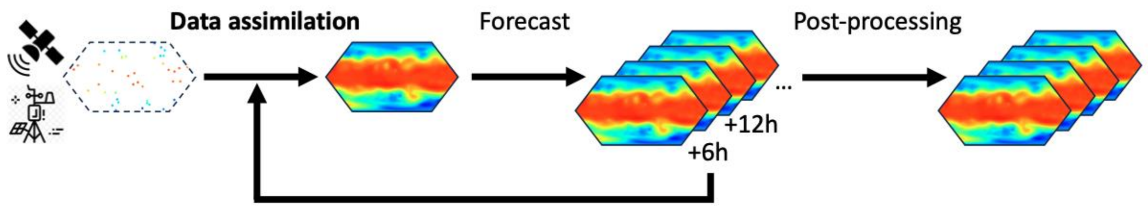


Figure 1*: Diagram of numerical weather forecast pipeline. It consists of data assimilation, forecast and post-processing. Data assimilation produces gridded values from sparse observations and predicted gridded values from previous time steps. Forecast takes in gridded values and produces predictions in gridded values at future time steps. Post-processing improves forecast results such that it is closer to future observations..*

From a probabilistic standpoint, data assimilation can be understood as sampling from a probability distribution of atmospheric states, taking into account observations and predicted states. Given its capacity to tackle conditional sampling, denoising diffusion models emerge as a promising option for data assimilation. Furthermore, the expanding realm of diffusion models offers a range of techniques for imposing various conditions. Particularly noteworthy are conditioning techniques for tasks like in-painting and super-resolution, which bear similarities to conditioning based on observations and predicted states, respectively.

While this approach using denoising diffusion models has been applied in smaller-scale data assimilation problems, none have been able to assimilate data at resolutions comparable to the ERA5 dataset (0.25 degree horizontal resolution), limiting their utility with machine learning forecast models. Similarly, denoising diffusion techniques have found application in weather forecasts and post-processing. For instance, GenCast capables of performing ensemble forecasts at a 1-degree resolution.

In this application, we introduce DiffDA [4], a novel approach to data assimilation centered around the denoising diffusion model, specifically tailored for weather and climate applications. Our method is capable of assimilating data with a resolution of 0.25 degrees with 13 vertical levels, leveraging the GraphCast ML weather forecast model as the backbone of the diffusion model. The work has already resulted in a scientific paper [4].

During the training phase, the diffusion model is conditioned by predicted states generated by the forecast model from earlier initial conditions. In the inference stage, we further condition the model with sparse column observations following the Repaint technique. Additionally, we employ a soft mask and interpolated observations to reinforce conditioning, leveraging the continuity of atmospheric variables.

Through this approach, the assimilated data can gradually converge towards the ground truth data as the number of samples increases.

3.2 Benchmarking Tools, Deployment, and Infrastructure (Tasks 2.3 and 2.6)

Deep500 [1] is a modular benchmarking tool that is developed to test high-tech deep learning programs. In this deliverable, we provide a simple and easy-to-use interface for Deep500 for dealing with weather and climate applications. We repeat the benchmarking of the applications in the MAELSTROM Deliverable 2.3 with the advanced versions of the MAELSTROM applications in this deliverable.

Our main focus is on making sure we can accurately measure and note important parts of a program's performance, like how long it takes to read and write data, and the time it takes for the program to learn from its mistakes. We also made sure Deep500 could handle any kind of program, no matter how it was built. Furthermore, we made sure it could easily connect with other programs used for keeping track of performance.

3.2.1 Timing for Machine Learning Applications

We added a new feature to Deep500 called the `Timer`. It lets developers mark the start and end of a section they want to measure. Then, it calculates how much time passed between the `start` and `end`. This feature gives developers the freedom to measure time in detail or in broad strokes. Each section is labeled with a key to describe what it's measuring, to make it easier to work with multiple timers within a single programme. Sections with different labels can overlap or be nested inside each other, which helps to break down the timing into details even more.

A timer is created as follows:

```
tmr = timer.Timer()
```

Then, the object can measure the time for every block of code. We define the following keys for the timer object:

- `timer.TimeType.EPOCH` - one complete pass over a dataset
- `timer.TimeType.BATCH` - one mini-batch
- `timer.TimeType.FORWARD` - forward propagation for one mini-batch
- `timer.TimeType.BACKWARD` - backward propagation for one mini-batch
- `timer.TimeType.COMM` - communication during one mini-batch
- `timer.TimeType.IO` - I/O to load data for one mini-batch
- `timer.TimeType.OTHER` - a catch-all for a user-defined region

Finally, the timer can be extended for measuring other parts of the code.

3.2.2 Timer Example Usage

Figure 2 shows the usage of our timer to measure the time of a single epoch in PyTorch.

```
for epoch in range(num_epochs):
  tmr.start(timer.TimeType.EPOCH)
  train_epoch(loader, net, criterion, optimizer, tmr)
  tmr.end(timer.TimeType.EPOCH)
```

***Figure 2**: Example of measuring epoch runtime using Deep500 timer.*

3.2.3 Timer Logging

Different deep-learning workloads use different logging frameworks. To this end, we provide several ways of logging the recorded times using our timers. We support both MLFlow and WandB in our timer. Currently, there are several ways to retrieve or save timing results.

- `tmr.get_time(key)` will return a list of all times recorded for `key`.
- `tmr.get_time_stats(key)` will return an object with various summary statistics for all times recorded for `key`.
- `tmr.print_all_time_stats()` will print (to `stdout`) summary statistics for all times recorded for all keys.
- `tmr.save_all_time_stats(filename)` is similar, but will write the output to the file named by `filename`.
- `tmr.log_wb_all(prefix='')` will log the mean time of all keys recorded to Weights & Biases (optionally prepending `prefix` to the logging keys).
- `tmr.log_mlflow_all(prefix='')` is similar, but will log to MLFlow instead.

3.2.4 Timing GPU codes in PyTorch and TensorFlow

Code running on the GPU typically executes asynchronously (i.e., the CPU launches a series of kernels, but does not wait for their computations to complete, only checking later). This means that timing solely on the CPU may not adequately reflect the actual computation time for certain regions. When using PyTorch or TensorFlow, we also support adding timers for GPU kernels. For PyTorch, this is low-overhead; due to technical issues, there can be additional overheads with TensorFlow.

To solve this issue, we made the Timer class flexible and adaptable. By default, it uses standard high-quality timers found in computer processors. Additionally, we created a special timer for regions using GPUs called CUDA events. These events are a part of the CUDA system, used for measuring time on graphics cards. They only communicate with the main processor when needed, so they don't slow down the program much.

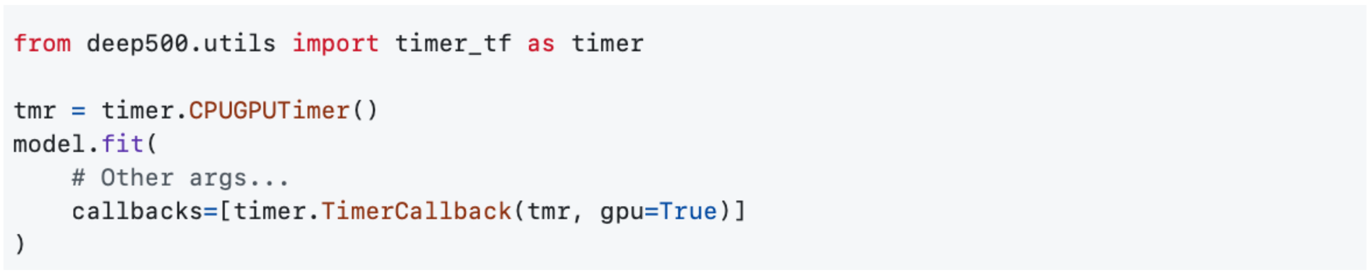
However, in the current setup, timing on GPUs might be slower when using frameworks other than PyTorch, like TensorFlow. This is because those frameworks don't show the inner workings of CUDA streams, making timings harder. We're working on updating our tools to fix this issue.

3.2.5 TensorFlow/Keras Support

Deep500 has a simple way to measure time on the CPU, which can be used in Deep500 recipes. Also, it can time specific parts of a program using its events feature. Our Timer feature does more—it can also time things on the GPU and keep track of what's happening. However, most programs in D1.4 aren't made to work with Deep500 for their benchmarking. However, users can add timing to their programs without changing anything or needing extra software.

To fix this, we extend our timing API to be compatible with TensorFlow/Keras. In such frameworks, the user usually involves a training loop using `model.fit()` and does not explicitly define training loops. We provide a callback that supports epoch- and batch-level timing during training.

The figure 3 shows an example of using our API for measuring the time for a TensorFlow program.



```
from deep500.utils import timer_tf as timer

tmr = timer.CPUGPUTimer()
model.fit(
    # Other args...
    callbacks=[timer.TimerCallback(tmr, gpu=True)]
)
```

***Figure 3**: Example of using our timing API in TensorFlow.*

4 Software Benchmarking Results

Working alongside the application teams of Work Package 1, we employed the Deep500 timing infrastructure to carry out the initial software benchmarking of the machine learning solutions detailed in Deliverable 1.4. With the support of Work Package 2, each application seamlessly integrated the timers and executed benchmarks to yield these findings. Subsequently, in Deliverable 1.4, we presented a comprehensive software benchmarking analysis for all applications leveraging Deep500. Here, we present another round of software benchmarking using the API we developed on top of Deep500 (the code is available at this repo: https://github.com/sashkboos/Deep500-for-MAELSTROM).

We unify all the benchmarking across all the applications and report the time for each epoch for both forward and backward pass as well as IO. Additionally, we include the benchmarking results of the new application (AP7 from ETH) about data assimilation. Below we report the results for each application, along with brief details of their benchmarking setup.

4.1 Application 1: Blend Citizen Observations and Numerical Weather Forecasts

Application 1 aims to produce high-resolution (1x1 km) hourly temperature forecasts for the Nordic countries, 58 hours into the future.

The benchmark consists of training a U-Net on a subset of the A1 dataset. We used the same model configuration as in D3-7. We chose a small subset (9 out of 362 available training files) of the dataset and ran it for 10 epochs. The 124GB training data was not cached in memory (even though it fits) in order to simulate the expected performance when running on the full dataset.

We ran the benchmark on a single 40GB A100 GPU on the JURECA system. We used NVIDIA's tensorflow container, which includes Python3.8, TensorFlow 2.12, and CUDA 12.1. As in other deliverables, the data processing for this application was done on the CPU, and the training on the GPU.

The variability of the processing times are relatively low, as the minimum times are close to the median times. The exception is the first batch, which is significantly slower than the others. This is due to the fact that it includes the data loading time for the first file and framework initialization overhead. This is similar to the results we got in D2.3.

Metric	Min [s]	Mean [s]	Median [s]	Max [s]	Stdev [s]
Epoch (CPU)	113.47643	121.92249	120.18363	142.09439	8.50714
Batch (CPU)	0.07399	0.11428	0.08892	15.80655	0.19250
Batch (GPU)	0.07400	0.11430	0.08895	15.80662	0.19251

***Table 1**: Benchmarking results for Application 1.*

4.2 Application 2: Incorporate Social Media Data into the Prediction Framework

Application 2 aims to use data provided by social media and weather sensors. Here, we use the text of Tweets to predict the occurrence of rain at the Tweet's location and time of creation.

We initialise the *small* variant of the DeBERTa model [3] with pre-trained weights from the *Hugging Face* repository[1]. We finetune the model during one epoch on our large dataset (~1.6 Mio. Tweets). We used the transformer package (Version 4.38.2) with the PyTorch backend (Version 2.2.1) to train our model on a NVIDIA Tesla A100 from the Jülich Supercomputing Centre (JSC) using CUDA 11.7. Batch, forward, and backward time was measured using GPU-side timing.

Metric	Min [s]	Mean [s]	Median [s]	Max [s]	Stdev [s]
Epoch	180.14980	180.14980	180.14980	180.14980	0.00000
Batch	0.11619	0.13142	0.12229	8.32839	0.24195
Forward	0.05676	0.06631	0.05759	7.70765	0.22562
Backward	0.04796	0.04861	0.04843	0.18809	0.00414

***Table 2**: Benchmarking results for Application 2.*

[1] https://huggingface.co/microsoft/deberta-v3-small/tree/main

4.3 Application 3: Neural Network Emulators to Speed Up Weather Forecast Models and Data Assimilation

Application 3 seeks to emulate the radiative transfer process for both short and long wavelengths, a columnar problem found within all weather and climate models.

For this software benchmarking exercise, we have used Tensorflow Version 2.11 with CUDA 11.7, using one A100 GPU from Jülich JURECA DC module. The model used is a predominantly RNN architecture, specifically using LSTM blocks to propagate information in the vertical dimension. Details of the architecture can be found in D1.3 and D1.4.

Training has been carried out following the same methodology already introduced in previous software and hardware benchmarking exercises. In this case, we use 5 epochs, and limit the number of batches per epoch to 1000.

Epoch time was measured using GPU-side and CPU-side timing, while batch time was measured using CPU-side timing.

Similarly to the previous deliverable, for the batch time the maximum time is larger than the mean and median time, but the low standard deviation indicates that this should be treated as an outlier due to the initialization overheads that occur in the first batch. That overhead can also be seen in the case of the epoch time, where the maximum recorded is probably associated with the first epoch.

Metric	Min [s]	Mean [s]	Median [s]	Max [s]	Stdev [s]
Epoch (CPU)	86.65345	104.82769	88.62053	170.86387	36.95771
Batch (CPU)	0.02116	0.07561	0.07220	13.67991	0.19281
Batch (GPU)	0.02117	0.07562	0.0722	13.67990	0.19281

***Table 3**: Benchmarking results for Application 3.*

4.4 Application 4: Improve Ensemble Predictions in Forecast Post-Processing

Application 4 aims to apply deep neural models to post-process the ensemble outputs of ensemble numerical weather prediction systems to improve the quality and skill of forecasts. We use ENS-10 [2] dataset with a U-Net style model and trained for 3 epochs with different batch sizes and the Adam optimizer, following the prior methodology. We used NetCDF format for saving the dataet. Training was performed using PyTorch 1.13.1 and TensorFlow 2.11.0 on a single 40 GB A100 GPU from local ETH computing resources using CUDA 11.6.2. Batch, forward, and backward time was measured using GPU-side timing.

After benchmarking the experiments with PyTorch and TensorFlow, we found that our results are quite similar to the last deliverable. This is expected as the application is unchanged and the infrastructure is also the same. Tables 4-1 and 4-2 summarise our results on both PyTorch and TensorFlow frameworks.

Metric	Min [s]	Mean [s]	Median [s]	Max [s]	Stdev [s]
Epoch	180.12	188.1	183.3	199.7	10.3
Batch	0.053	0.15	0.14	5.99	0.57
Forward	0.006	0.007	0.006	5.1	0.054
Backward	0.015	0.016	0.015	0.43	0.007
IO	0.03	0.11	0.11	5.8	0.19

***Table 4**: Benchmarking results for Application 4 in PyTorch.*

Metric	Min [s]	Mean [s]	Median [s]	Max [s]	Stdev [s]
Epoch	195.1	252.1	263.1	288.2	36.18
Batch	0.05	0.15	0.09	33.1	1.3
IO	3e-4	0.04	2e-3	2.7	0.62

***Table 5**: Benchmarking results for Application 4 TensorFlow.*

Comparing two tables, we can see that TFRecords (in TensorFlow) reduces the IO by order of magnitude. However, the total time for each epoch is not that much different from PyTorch implementation due to other framework related overheads.

4.5 Application 5: Improve Local Weather Predictions in Forecast Post-Processing

Application 5 explores the usage of deep neural networks for statistical downscaling of meteorological fields. The Tier-2 dataset, which is tested here, is designed for downscaling the 2m temperature from ERA5 reanalysis data with $\Delta x_{ERA5} \simeq 30$ km to the spatial resolution of the COSMO REA6 dataset ($\Delta x_{CREA6} \simeq 6$ km).

The same WGAN configuration and dataset as used in D3-7 has been used. Compared to D2-3, this results in an increased number of trainable parameters (about 9M instead of 5M) and a larger number of predictor variables (15 instead of 10). For this benchmark, the downscaling model has been trained for 5 epochs on a single A100 GPU on JURECA DC using Tensorflow 2.6.0 and CUDA 11.5 as provided by the system's software module stack.

During training, the 132 monthly netCDF-files (providing 94052 training samples) are split into four subsets that are iteratively pipelined through the model. Thus, training data is not cached during training, although the dataset size (about 62 GB) would allow for it.

The overall results attained in this benchmark test are similar to the results in D2-3. The first epoch (batch) takes significantly more time than the subsequent epochs (batches) due to the initialisation overhead (building up the computation graph and first data loading). The small temporal distance between the minimum and the median indicates that the training process shows little fluctuation and that the mean is strongly influenced by the first epoch (batch) due to the small number of epochs probed here. The strong agreement between processing times on the CPU and GPU furthermore indicate that the application is not I/O bottlenecked which can furthermore be verified from the LLview job reports available for jobs on JURECA DC.

Metric	Min [s]	Mean [s]	Median [s]	Max [s]	Stdev [s]
Epoch (CPU)	898.971	925.9487	906.38677	1007.75919	46.30957
Batch (CPU)	0.20732	0.29217	0.22048	75.90584	0.73326
Batch (GPU)	0.20733	0.29220	0.22050	75.88095	0.73308

Table 6*: Benchmarking results for Application 5.*

4.6 Application 6: Provide Bespoke Weather Forecasts to Support Energy Production in Europe

Application 6 explores the potential of weather models influenced by extensive weather patterns across Europe to diminish the uncertainty associated with power forecasting for wind and solar resources. To achieve this, we make use of a nonlinear Deep Learning algorithm that was developed for unsupervised clustering in image recognition. [5]

For benchmarking we use the same model architecture and configuration, and dataset as in D3-7, and train the model for 20 epochs. The benchmarks are performed on a single node of the JUWELS Booster system with a single A100 GPU (40 GB VRAM) with the following software configuration:

- CUDA 12.1.0
- Python 3.11
- PyTorch 2.2.1

Data processing is performed on the CPU, whereas the model is trained on the GPU.

The benchmarks show the epoch time is stable throughout the training process with no large difference between minimum and maximum, and median and mean being very close with a low standard deviation. This is almost equally true for the batch time, although the batch time has a significantly lower minimum, which becomes irrelevant though due to the small standard deviation.

IO, Forward and Backward times reveal that IO is the training step that accounts for most of the runtime of the algorithm. This is expected, however, since the algorithm by design doesn’t allow any sort of caching since it relies on various different data augmentation strategies to create random subsamples for each data point of a given batch. This is also included in the measurement of the IO times.

Metric	Min [s]	Mean [s]	Median [s]	Max [s]	Stdev [s]
Epoch (CPU)	220.97855	221.93730	221.77043	224.89921	0.89980
Batch (CPU)	0.04801	0.64551	0.64088	3.41739	0.05295
Batch (GPU)	0.04802	0.64552	0.64088	3.41728	0.05295
IO (CPU)	0.01964	0.59473	0.59078	2.24408	0.04050
Forward (CPU)	0.01097	0.01377	0.01359	0.86987	0.01075
Forward (GPU)	0.01095	0.01372	0.01355	0.86981	0.01075
Backward (CPU)	0.01467	0.01523	0.01485	1.81851	0.02274
Backward (GPU)	0.01545	0.01889	0.01854	1.81861	0.02269

***Table 7**: Benchmarking results for Application 6.*

4.7 Application 7: Data Assimilation

Application 7 explores a pure machine learning approach for data assimilation. We create a denoising diffusion model with a GraphCast backbone. The model performs data assimilation by sampling from the posterior distribution of $p(X^t|X^{t-1}, y)$ where X^{t-1} and X^t are gridded atmosphere states at time steps $t-1$ and t, and y is a sparse measurement of X^t. It starts from a pure white with the same shape as X, denoises the inputs in multiple iterations, and finally results in a plausible sample from the target distribution. To guide the generation of each samples, the conditioning of X^{t-1} is added to the process by allowing it as the second input of the denoising diffusion model. This requires training a dedicated model with two inputs. In contrast, we apply the conditioning of y only in the inference process using inpainting techniques: in each denoising iteration, the denoised state is mixed with the measurements in y. To enhance the effect of the conditioning, we augment y with interpolated values in the neighbouring regions of measurement locations before mixing it with the denoised state.

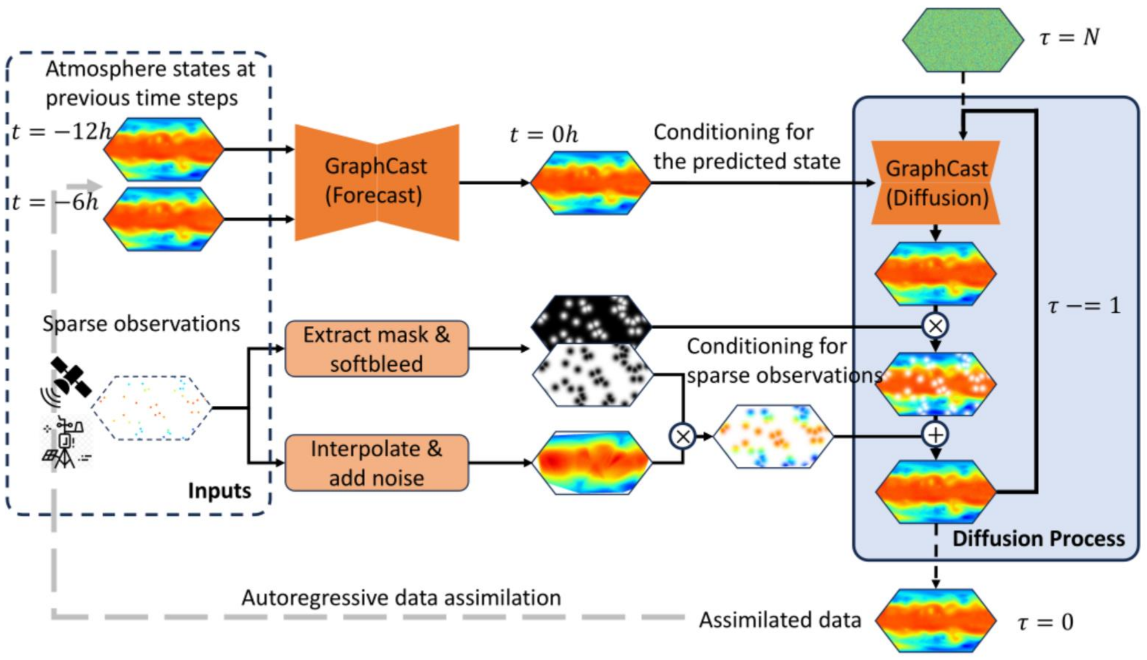


***Figure 4**: Architecture of the diffusion based data assimilation method (from [4]). We take advantage of the input and output shape of the pretrained GraphCast model which takes the state of the atmosphere at two time steps as input. In each iteration of the denoising diffusion process in our method, the adapted GraphCast model takes the predicted state and the assimilated state with noise, and further denoises assimilated state. To enforce the observations at inference time, the denoised state is merged with interpolated observations using a soft mask created by blurring the hard mask of the original observations.*

With the help of the GraphCast [7] backbone, we are able to run data assimilation with 6 pressure level variables and 5 surface level variables at a spatial resolution of 0.25 degree globally (~ 25 km resolution) and 13 pressure levels. The model is implemented in the JAX framework and trained with 48 NVIDIA A100 80GB GPUs in the CSCS research cluster. We use a subset of the ERA5 dataset as the training data containing 6 hourly data for each variable from 1979 to 2016 with a total size of around 25 TB. The training lasts for 2 days with 20 epochs and a (global) batch size of 56.

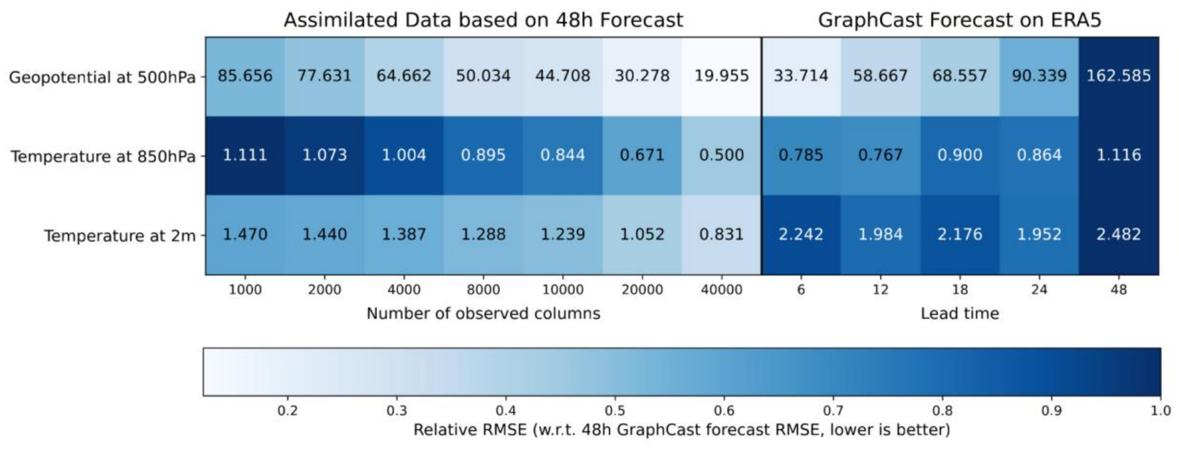


***Figure 5**: Root mean square error (RMSE) of geopotential at 500hPa, temperature at 850hPa, and temperature at 2m from the single step assimilated data and GraphCast forecast data (shown by the numbers in the cell). The error is calculated against the ERA5 data. The cells are color-coded with the RMSE relative to the 6h forecast RMSE where green means better and red means worse. The figure and results are from [4].*

We test our method by running data assimilation from a 48-hour GraphCast prediction and columns of simulated observations ranging from 1,000 to 40,000, then calculating the error between the assimilated data and the ERA5 data. The simulated observations columns are taken from the ERA5 dataset. With only 1,000 observed columns (<0.1% total columns), the assimilated data achieves lower RMSEs than the input 48-hour forecast for z500 (geopotential at 500hPa) and t850 (temperature at 850hPa). The errors of t2m (temperature at 2m) are even lower than the 6-hour forecast. The errors further decrease as the number of observed columns increases. On the other side of the spectrum, with 40,000 observed columns (<3.9% total columns), the RMSEs of all three variables are lower than the 6-hour forecast error. This indicates our assimilation method can improve over the given predicted state while remaining consistent with the observations.

When used as an input for forecast models, those assimilated data resulted in a maximum lead time loss of 24 hours compared with using the ERA5 dataset as inputs. This enables running data assimilation and simulation in an autoregressive cycle.

All the data assimilation experiments can run on a single high-end PC with a GPU within 15-30 minutes per data assimilation step, while a similar task using traditional methods typically requires large compute clusters. This indicates a significant reduction in computational costs. It also opens up the possibility of assimilating more observational data that is otherwise discarded by traditional methods and producing assimilated data with higher accuracy.

Metric	Min [s]	Mean [s]	Median [s]	Max [s]	Stdev [s]
Epoch	7314.87	7344.07	7337.00	7437.38	34.65
Batch	6.30	10.11	10.11	12.51	0.13
Computation	5.77	9.74	9.74	12.02	0.13
IO	0.25	0.37	0.36	1.39	0.06

***Table 8**: Benchmarking results for Application 7 on JAX.*

5 Conclusion

This deliverable outlines the MAELSTROM software benchmarking infrastructure and presents the outcomes of the benchmarking conducted on the machine learning solutions from Deliverable 1.4. We also include a new application (AP7 from ETH) and benchmark it using our API on top of Deep500. The benchmarking infrastructure proves its simplicity (and ease of use) across all applications and we show a unified way of benchmarking different applications.

6 References

[1] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. “A modular benchmarking infrastructure for high-performance and reproducible deep learning.” In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[2] Saleh Ashkboos, Langwen Huang, Nikoli Dryden, Tal Ben-Nun, Peter Dueben, Lukas Gianinazzi, Luca Kummer, and Torsten Hoefler. “ENS-10: A dataset for post-processing ensemble weather forecasts.” In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[3] Pengcheng He, Jianfeng Gao, and Weizhu Chen. “DeBERTaV3: Improving DeBERTa using ELECTRA-Style Pre-Training with Gradient-Disentangled Embedding Sharing.” *arXiv preprint:2111.09543,* 2021.

[4] Huang, Langwen, Lukas Gianinazzi, Yuejiang Yu, Peter D. Dueben, and Torsten Hoefler. "DiffDA: a diffusion model for weather-scale data assimilation." *arXiv preprint arXiv:2401.05932* (2024).

[5] Caron, Mathilde, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski and Armand Joulin. “Unsupervised Learning of Visual Features by Contrasting Cluster Assignments.” *ArXiv* abs/2006.09882 (2020): n. pag.

[6] Lam, Remi, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Ferran Alet, Suman Ravuri et al. "GraphCast: Learning skillful medium-range global weather forecasting." *arXiv preprint arXiv:2212.12794* (2022).

Document History

Version	Author(s)	Date	Changes
0.1	Saleh Ashkboos (ETH)	02/02/2024	Initial draft

Internal Review History

Internal Reviewers	Date	Comments
Peter Dueben (ECMWF)	21/03/2024	Minor comments and suggestions provided
Peter Duebenn (ECMWF)	24/03/2024	Minor comments and suggestions provided
Mats Brorsson (UL-SNT)	24/03/2024	Minor comments

Estimated Effort Contribution per Partner

Partner	Effort
ETH	1 PM
4cast	1 PM
Total	**2 PM**