D8.3 Benchmarking Results all Use Case Studies

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 2.1

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

This document reports on the Benchmarking Results all Use Case Studies.

In this deliverable, we recap on the use cases, give detailed information about the benchmarking setup and report on the benchmarking results of the various use cases for the DAPHNE system infrastructure.

D8.3 Benchmarking results all use case studies			
WP8 – Use Case Studies			
Type of document	R	Version	2.1
Dissemination level	PU		
Lead partner	KAI		
Author(s)	Benjamin Steinwender (KAI), Vytautas Jancauskas (DLR), Andreas Laber (IFAT), Marius Birkenbach (KAI), Bernhard Einberger (AVL), Daniel Krems (AVL), Aleš Zamuda (UM)		
Contributors	Ilin Tolovski (HPI), Matthias Boehm (TUB)		

Revision History

Version	Item	Comment	Author / Reviewer
V1.0	Merged documents of individual uses cases		Steinwender
V1.1	Updates to various use-cases		Steinwender
V2.0	Incorporated reviewer feedback		Steinwender
V2.1	Incorporated updates from WP partners		Steinwender

Executive Summary

The use case pipelines presented in the DAPHNE project present a diverse set of real-world scenarios. The use cases are enhanced by utilizing the DAPHNE system infrastructure. Furthermore, the use cases help to identify possible problems in the early version of DAPHNE.

To perform the benchmarking experiments, it was first necessary to port our use-cases to use either DaphneLib (Python library) or DaphneDSL (native DAPHNE language). DAPHNE provides a variety of readily available algorithms as callable functions directly[1] (among them are e.g., decision trees, linear regression, principal component analysis, random forest, …). Furthermore, the DAPHNE system infrastructure is documented publicly on the Internet[2]. Hence, writing the required DAPHNE script is generally user friendly.

Afterwards, we were able to run the experiments and depending on the use-case, we can achieve a substantial performance improvement in some cases. In the use cases, DAPHNE competes against numerical packages like NumPy[3], which have been optimized since many years. Still, DAPHNE is able to outperform NumPy in the KAI use case by a factor of about 3.16 using the same algorithm. By switching the method in the AVL2 use case, the execution time could be reduced from about four days to less than one day.

However, we have also identified some weaknesses in the currently available early version of the DAPHNE system infrastructure as well as the UMLAUT[4] benchmarking tool provided by our partners from WP9. We reported the issues to the technical partners – most notably the memory leaks mentioned further down in the document. Most of these leaks have already been fixed lately and even some of our experiments could be rerun to collect most recent results.

[1] https://github.com/daphne-eu/daphne/tree/main/scripts/algorithms
[2] https://daphne-eu.github.io/daphne
[3] https://numpy.org
[4] https://github.com/hpides/End-to-end-ML-System-Benchmark

Table of Contents

List of Abbreviations

Abbreviation	Meaning
APC	Advanced Process Control
CFD	Computational Fluid Dynamics
CI	Computational Intelligence
DE	Differential Evolution
DOE	Design of Experiments
DUT	Device Under Test
GB	Gigabyte
HDEMG	High-density surface electromyography
HPC	High-Performance Computing
LCZ	Local Climate Zones
LLVM	LLVM is a project name that originally was an initialism for Low Level Virtual Machine
LSF	Load Sharing Facility
MLIR	Multi-Level Intermediate Representation
RDP	Ramer-Douglas-Peucker
ROA	Randomized Optimization Algorithms
VLS-GO	Very Large Scale Global Optimization
VW	Visvalingam-Whyatt

1 Introduction

In the past months since submitting the early version of the benchmark results (deliverable D8.2), we have further provided feedback to the technical partners. By evaluating our real-world use-cases, we have spotted several weaknesses and bugs in the still early version of the DAPHNE system. Therefore, we reported these issues online on GitHub. The technical partners have already been able to already fix most of the encountered issues (e.g., GH#788) and provided discussions and concepts where an immediate implementation is not possible (e.g., GH#755).

Therefore, we are now able to report on our benchmarking results and where the DAPHNE system provides usability and/or performance improvements.

1.1 Outline

In the following chapters, the individual use cases report on their status in more detail. The benchmarking setup of the various use cases is described as well as the comparison to the respective baseline implementation.

Furthermore, our partners within WP8 have also provided the description and benchmarking results of additional integrated pipelines, which are reported in chapter 7.

2 Earth Observation Case Study: Local Climate Zone Classification (DLR)

Local Climate Zones (LCZ) are a way to classify land based on its use. Unlike other land use classification schemes, it focuses on urban environments. LCZ classification divides land in to 17 classes, 10 of which correspond to different built-up areas while the remaining 7 correspond to land without buildings on it. Those classes are listed in Figure 2.1. While it was originally developed as an aid to study urban heat islands, it has found further use in various other research scenarios, for example, urban development, transport research, disaster mitigation and population assessment among others. At DLR, we use LCZ to facilitate research in such fields as the study of transport flows and building height estimation. It is thus of interest to us to have a way to acquire up to date LCZ classifications of any area on earth. For that, we need to rely on satellite observations of the Earth. Both radar and optical imagery can be used but we focus on optical data in this case. Ideally the user would specify a region of interest and a period and be able to retrieve an LCZ classification for that area that will look similar like the one in Figure 2.1 (right).

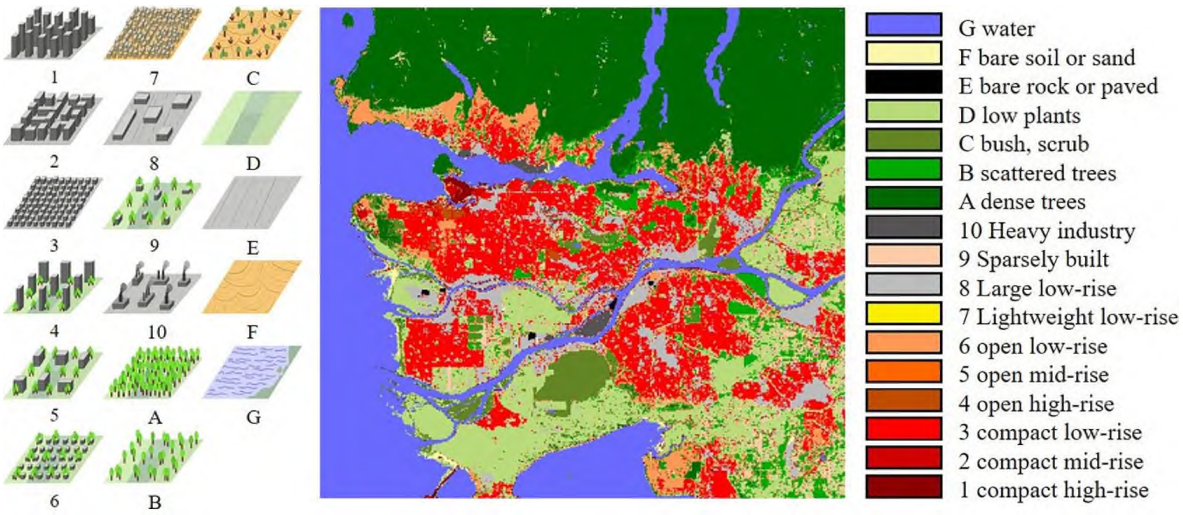


Figure 2.1: LCZ classification classes

At DLR we developed a pipeline to allow for LCZ classification of arbitrary geographical areas. LCZ classification uses a ResNet deep learning model trained on the LCZ42 data set to classify land according to its LCZ label.

2.1 Description of the Use-Case Pipeline

The pipeline for global LCZ classification consists of four main steps that can be summarized as follows:

1. Dividing the geographical area of interest into chunks of manageable size (half a degree of geographical latitude and longitude in our case).
2. Constructing a cloud-free composite image of that area from Sentinel-2 data, using a user specified date range.
3. Using a deep learning model (ResNet50) to identify LCZ labels for that patch.
4. Reconstruct the original geographical area from the LCZ classified patches.

Out of these steps we have identified the construction of the cloud-free composite image as the most likely to benefit from accelerations provided by DAPHNE. That is because the other steps are either very computationally cheap (like the patch division and assembly steps) or neural network inference. Inference is already very fast, as modern frameworks used for it such as PyTorch are very well optimized due to their prominent role in industry.

Let us quickly explain the compositing algorithm and its implementation. It can be outlined as follows:

Inputs: A date range and geographical coordinates for a rectangular patch of size 0.5 times 0.5 degrees.

1. Retrieve all Sentinel-2 tiles that overlap with the patch and that were taken within the date range.
2. For each pixel in the patch:
    1. For each image in the time series choose corresponding pixels that are there (not missing in that Sentinel-2 tile), cloud free and shadow free. We use cloud masks provided with the Sentinel-2 L2A product for this purpose.
    2. Calculate a medoid of these pixels (explanation in the following paragraph).
    3. Use the medoid as the composite pixel in the resulting cloud-free patch.
3. Write the results into a GeoTIFF file with the same dimensions and same location as the user specified patch.

We use the medoid as the compositing statistic. The medoid is an extension of the concept of the median to more than one dimension. The medoid is the point in multi-dimensional space that has the lowest average distance to all the other points.

$$x_{\text{medoid}} = \arg\min_{y \in X} \sum_{i=1}^{n} d(y, x_i).$$

It can be illustrated with Figure 2.2:

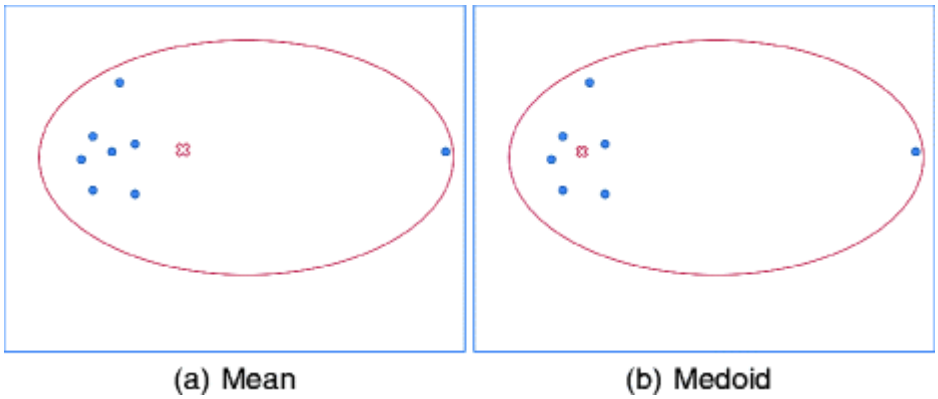


Figure 2.2: Mean vs. Medoid

The reason for preferring the medoid over other statistics lies in the fact that a medoid of a group of multi-dimensional points (in this case 13-dimensional since there are 13 Sentinel-2 bands) is a point that actually belongs to that group. A mean of a group of multi-dimensional

pixels can end up being in a sense synthetic (not found in the original data) which can lead to some questions about the validity of performing further data analysis on them. The code for calculating the medoid is not very complex, but it involves some fair number of matrix operations which we were hoping DAPHNE could optimize.

2.2 Benchmarking Setup

For benchmarking purposes, we have used the area shown in Figure 2.3. We have downloaded all Sentinel-2 tiles between 2019-05-01 and 2019-10-01 that overlap with this area (688 x 843 pixels in size) from the Sentinel-2 archive. We then cropped the tiles to retrieve only the part that overlaps with the area of interest. In the end we have ended up with 36 images to be used in the medoid compositing algorithm.

Figure 2.3: Area of interest

Two implementations of the compositing algorithm were written – one in NumPy and another using the DAPHNE Python API (DaphneLib). Then, the execution time for each version using a subset of the 36 downloaded images was measured. The algorithm was run on a single image, then two images, then three and so on until 36 images were reached. The time it takes to complete the composite was recorded for each subset and for each implementation. An example composite image result is shown in Figure 2.4.

The code that was used for this benchmark can be found in a GitHub repository[5]. The comparison plot is given below. In the "Number of images" axis of this plot we have the number of images used for the cloud compositing algorithm. We artificially limit the number of images retrieved from the Sentinel-2 archive and used in the medoid algorithm in order to explore the scaling of the algorithm with regards to number of images available. The scaling should be quadratic since we are measuring pairwise distances between multi-spectral pixels of individual images. This seems to be consistent with the results seen in Figure 2.5.

[5] https://github.com/DLR-MF-DAS/daphne-cloud-removal-example

Figure 2.4: Example cloud-free composite image result

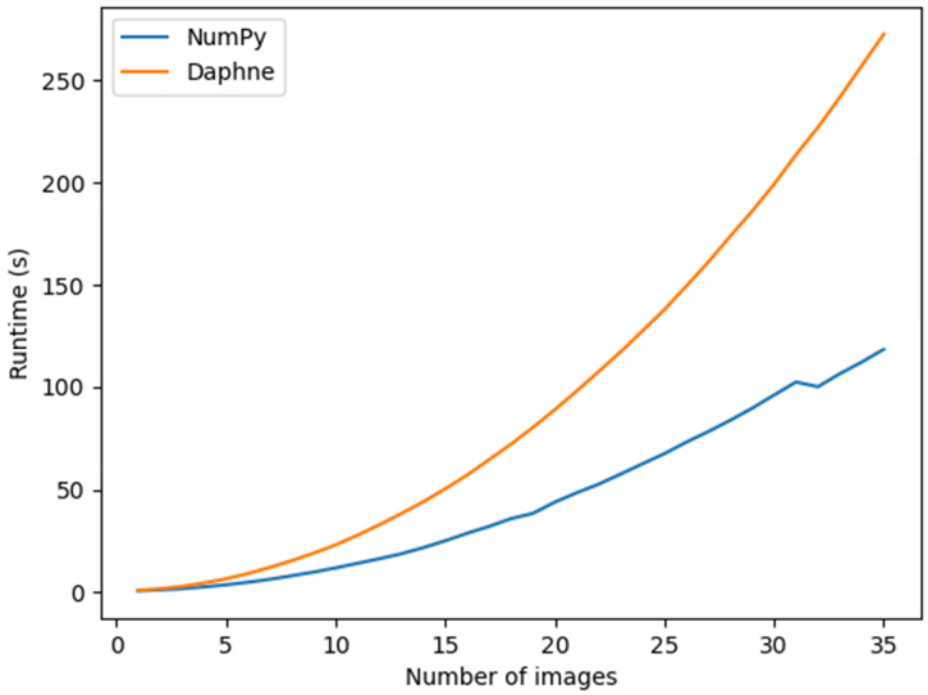


Figure 2.5: Performance comparison between DAPHNE and NumPy versions of the compositing algorithm. Lower values mean better performance

2.3 Productivity & Performance Improvements

Since the previous deliverable 8.2, we have decided to re-focus on a different aspect of the global LCZ classification pipeline. Instead of focusing on the inference part of the pipeline which takes a relatively small amount of time we have instead implemented the cloud removal algorithm in DAPHNE. The resulting implementation is considerably more complicated than the NumPy version because DAPHNE only supports two-dimensional matrices, so each image must be handled as a list of such matrices as opposed to a single 3D tensor. Performance of the DAPHNE version is also significantly worse than that of the NumPy version. While both

display roughly quadratic scaling with regards to the number of images used, NumPy is roughly twice as fast as DAPHNE. The causes of this are not clearly apparent but could be since it is not currently possible to use tensors in DAPHNE.

The primary effort during the course of this project went into the scaling of the code towards global (Earth) scales. To this end we have implemented a convenient system to describe processing workflows using YAML files (using pypyr[6]) as well as implementing code necessary to split large areas into smaller patches, process them and re-assemble the results. In these larger scales, fault tolerance becomes an issue and we have implemented functionality to resume processing after failure at any of the previously described processing steps as well as a logging and debugging system that lets users find and debug issues at any of the processing stages without losing progress. This is very important since processing a large area can take weeks of time.

[6] https://pypyr.io

3 Semiconductor Manufacturing Case Study (IFAT)

In view of the high total cost of semiconductor manufacturing assets, respective equipment needs to be as productive as possible. To avoid needless idling and unnecessary downtime, scheduling and maintenance strategies are important in practice. We developed a novel approach to reduce the substantial setup costs inherent to ion implantation by deriving scheduling constraints based on current equipment conditions. Consequently, a supervised learning pipeline is established that utilizes built-in sensors and process target data to accurately predict recipe transition costs. The derived constraints are integrated into scheduling, thereby enhancing its efficiency through dynamic dispatching adaptations. The application of our method is projected to significantly improve equipment availability by avoiding more than 100 hours of potential downtime annually.

3.1 Description of the Use-Case Pipeline

The implantation process requires precise control to ensure that ions are implanted into the semiconductor material at the correct depth and with the desired distribution, which in turn control the electrical properties of the manufactured device. For comprehensive information regarding the physical aspects of the implantation process, please refer to our deliverable D8.1.

We develop our solution for medium current implantation equipment. The built-in sensors and additional feedback values of implanter components are tracked via the Advanced Process Control (APC) system. Together, they provide us the current equipment condition. In the context of manufacturing, a recipe specifies parameters to meet the requirements of a process step. For ion implantation, the most critical parameters are energy, species, and dose. Whenever recipes are changed, a setup is necessary, i.e., ion beam tuning. This tuning is instrumental in minimizing deviations from recipe target specifications under varying equipment conditions. We develop predictive models that forecast the results of the tuning process in terms of its success and duration. The former being of utmost importance since it enables proactive scheduling adaptations to enhance equipment stability and utilization. By incorporating respective success predictions as (soft-)constraints, scheduling systems can proactively address potential tuning issues. This is achieved by avoiding the dispatch of lots to equipment that is currently not in the condition to tune for the associated recipes efficiently. Dynamic updates to the setup cost matrix with more precise estimates of tuning duration through regression analysis assist in identifying more efficient setup sequences.

To enhance equipment performance and process stability (uptime increase of >1 percentage point), we must integrate data engineering, Machine Learning (ML) modeling, and deployment within a robust Machine Learning Operations (MLOps) framework.

3.1.1 Employed Data Engineering, Preprocessing and Resulting Dataset

Since the beginning of the project, we have adapted our data engineering method multiple times to fit our current needs, as outlined in Deliverable D8.2. The resulting dataset has been published to foster research in this area via the Zenodo research repository[7] [1]. To not leak critical information about our processes and products, the data has been anonymized by scal-

[7] https://zenodo.org/records/11084332

ing data and not providing the scaler and by replacing the feature names with generic placeholders, e.g. sensor_1. Random generators cannot be aligned across programming languages easily, thus we avoid it by splitting the dataset into train and test sets, instead of providing one file with all the observations. As the preprocessing is already done for the purpose of anonymity, it is not required after loading the data into memory.

3.1.2 Modeling, Evaluation and Deployment

We made further advancements towards a productive implementation, which required another iteration of feature selection. We now only include features in our dataset that were carefully curated by implantation experts. Transitioning from the research phase to a dynamic production environment comes with its unique set of challenges. Lot scheduling and transportation takes time, therefore we need to minimize frequent shifts in our predictions, as changing plans incurs diverse costs. To achieve this, we removed certain features from the input and regularized our models adequately.

For the classification model, evaluation metrics such as the confusion matrix, F1 score, precision, and recall are analyzed. It is important to note that fluctuations in performance across various equipment units and time periods can be ascribed to volatility in the dataset.

Deployment practices have been guided by MLOps principles, underscoring the significance of containerization and orchestration for stable operation. We refactored our code to split the preprocessing, modeling and postprocessing into separate microservices, that can be called individually.

Postprocessing is currently being developed. It will prepare the output table, as initially described in Deliverable D8.2 Section 3.3.5 but will at the same time reduce the amount of data stored. This is achieved by only actually transmitting recipe transitions which are predicted to fail and by introducing wildcards (*) for recipe transitions that are generally predicted to fail. The updated interface concept is depicted in Table 3.1. Greyed and crossed out columns and rows are no longer necessary. Additionally, a row has been added symbolizing the wildcard approach. Moreover, it performs sanity checks on the model output, e.g., by checking the ratio of recipes to be deprioritized. If it exceeds a certain threshold on equipment-level, a maintenance activity is triggered automatically.

Table 3.1: Updated Interface Concept between Machine Learning and Scheduling System

Equipment	Current Recipe	Next Recipe	~~Duration (min)~~	~~Tune Success~~
~~**IMP01**~~	~~X1-80E3B020A~~	~~X5-00E2B170A~~	~~2.8~~	~~1~~
IMP04	X3-22E4P020A	Y1-00E2B400A	~~4.2~~	~~0~~
~~**IMP03**~~	~~Y1-31E3B050A~~	~~Y2-50E4B130A~~	~~3.1~~	~~1~~
~~**IMP01**~~	~~Y6-11E5A150A~~	~~Y6-71E2P285D~~	~~5.3~~	~~1~~
IMP05	*	Z5-66E2P020A	~~3.3~~	~~0~~
IMP02	Y8-05E2P285D	..	~~5.8~~	~~0~~

3.2 Benchmarking Setup

We implemented the core of the machine learning pipeline within DaphneDSL, aiming for comparable workflows in Python and DaphneDSL. We benchmark both computer languages

against each other based on the above-mentioned fixed dataset. Moreover, we compare machine learning model implementations for decision trees. The DecisionTree algorithm and others are publicly available within the DAPHNE source code repository[8], use-case details can be found in our private git repository shared with consortium partners (commit 90af0e73).

After setting up the UMLAUT framework from WP9 in our Python environment, we only need to assign decorators to Python functions. This enables benchmarking Python processes based on manifold metrics, such as time, memory, power, energy and CPU. Details are provided in Deliverable D9.3. We can also benchmark our DaphneDSL implementation in this way by invoking the DaphneDSL script execution via a Python subprocess.

For our tests we will use a single notebook (HP EliteBook x360 830 G6) and we also perform tests on Infineon's High-Performance Computing (HPC) cluster. Details about these environments are given in Table 3.2. Usage of HPC was initially limited, as outside connections are blocked by default. It only became possible after assistance from consortium partners, which jointly built DAPHNE binaries for RedHat Enterprise Linux (RHEL8). The obtained results of our experiments with DAPHNE (4e96943) using these environments are presented in section 3.3.

Table 3.2: Hardware and Software Environment Details of Experiments

Category	Notebook	HPC Node
Operating System	Windows 10 Pro 64-bit (10.0, Build 19045) with WSL2	Linux 4.18.0-477.36.1 (x86_64) RedHat Enterprise Linux 8
Processor	Intel Core i5-8365U @ 1.60 GHz (8 cores)	Intel Xeon CPU E5-2690 v4 @ 2.60GHz (28 cores)
RAM	8 GB (of 16 GB due to WSL)	512 GB
Python	Python 3.9 (scikit-learn 1.5.1)	Python 3.9 (scikit-learn 1.3.0)

3.3 Productivity & Performance Improvements

In this report we focus on the quantitative improvements achieved by the DAPHNE stack. We will delineate the runtime efficiencies and accuracy enhancements by directly comparing DaphneDSL with the Python implementation. To illustrate the productivity gains, we provide empirical data demonstrating the reduction in computational time and improvement in validation measures.

When benchmarking our DaphneDSL and Python pipelines, we focus on the following metrics:

- classification metrics: Accuracy, F1, Precision and Recall
- runtime,
- memory and CPU usage

[8] https://github.com/daphne-eu/daphne/tree/main/scripts/algorithms

On the notebook with WSL running DAPHNE (commit gc4c928af) with docker image (commit e17fe114a8c4), we observed the following metrics:

Table 3.3: WSL performance table

Metric / System	Python	DaphneDSL
Runtime (s)	6.34	237.47
Accuracy	0.74	0.68
Precision	0.77	0.78
Recall	0.68	0.50
F1 Score	0.72	0.61
Confusion Matrix	**0** **1** **0** 4299 1071 **1** 1687 3673	**0** **1** **0** 4613 757 **1** 2689 2670

Figure 3.1 and Figure 3.2 display the usage trends for CPU and memory within WSL.

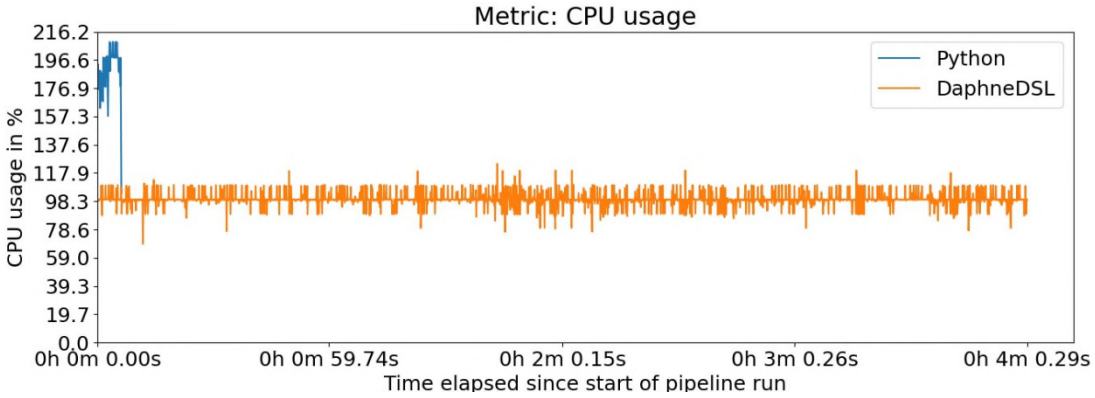


Figure 3.1: WSL CPU usage

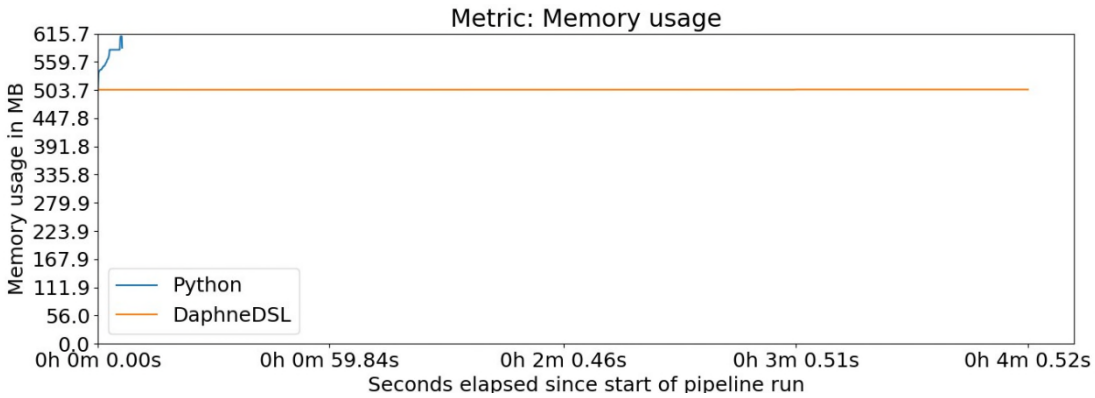


Figure 3.2: WSL memory usage

The runtime of DaphneDSL is shown to be about 40 times longer than Python. However, having to rely on a subsystem within Windows and due to other applications open on the notebook, the comparison is prone to errors. Yet, we wanted to showcase that there is an option to develop and test DaphneDSL scripts on a notebook running Windows as an operating system with the already existing resources and software artifacts.

For a more comparable benchmark, we ran experiments on our HPC with RHEL8 with DAPHNE (commit gc4c928af), where we observed the following metrics:

Table 3.4: HPC performance table

Metric / System	Python			DaphneDSL		
Runtime (s)	17.66			74.66		
Accuracy	0.74			0.69		
Precision	0.79			0.70		
Recall	0.65			0.67		
F1 Score	0.72			0.68		
Confusion Matrix		**0**	**1**		**0**	**1**
	0	4461	909	**0**	3810	1560
	1	1827	3533	**1**	1768	3591

Figure 3.3 and Figure 3.4 depict the CPU and memory usage within our HPC environment.

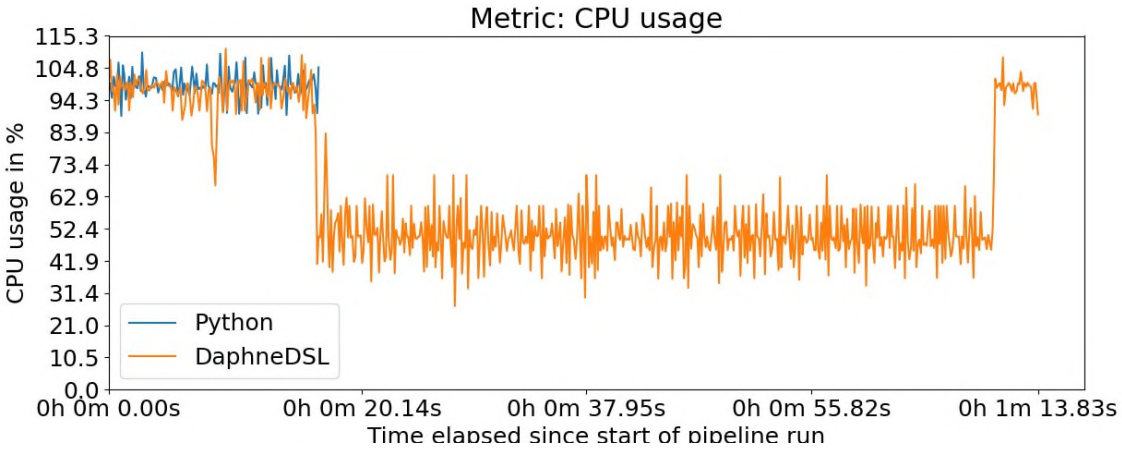


Figure 3.3: HPC CPU usage

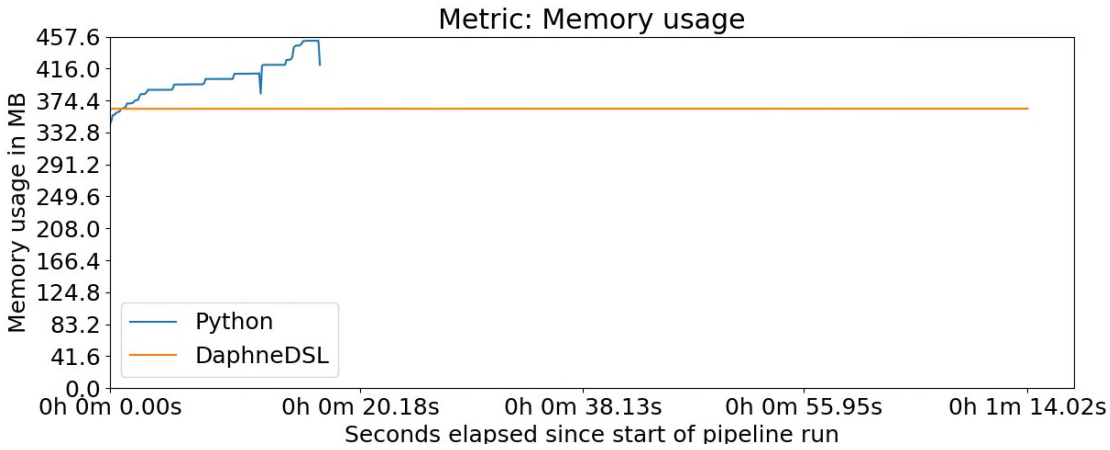


Figure 3.4: HPC memory usage

With the current implementation of Decision Trees, Python is faster by a factor of 4.

3.4 Conclusion and Outlook

Differences in classification metric are due to random initialization effects. For our WSL pipelines we used the default hyperparameters provided from scikit-learn. To enhance model performance, we regularized our HPC models by limiting the max_depth of a tree. This resulted in less diverging metrics between implementations.

When inspecting the CPU usage, DaphneDSL only uses roughly 50% of what is consumed by Python per timestep but also takes longer time to finish the overall computation. Memory usage appears to be static for DaphneDSL and is lower than what is requested from Python. Optimization possibilities to further speed up the execution time of decision trees within DaphneDSL have been identified and are being worked on. We expect to see a 10x to 15x improvement, which would result in DaphneDSL requiring less than 50% of the runtime observed with Python.

As for the next steps, we will finish deployment of our productive solution in our manufacturing environment. Moreover, we want to verify the above-mentioned speed-up of the decision tree implementation in DaphneDSL. Simultaneously, we will strive to complete the implementation of a second model for our pipeline, which calculates classifications based on a Multi-Layer-Perceptron (MLP) – a neural network with three hidden affine layers, two distinct activations (reLu, sigmoid) and one loss (log_loss) function together with an optimizer (Adam) for a faster learning process.

4 Material Degradation Case Study (KAI)

Within the KAI Material Degradation Case Study, we differentiate between two use cases. On the one hand, there is the line simplification use case (UC4.1) and on the other there is the RUL (remaining useful lifetime) prediction use case (UC4.2). Our data is represented as time series originating from accelerated stress test systems which measure the electrical behavior of the devices under test (DUT) while being stressed. The purpose of the line simplification use case is to reduce all the time series down to the most significant sample points to allow for finite-element simulation of the thermo-electrical behavior of the DUTs. In parallel, during the RUL prediction use case, we develop an approach to predict the remaining lifetime of a DUT. Put another way, the model predicts how soon in the future a device will fail. Deliverables D8.1 and D8.2 provide more details on both use-cases. Section 4.1 additionally offers a brief recap of the respective pipelines.

4.1 Description of the Use-Case Pipelines

This section gives a brief recap of the pipelines of our two use cases and additionally points out the progress since deliverable D8.2.

For both pipelines we hold a Python implementation which acts as a baseline and a DAPHNE DSL (domain specific language) implementation.

The Python implementation of the line simplification pipeline makes use of NumPy[9], pandas[10] and a proprietary implementation of the line simplification algorithms. For the RUL prediction pipeline, we make use of PyTorch[11] and TorchMetrics[12].

4.1.1 Line Simplification Pipeline

First, let us look at the line simplification pipeline. At its core is a so-called line simplification algorithm coming from cartography is reducing the time series in a lossy way. Meaning that from 768 sample points of a single time series, only about 20 are retained. The rest is discarded. At KAI, we make use of two different line simplification algorithms, the Visvalingam-Whyatt algorithm (VW) [2] and the Ramer-Douglas-Peucker algorithm (RDP) [3]. These algorithms optimize the data reduction with a different target metric. The benchmarking efforts were put into the RDP algorithm. Hence, this document focuses on the RDP algorithm.

Since deliverable D8.2, there are minor adaptions. We changed the data CSV file representation originally introduced in D8.2 Section 4.2. Instead of storing one CSV file for each test cycle, we switched to storing the values of one measured variable of all test cycles in a single file each. This tremendously reduces the number of CSV files from several thousands to just three. Such approach thereby leads to fewer file I/O operations, in turn increasing the memory footprint since all data of a test run is loaded into memory as a whole.

[9] https://numpy.org
[10] https://pandas.pydata.org/docs
[11] https://pytorch.org
[12] https://lightning.ai/docs/torchmetrics/stable

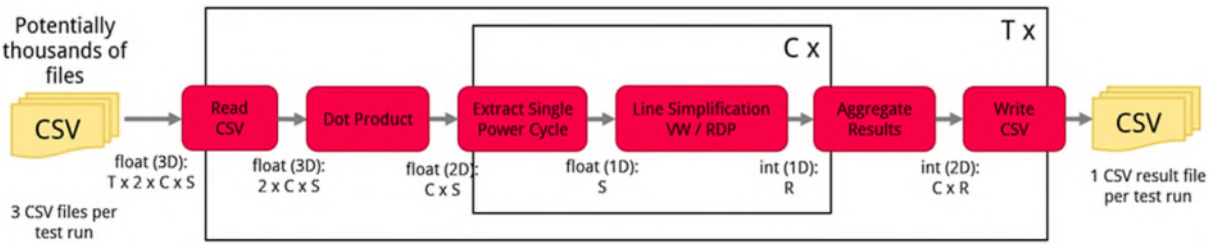


Figure 4.1: Line simplification pipeline. The red color indicates that the pipeline component is implemented with DAPHNE. (T = number of test runs, C = number of cycles in test run, S = number of sample points per cycle, R = number of indexes after reduction)

Figure 4.1 depicts the final line simplification pipeline. First the data is read from CSV files. Per DUT there are three CSV files, where one contains the electrical current measurements, the second contains the voltage measurements and third one the relative time of the sample points. Afterwards as a preparation step, the electrical current vector is multiplied (dot product) with the voltage vector to obtain the power vector. The latter is forwarded to the RDP algorithm implementation. The results are represented as a list of indexes. They are aggregated from all test cycles into a matrix and written together into a CSV file with a single write operation at the end.

4.1.2 Remaining Useful Lifetime Prediction

Our second use case, the RUL prediction pipeline is modeled as a regression problem. A deep ML model predicts a value between 1.0 and 0.0. A value 0.9 indicates that the device still has 90% of its lifetime left and 0.1 indicates that only 10% lifetime are left (wear out). Since wear means that the device behavior and respectively the measurement data will change over time, the model needs to be able to learn this. To map time dependency into the input data, we stack a specific number of subsequent cycles into a tensor forming a window.

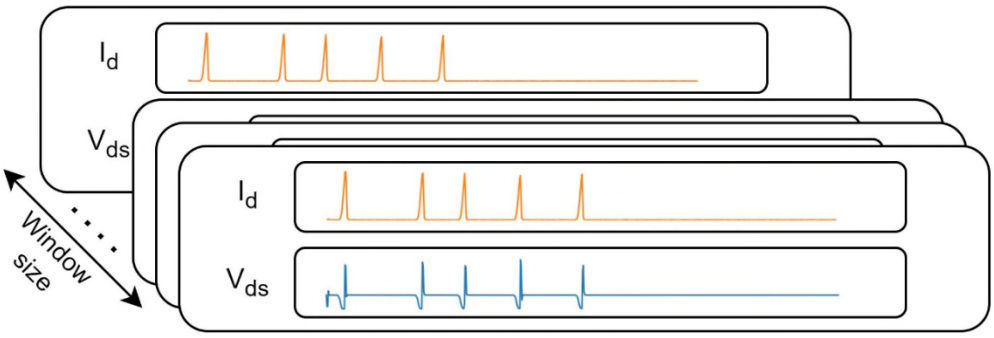


Figure 4.2: The input data format for the RUL prediction model

Figure 4.2 presents the input data format. A test cycle consists of a current waveform (Id) and a voltage waveform (Vds). These two arrays are stacked to a matrix. Several cycles in turn are stacked to a tensor. Increasing the start index of the tensor forms a moving window. For training purposes, the windows are sampled randomly. This approach enables the usage of CNNs.

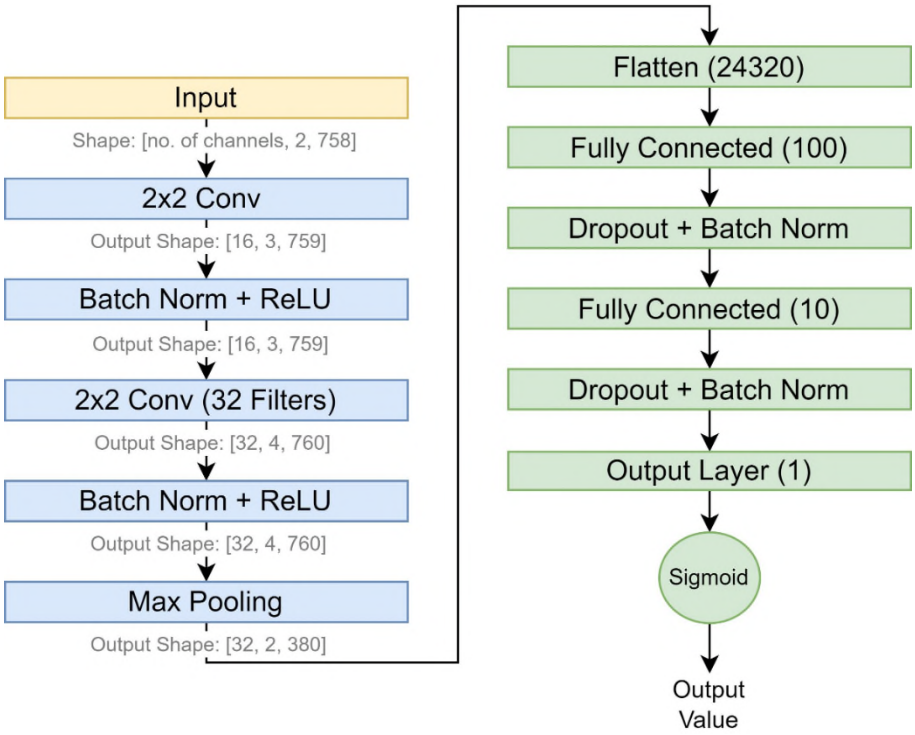


Figure 4.3: The architecture of the CNN model

In Figure 4.3, the layers of the CNN model are shown. As you can see, default CNN layers are used for feature extraction. Several dense layers solve the regression task. *no. of channels* stands for the window size (number of stacked cycles).

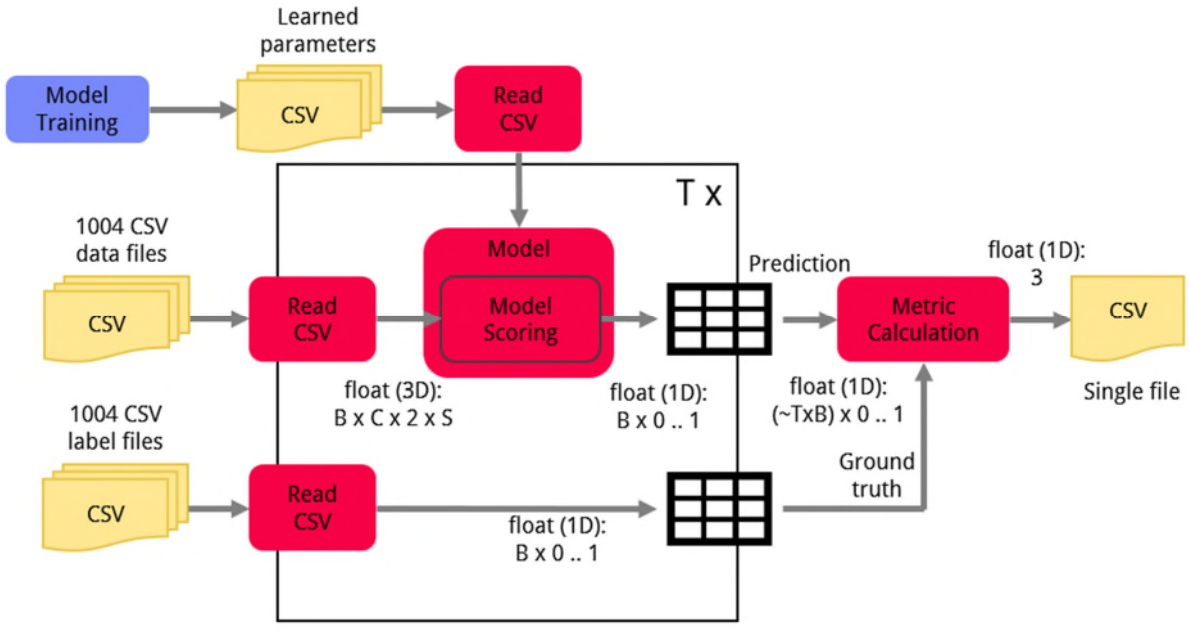


Figure 4.4: RUL prediction pipeline (inference). The red color indicates that the pipeline component is implemented with DAPHNE. The lavender color indicates that the respective component was running with another system (here PyTorch). (T = number of batches (1004

Figure 4.4 shows the RUL prediction pipeline. During the implementation of this pipeline, DAPHNE was not yet providing the backwards passes of the NN functions. Hence, it was not actually possible to perform model training with DAPHNE back at that moment. To bypass this,

we reused the learned parameters from PyTorch by exporting them into CSV files and loading them into DAPHNE.

The input data has a CSV representation as well. It already contains the prepared batches with 64 windows of a test run. This was done with a Python script which translated the original dataset from PyTorch `.pt` files into the described representation.

After loading, the pipeline forwards the batches to the model for inference. All predictions are stacked in a temporary matrix. After inferring all batches, the respective ground truth is also loaded and stacked. The prediction matrix and the label matrix are both fed into the three error metric functions specified in 4.2.2. In the end the final metric values are stored in a CSV file.

For the RUL prediction pipeline, the DAPHNE DSL implementation differs substantially from the Python baseline implementation. Python offers OOP and PyTorch makes use of this at several points. For our Python ML pipeline, we have a single cycle stored in `.pt` files (in contrast to a whole batch per CSV file). The windows and batches are then loaded and composed dynamically during runtime by leveraging PyTorch's `Dataset` and `DataLoader` classes. For better comparison the parameter `num_workers` of `DataLoader` is set to `0` which avoids data loading in parallel. Additionally, we set the number of threads for PyTorch to `1`. Although PyTorch allows for tracking the metrics on the fly, we still stack all predictions and calculate the error metrics as one batch for better comparability. TorchMetrics is used to calculate the error metrics.

4.2 Benchmarking Setup

This section describes the benchmarking setup for the KAI use cases. It describes which metrics are used to compare the baseline implementation with the DAPHNE implementation and how the metrics are measured.

4.2.1 Line Simplification Benchmark

In order to benchmark the line simplification pipeline, we extracted a subset of the KAI dataset consisting of 60 test runs (specific test types of lower interest were filtered out). This means the dataset counts an overall of 180 CSV files (and another 180 DAPHNE specific `.csv.meta` files). It has a total of ~1.3 million cycles to reduce and ~17 GB in aggregated file size. Depending on the benchmarking purpose we only make use of a part of this data.

The line simplification pipeline is a pure CPU workload. We decided to evaluate the following performance metrics:

- Runtime (scalar)
- CPU utilization (time series)
- Memory usage (time series)

We are running the experiments on our company internal High-Performance Computing (HPC) cluster. Specifically, the experiments are conducted on a compute node with an Intel Xeon Gold 6242 CPU (2.80 GHz) with 32 cores with a Red Hat Enterprise Linux (RHEL) 8 installation. IBM

Spectrum LSF[13] provides job queueing, submitting and scheduling. Our internal IT provides an in-house developed extension of LSF for tracking benchmarking metrics of LSF jobs.

4.2.2 Remaining Useful Lifetime Prediction Benchmark

For the data science experiments, we extracted a subset of 130 test runs. This filtering was necessary to compose a homogenous dataset with comparable device types and test conditions. The selected data comprises 1004 files as CSV representation for DAPHNE and it counts ~307,000 files (and test cycles at the same time) as PyTorch binary file representation for Python. The CSV files have 51 GB and the PT files 2.4 GB in overall file size. The significant difference of the dataset size roots from data duplication in the CSV representation since whole windows are stored and windows do have an overlap. However, the PT files are read out many times instead of only once.

Because the RUL prediction use case pipeline makes usage of a deep CNN model, GPU kernels are leveraged. Still, the data loading is performed by a CPU. The following performance metrics are selected:

- Effective end to end runtime (scalar)
- CPU utilization (time series)
- Memory usage (time series)
- GPU utilization (time series)
- GPU memory usage (time series)
- GPU power consumption (time series)

Additionally, the performance of the ML model is measured with the following error metrics:

- Mean absolute error (MAE) (scalar)
- Root mean squared error (RMSE) (scalar)
- Coefficient of determination (R^2) (scalar)

The model error metrics are calculated as described in 4.1.2. All experiments were conducted on a machine with two Intel Xeon Gold 6238R (2.20GHz) CPU, a NVIDIA Tesla T4 GPU and an Ubuntu 20.04.6 LTS installation. The performance metrics are obtained with a Python package named gpustat[14]. This package uses NVIDIA's official Python bindings for the NVML library[15].

4.3 Benchmarking Results

This section shows the results of the benchmarking experiments, comparing the DAPHNE DSL implementation with the Python baseline implementation.

4.3.1 Line Simplification Experiments

The following DAPHNE version was used for the experiments: commit `bed9bcde` at `daphne-eu/daphne/main` on 2024-08-08. The DAPHNE binary was executed with default arguments (`daphne myScript.daphne`).

[13] https://www.ibm.com/docs/en/spectrum-lsf
[14] https://github.com/wookayin/gpustat
[15] https://pypi.org/project/nvidia-ml-py

To compare the runtime between the DAPHNE and the Python baseline, the whole benchmarking dataset of 60 test runs is used. Both implementations processed the exact same amount of data.

Table 4.1: Runtime of the line simplification use case of the DAPHNE DSL implementation and the Python implementation on the exact same dataset

	Python	**DAPHNE DSL**
Runtime	3h15m26s	61m54s

Table 4.1 shows that the runtime of the DAPHNE DSL script is about 3.16 times faster than the Python script.

For tracking the CPU utilization and the memory footprint, we chose a differing dataset size to achieve an hour of runtime for both implementations. The benchmarking tool only stores the sample points if changes occur.

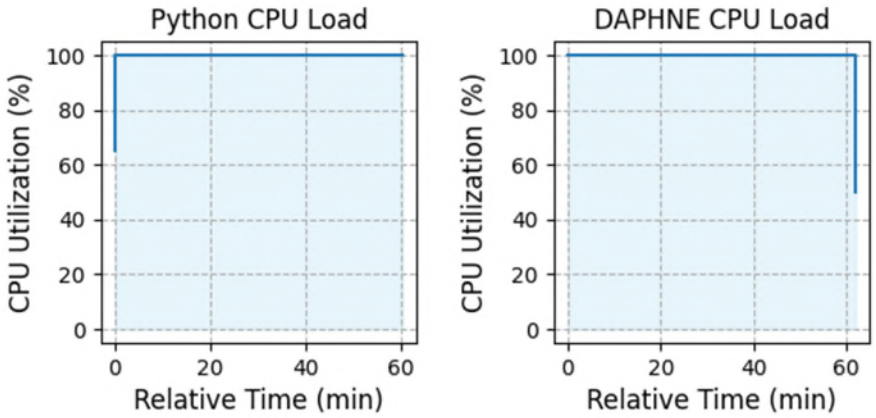


Figure 4.5: CPU utilization of the Python implementation and the DAPHNE DSL implementation of the line simplification pipeline

Figure 4.5 shows the CPU load. Both implementations fully utilize one CPU core.

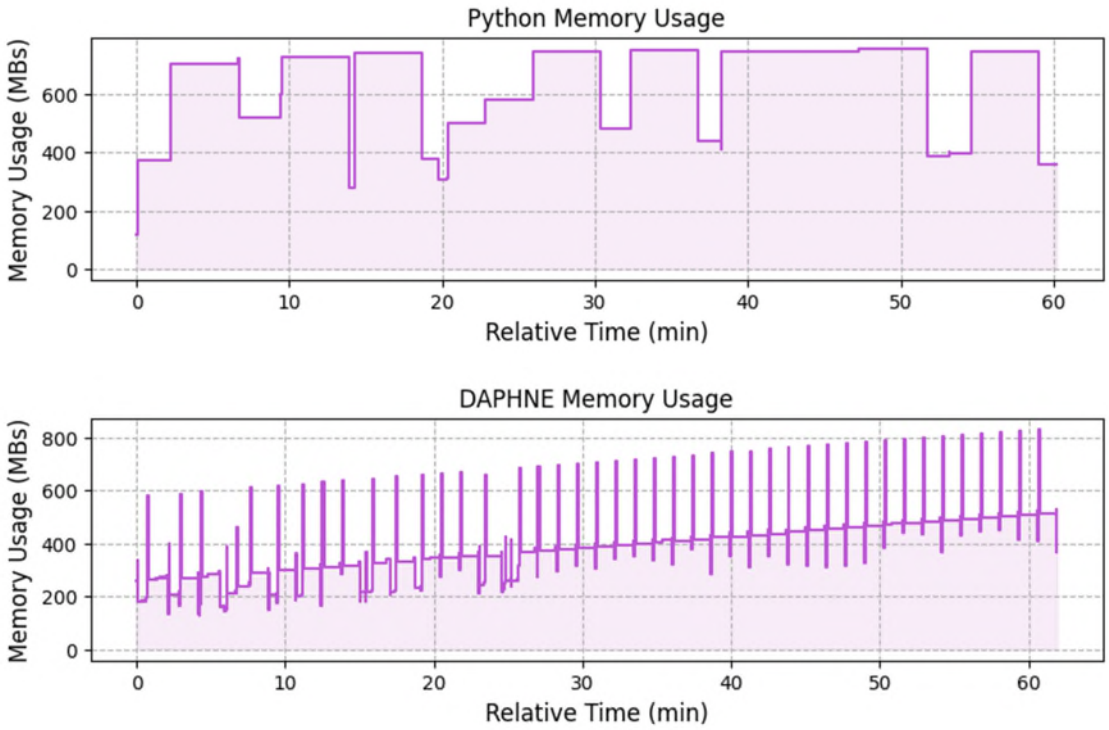


Figure 4.6: Memory usage of the Python implementation and the DAPHNE DSL implementation of the line simplification pipeline

Figure 4.6 depicts the memory usage (resident set size) of both implementations. The line simplification iterates over many CSV files. First it loads a file, then processes the data and finally writes the result. As can be seen, the Python implementation frees the memory which is used up within a loop iteration and not of relevance for the next iteration anymore. DAPHNE uses less memory overall. The spikes show a fast allocation and freeing of memory. As can further be seen in Figure 4.6, the DAPHNE implementation shows a small memory leak: It needs a slightly increasing amount of memory with each iteration because not all disposable objects are freed. We have reported this issue to the technical partners who are working on the bug fixes.

4.3.2 Remaining Useful Lifetime Prediction Experiments

The following DAPHNE version was used for the experiments: commit with hash `18833321` at `corepointer/daphne/dnn-ops` on 2024-07-29. The DAPHNE binary was executed with the arguments, `daphne --cuda --config UserConfig.json myScript.daphne`, mainly to ensure the usage of GPU kernels.

Table 4.2: Runtime of the RUL prediction use case of the DAPHNE DSL implementation and the Python implementation on the exact same dataset

	Python	**DAPHNE DSL**
Runtime	7m53s	11m27s

In Table 4.2, we can see that for the RUL prediction pipeline, the DAPHNE script needs ~150% of the runtime of the Python script.

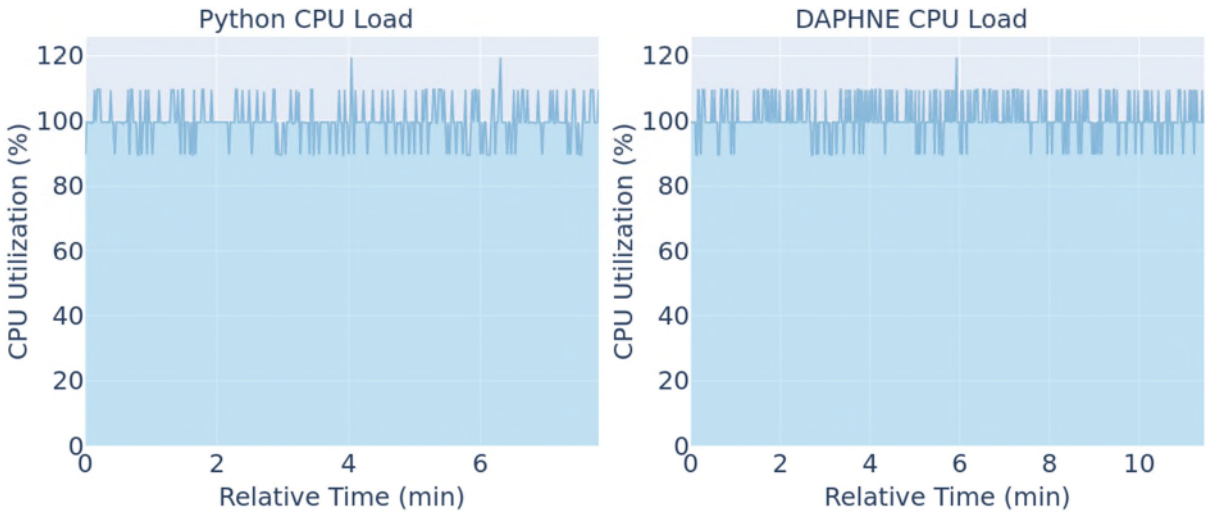


Figure 4.7: CPU utilization of the Python implementation and the DAPHNE DSL implementation of the RUL prediction pipeline

In Figure 4.7, the CPU load can be seen. Both implementations fully utilize one CPU core.

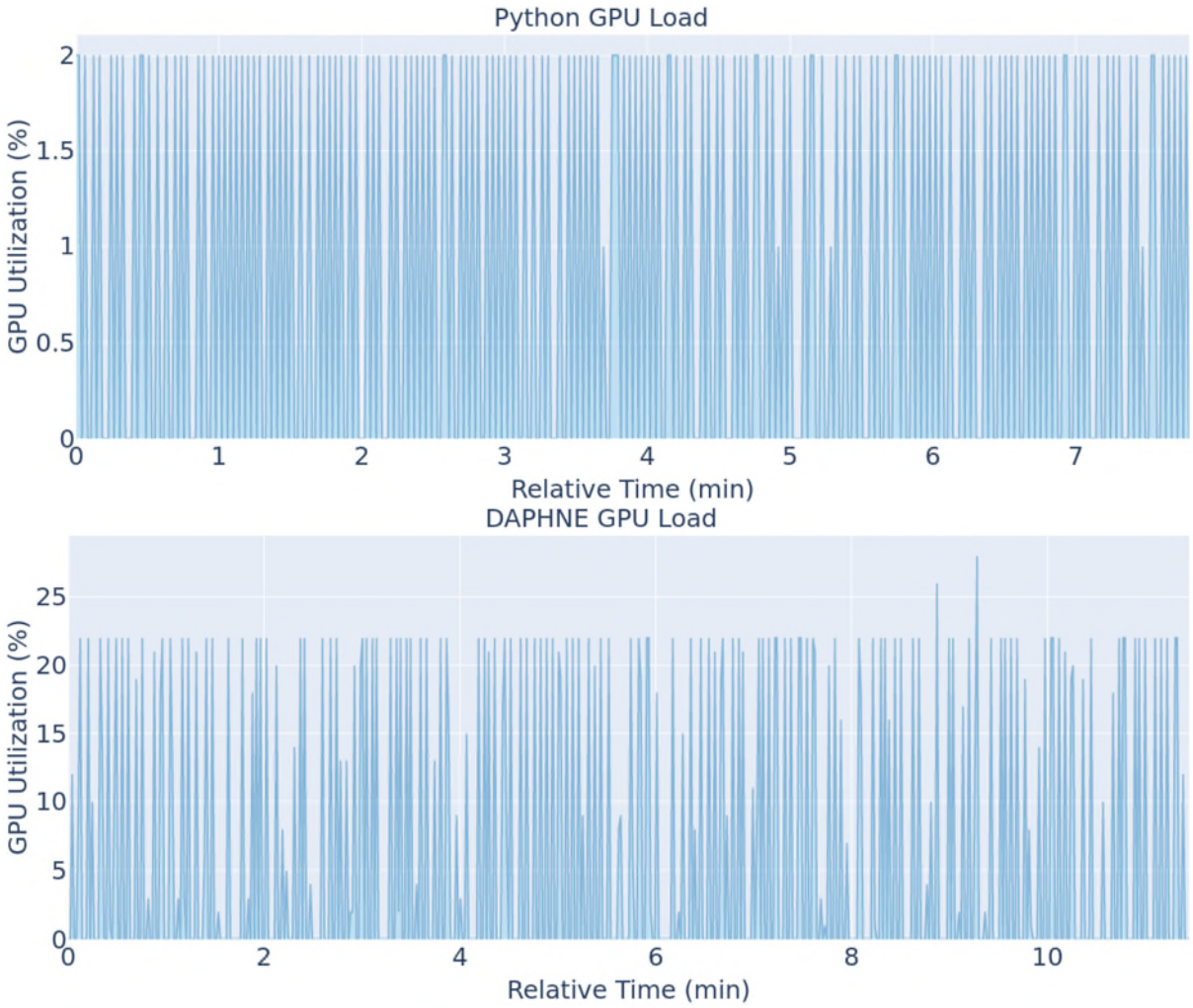


Figure 4.8: GPU utilization of the Python implementation and the DAPHNE DSL implementation of the RUL prediction pipeline

Figure 4.8 shows the GPU utilization. One pulse represents the processing of one batch or in other words one inference pass. This figure shows that the DAPHNE implementation puts a very fluctuating load onto the GPU although the batch size is kept constant. Furthermore, the peaks are more than ten times higher.

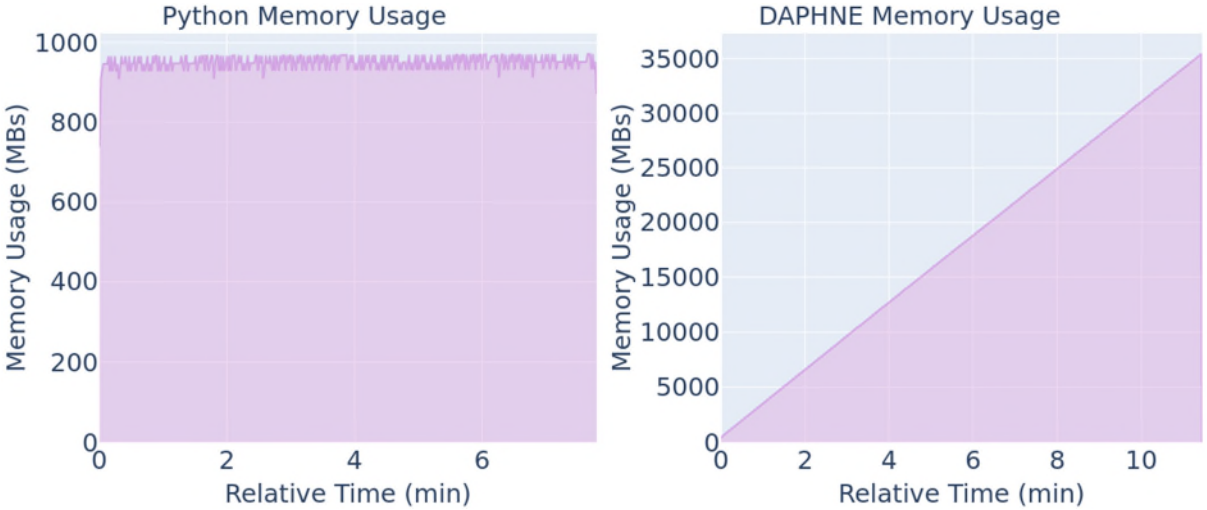


Figure 4.9: Memory usage of the Python implementation and the DAPHNE DSL implementation of the RUL prediction pipeline

Figure 4.9 depicts the memory footprint of the RUL prediction pipeline. The DAPHNE implementation shows a continuous increasing memory usage. This result was retrieved with an earlier version of DAPHNE. Most of the reasons leading to memory leaks are already fixed as can been seen in Figure 4.6. Since performing this experiment is connected to more effort on our side, there was not enough time left between the memory leak fixes in the past month and the deliverable deadline to rerun the benchmarking experiment.

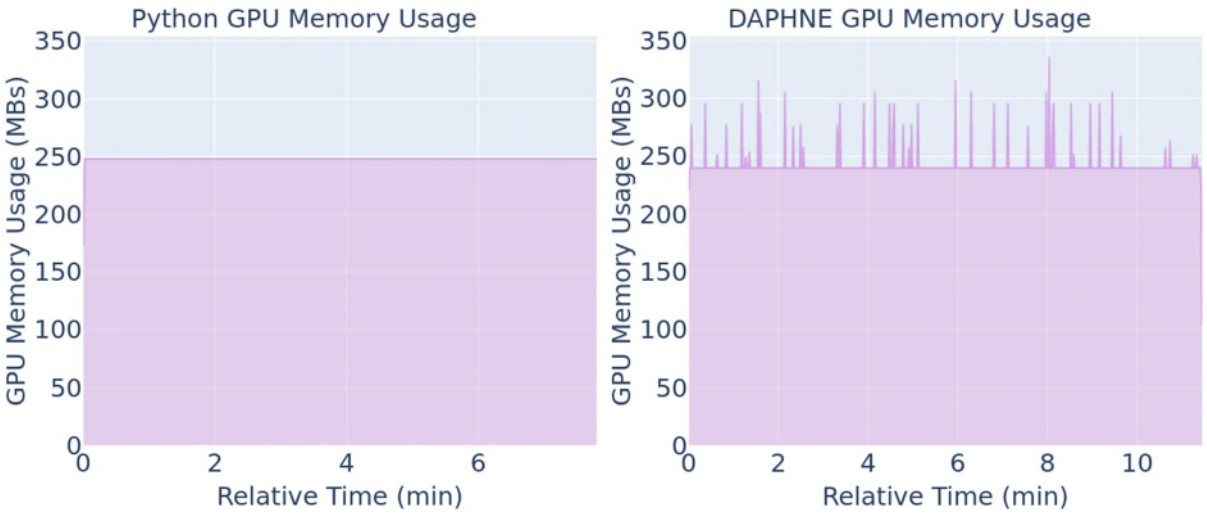


Figure 4.10: GPU memory usage of the Python implementation and the DAPHNE DSL implementation of the RUL prediction pipeline

Figure 4.10 shows the GPU memory usage. The behavior of both implementations is very similar.

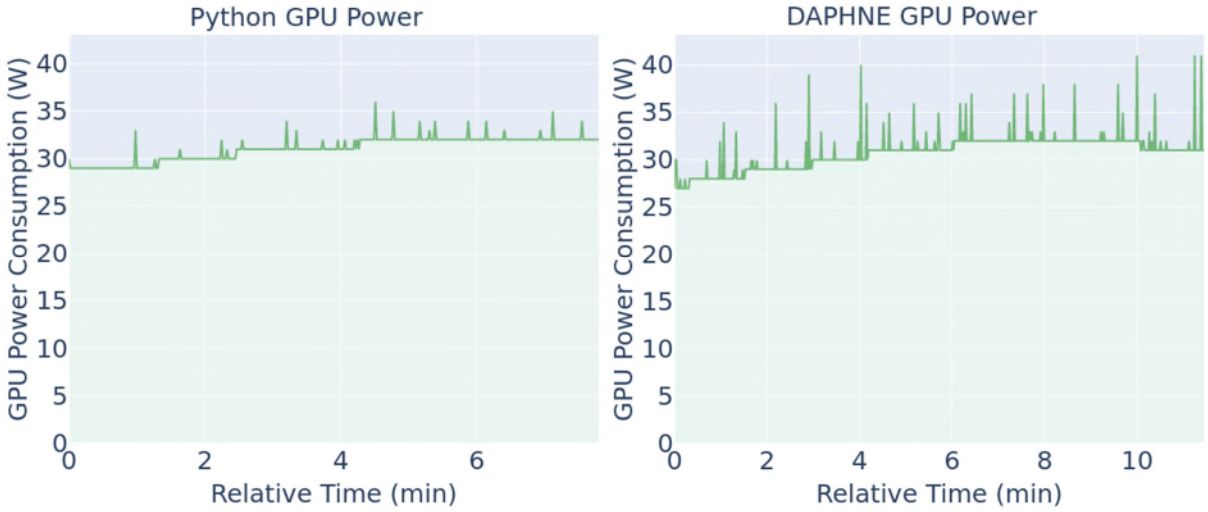


Figure 4.11: GPU power draw of the Python implementation and the DAPHNE DSL implementation of the RUL prediction pipeline

The plot in Figure 4.11 shows the GPU power draw. DAPHNE draws slightly less power but since the runtime is longer it consumes more energy.

Table 4.3: Inference scores of the PyTorch model and the DAPHNE model of the RUL prediction use case

	Python	**DAPHNE DSL**
MAE	0.051	0.051
RMSE	0.071	0.071
R2 Score	0.937	0.932

Table 4.3 shows the MAE and RMSE metrics as well as the R2 score. The values coincide substantially. Since the use case reuses already learned parameters, similar results are expected.

4.4 Benchmark Conclusion of KAI Experiments

For some tracked benchmarking metrics, the behavior of the DAPHNE DSL and Python are similar (CPU load, GPU memory usage, GPU power draw). Significant differences could be observed for the memory usage and the GPU load. Performance benchmarks showed that DAPHNE was suffering from memory leaks. Investigations with the analysis tool Valgrind[16] revealed that the memory leaks were caused by string concatenations, file input/output and kernel casting (this can still be observed in Figure 4.9). All three have been fixed in the meantime. It remains to find the reasons for the smaller leaks. Regarding the GPU load, the investigations are still running.

For pure CPU loads, like the line simplification pipeline, DAPHNE is already several times faster than the respective Python implementation. When it comes to GPU load, DAPHNE cannot yet keep up completely with PyTorch. The latter is a very mature and dedicated framework for GPU loads and hence hard to beat. Anyway, DAPHNE aims to improve the performance of heterogenous loads.

[16] https://valgrind.org

5 Automotive Vehicle Development Case Study: Ejector Geometry Optimization (AVL 1)

Within the DAPHNE project, two major improvements have been implemented:

- Active DOE Workflow (see D8.2 – section 6.2)
- Machine Learning & Optimization Python Implementation (see D8.2 – section 6.3)

The active DOE (Design of Experiments) implementation replaces all manual steps of the pipeline, enhancing productivity and minimizing throughput time. This improvement is achieved through a Python script that automates all previously manual steps. As a result, engineers can focus on analyzing results and understanding key concepts, such as those caused by flow separation.

In the initial pipeline, the entire Machine Learning and optimization process was handled by AVL's in-house software, CAMEO. To diversify the pipeline and enable the implementation of the DaphneLib Python API, an alternative option was developed using the Python ecosystem. The Python implementation expanded the range of available Machine Learning architectures, overcoming the initial limitations posed by CAMEO. This capability to test a broader set of Machine Learning architectures proved to be a significant advantage resulting from the DAPHNE project. By implementing a Kernel Ridge Regression model, the initial layout of an ejector for a robustness test operating condition was improved by approximately 60% (see Figure 5.1).

These enhancements significantly streamline the workflow, allowing for more efficient and effective analysis and optimization processes.

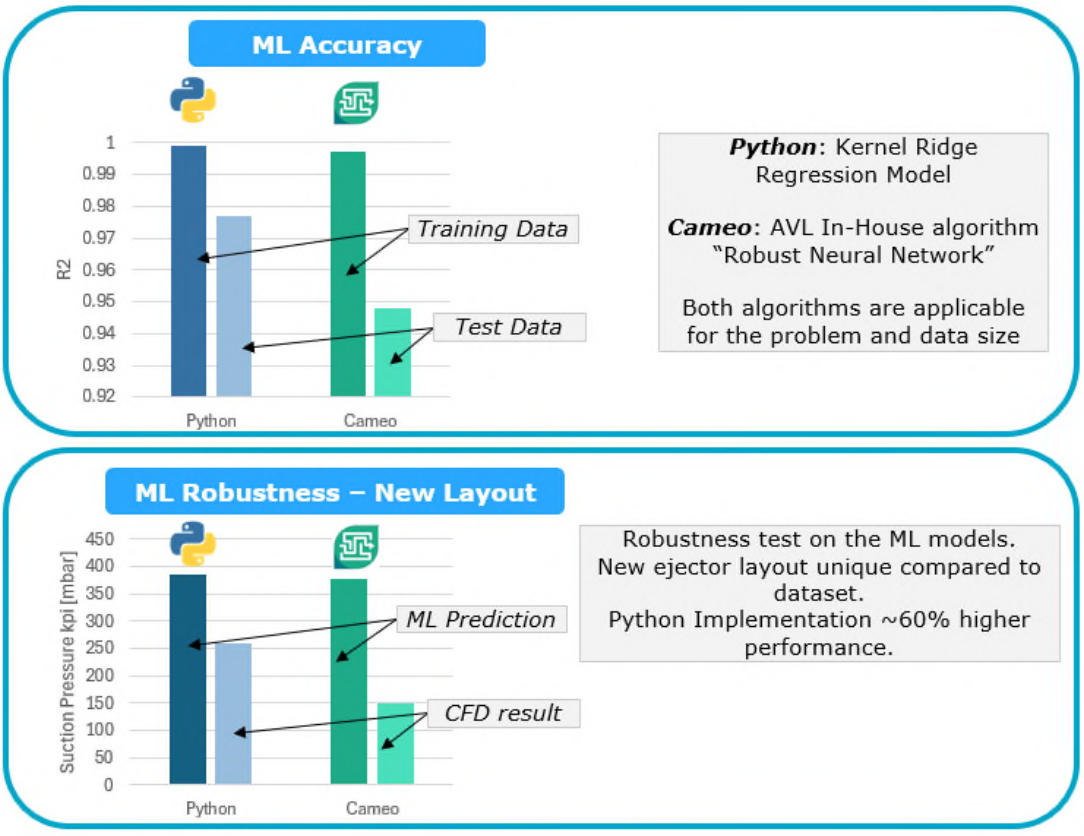


Figure 5.1: Kernel Ridge Regression vs. Robust Neural Network

5.1 Description of the Use-Case Pipeline

The AVL Ejector Use-Case pipeline is designed to optimize the design of an ejector com-ponent for fuel cell applications. An ejector replaces a blower to increase overall efficiency by utilizing high-pressure gas available in tanks, thereby enhancing the system's performance.

In the ejector pipeline, the first step involves performing a data-driven initial layout of the component to meet performance requirements for specific operating conditions, such as a new customer project. The performance prediction is achieved through 3D CFD (Computational Fluid Dynamics) simulation. For this, the initial layout must undergo meshing, a process that spatially discretizes the fluid domain. Additionally, the simulation setup is generated based on the specified operating conditions. Following this, a finite volume-based 3D CFD simulation is executed, and the results are post-processed for further analysis.

After completing the initial simulation loop, the performance results are compared against the requirements of the system under investigation. If the target performance is achieved, the de-sign process can be concluded, or further fine-tuning can be carried out to optimize the design. If the target is not met, Machine Learning models are retrained with the new data. The geom-etry is then optimized to enhance performance, and approximately 20 new geometric variants are generated. This initiates a loop of geometric variant creation, simulation, post-processing, Machine Learning training, and optimization, which continues until the performance require-ments are satisfied.

The DAPHNE Library (DaphneLib) is integrated into the Python Machine Learning workflow at the prediction function stage. During optimization, this function is called approximately 10,000 times using a genetic optimization algorithm (optimizer block in Figure 5.2) to evaluate a pop-ulation size of around 1,000 candidates. This results in a total of 10 million evaluations, ensuring a thorough and comprehensive optimization process.

These detailed steps ensure that the design and optimization of the ejector component are both efficient and effective, ultimately leading to enhanced performance and reliability in fuel cell applications.

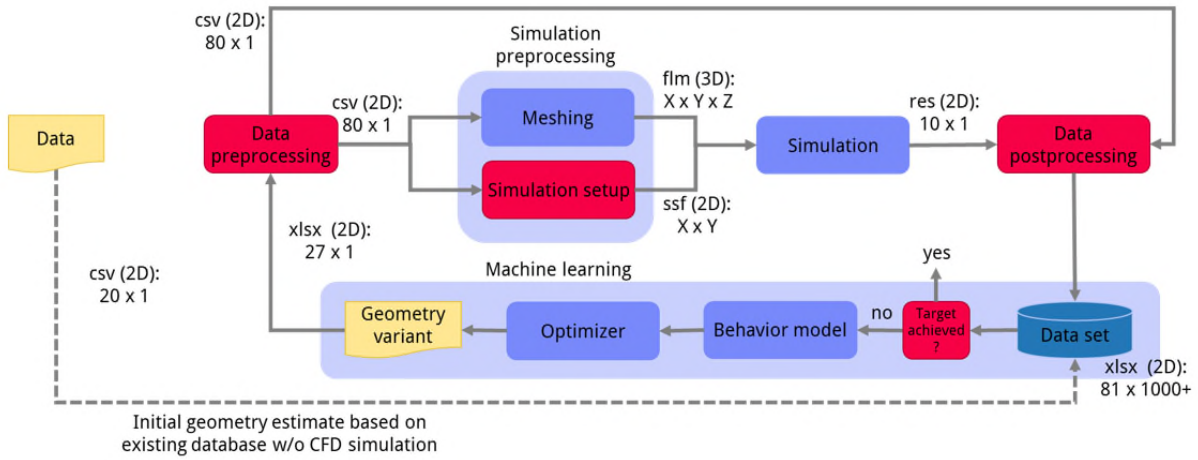


Figure 5.2: Ejector dimensioning pipeline

5.2 Benchmarking Setup

For benchmarking, we utilized a Dell Precision 7865 workstation equipped with 128GB of RAM and a 16-core AMD Ryzen Threadripper PRO 5955WX processor. To ensure accurate results, all other productive tasks were halted to free up the CPUs for benchmarking purposes.

The benchmarking process employed the UMLAUT framework, which was developed as part of the DAPHNE project. UMLAUT is applied to the Python scripts by simply adding a decorator to the function that is to be benchmarked. UMLAUT allows for seamless integration and easy setup for performance analysis.

An instance of the benchmark object is created to define the parameters we want to analyze, such as RAM utilization and CPU utilization. The results of the benchmarking process are stored in a SQL database. Access to these results is facilitated by a shell script provided by the UMLAUT framework, which can be found at UMLAUT Framework GitHub Repository.

By utilizing this comprehensive benchmarking setup, we can accurately assess and optimize the performance of our systems, ensuring they meet the required standards and per-form efficiently under various conditions.

The benchmarking takes place with a sample rate of 0.01 s and 1.5 s

5.3 Productivity & Performance Improvements

The productivity improvements of the overall development pipeline were documented in deliverable 8.1 and 8.2 and covers the following bullet points:

- Active DOE Workflow (see D8.2 – section 6.2)
- Machine Learning & Optimization Python Implementation (see D8.2 – section 6.3)

The implementation of DaphneLib at the application level was straightforward and user-friendly. This ease of integration was a key factor in our decision to choose DaphneLib over a direct implementation of DaphneDSL.

This section will focus on the performance comparison between the stand-alone python implementation and a DaphneLib supported python workflow.

5.3.1 DaphneLib Implementation

To generate predictions using a Kernel Ridge Regression model, the following calculations are required:

- Train the model with the dataset to generate:
  - Support vectors (feature vectors from the dataset)
  - Alphas (model weights)
- Compute the difference matrix between the support vectors and new feature vectors for prediction.
- Calculate the L1 norm of the difference matrix.
- Input the norm into the kernel function and multiply it by the model parameter gamma.
- Obtain predictions via the dot product of alphas and the kernel matrix.

For this purpose, the prediction function is implemented in Python in two ways: once using the DaphneLib and once without it. This dual implementation aims to provide a fair comparison.

DaphneLib is a Python API that shifts computations to DaphneDSL, enhancing computational performance. Although, in this use case, the primary computational load lies in the 3D-CFD simulation rather than the Machine Learning and Optimization components. Nonetheless, the DAPHNE ecosystem, particularly the DaphneLib Python API, can streamline the Python workflow.

On the Optimization and Machine Learning front, the primary computational burden is related to predictions within the genetic algorithm used to optimize the feature vector. For instance, running 1,000 generations with a population of 1,000 feature vectors results in 1 million evaluations.

5.3.2 Benchmarking Results

The pipeline is benchmarked with a fixed number of 1,000 generations, varying the number of feature vectors in each generation (population size) across three scenarios: 100, 1,000, and 10,000 feature vectors. The following figure presents a comprehensive comparison of runtime for these three different population sizes:

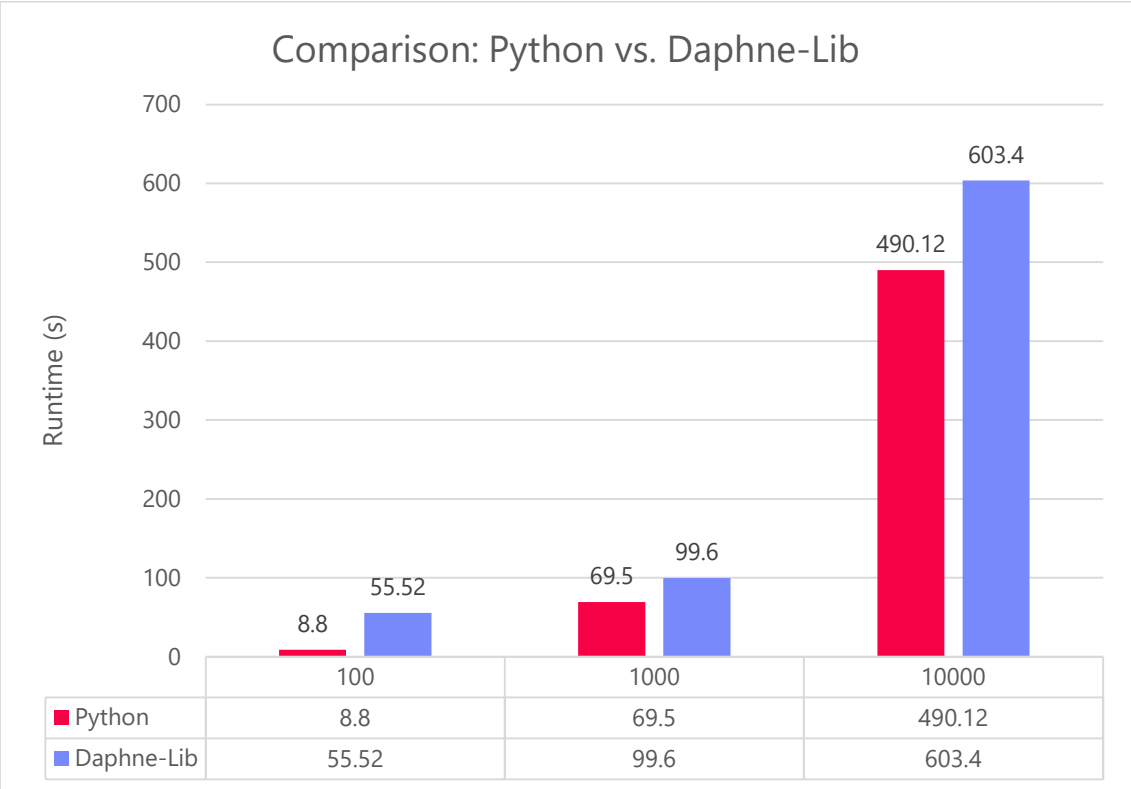


Figure 5.3: Runtime comparison

The DaphneLib implementation demonstrates increased speed compared to pure Python as the number of evaluations rises. The DaphneContext() instance is computed once per generation utilizing the DaphneLib function `DaphneContext.from_numpy()` for data exchange from Python towards DAPHNE. At smaller feature vector sizes, the overhead associated with shifting computations from NumPy and Python to DaphneDSL is more pronounced. However, as the feature vector sizes grow with increased generation sizes, this overhead becomes less significant. This indicates that a full DaphneLib implementation of the optimizer could be a viable option to minimize overhead and enhance performance.

The next sections will show runtime, CPU utilization and memory allocation for the three scenarios of different population sizes.

5.3.2.1 Population size 100

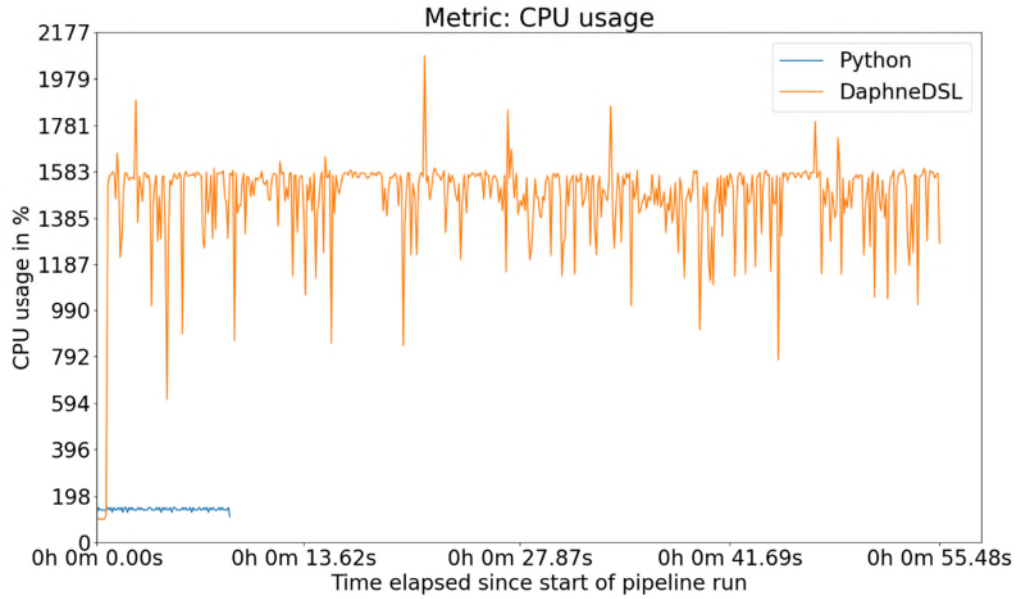


Figure 5.4: CPU utilization – population size 100, sampling 0.1s

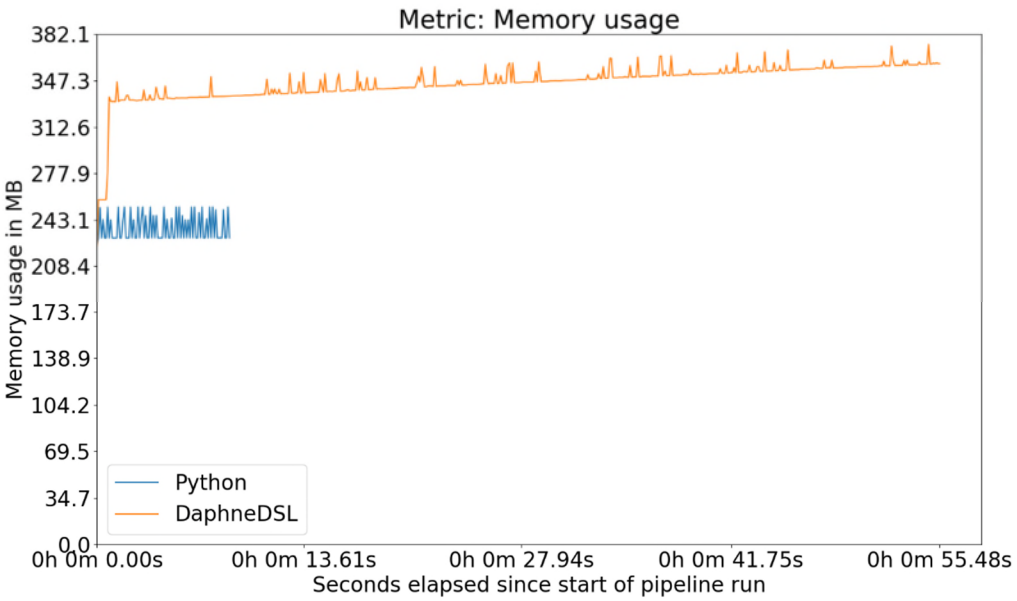


Figure 5.5: Memory allocation – population size 100, sampling 0.1s

Note that the plain Python implementation does not utilize all available CPUs, but DAPHNE does. Figure 5.4 indicates that the faster execution in Python is resulting from parallelization overhead associated with DaphneLib execution. There is an indication of a memory leak when utilizing DaphneLib and it likely significantly slows down computation.

5.3.2.2 Population size 1000

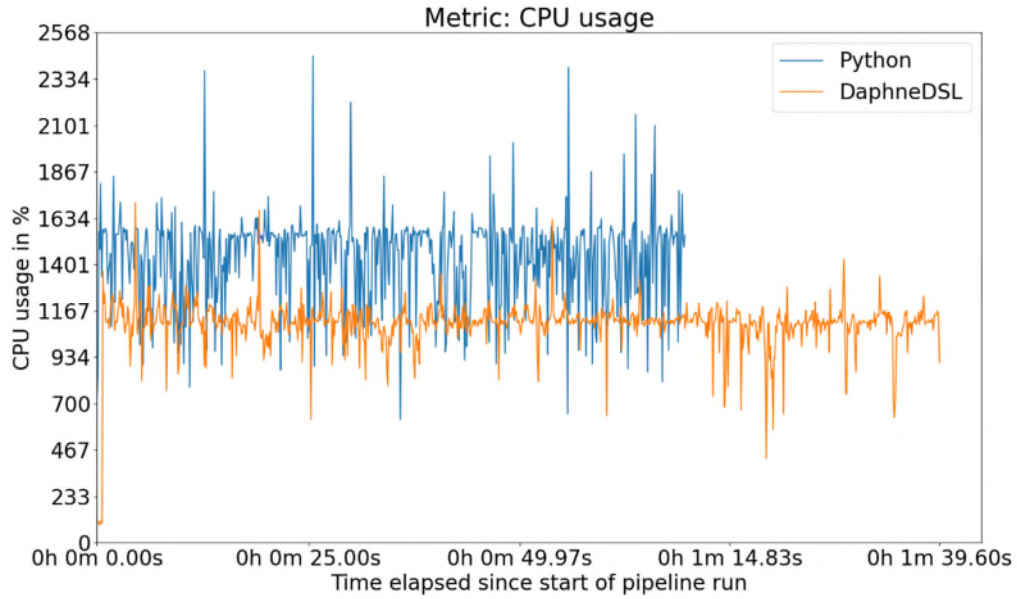


Figure 5.6: CPU utilization – population size 1000, sampling 0.1s

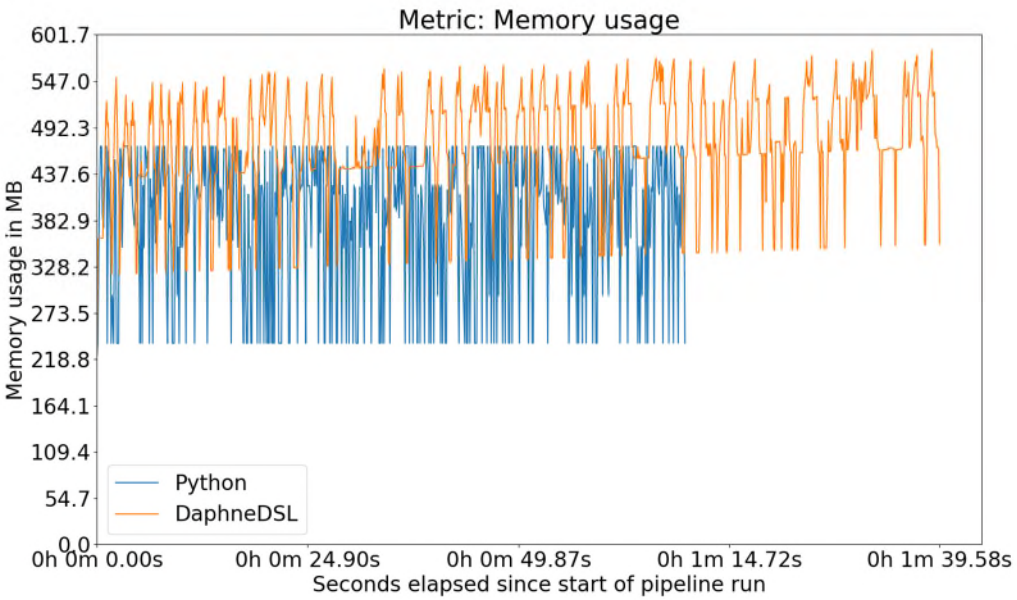


Figure 5.7: Memory allocation – population size 1000, sampling 0.1s

Compared to a population size of 100, the plain Python implementation now utilizes more of the available CPU resources. However, the memory allocation leak remains an issue on the DAPHNE side. The performance advantage of the plain Python implementation diminishes significantly as the population size increases, reducing from being 6.3 times faster to only 1.4 times faster.

5.3.2.3 Population size 10000

The trends observed in other population sizes continue. The plain Python implementation is now 1.2 times faster than the DaphneLib implementation. Overall, these scenarios demonstrate that the more computations are shifted towards DAPHNE, the faster the pipeline becomes.

The following figures present the benchmark:

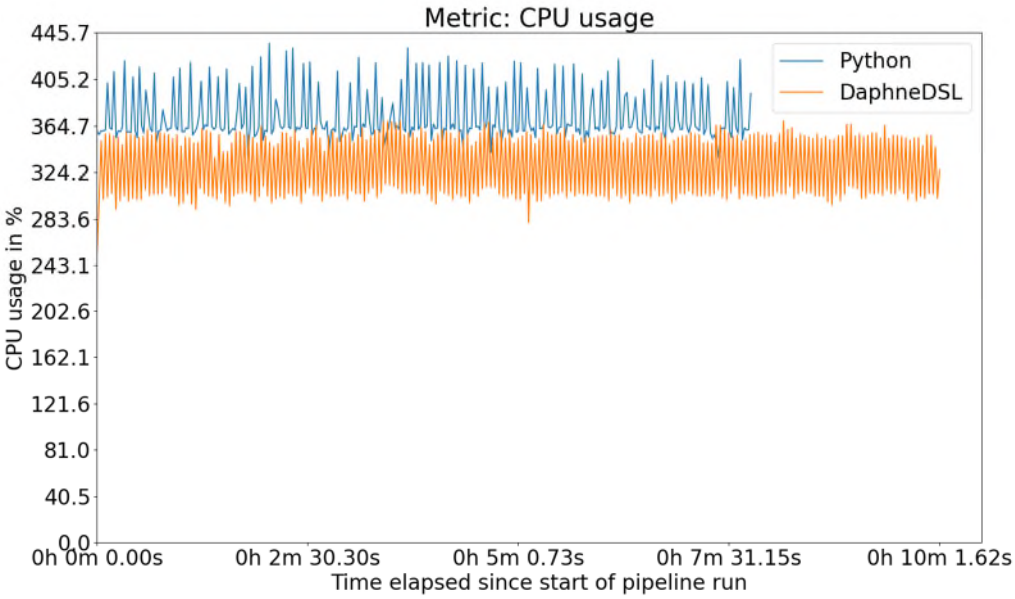


Figure 5.8: CPU utilization – population size 10000, sampling 1.5s

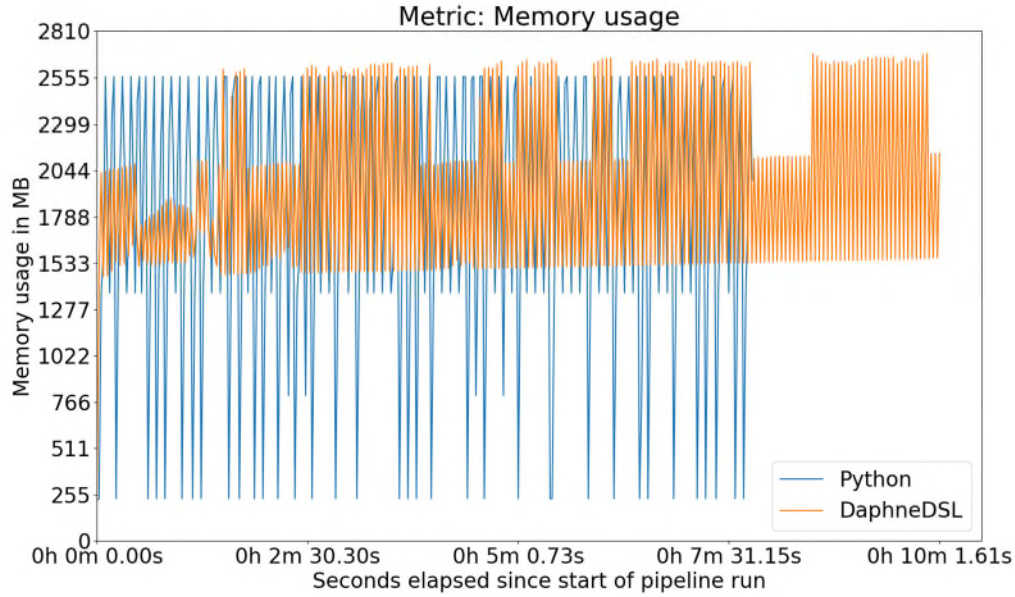


Figure 5.9: Memory allocation – population size 10000, sampling 1.5s

6 Automotive Vehicle Development Case Study: Virtual Prototype Development (AVL 2)

This case study is a step towards the vision of data-driven process optimization: The data (CAD models capturing product geometry, simulation models capturing product behavior etc.) describing the product under development and the evolution thereof along the development process shall be utilized to find patterns which can be improved in future product development. For instance, an example pattern might indicate similar simulation and physical test results for specific test cases. This entails that these physical tests can be avoided in future product development, rendering the underlying development process more efficient in terms of time and cost, because costly and time-intensive physical tests can be replaced by significantly cheaper and quicker simulation without compromising product quality.

Specifically, this case study focuses on creating synthetic data, i.e., data describing the product under development and the evolution thereof along the development process. This synthetic data is the prerequisite for a follow-up step (beyond the DAPHNE project): The pre-training of machine learning models (to be utilized for process optimization), which can eventually be refined with relatively small volumes of real-world data. Synthetic data is inevitable in the context of the given case study, which poses a few-shot learning problem: There are not millions, but only tens of different development projects available to learn from in a typical organization, e.g., an automotive OEM or supplier.

The synthetic data to be created shall be as realistic as possible, i.e., show the same patterns and flaws that are observed by experts in real-world product development. Moreover, the pipeline producing this data shall work autonomously, without requiring manual input, so that datasets can be created automatically and in sufficient volume to be able to train state-of-the-art machine learning models.

Since creating such training data in sufficient volume is a numerically highly demanding task (as are the follow-up steps of data management, data analysis, and machine learning model training), DAPHNE technology needs to be leveraged.

6.1 Description of the Use-Case Pipeline

The pipeline has been substantially reworked and improved within the DAPHNE project: As described in deliverable D8.2 (section 6.2), a severe restriction of the baseline pipeline has been the requirement of manual input during pipeline execution. To achieve pipeline automation, genetic algorithms are utilized as a new approach to synthetic data generation. The conceptual framework of applying genetic algorithms to development process data generation has been elaborated in collaboration with University of Maribor: The essential concepts of genetic algorithms (candidate solution representation, fitness function, manipulation operators) have been defined for the domain of product development processes. This conceptual framework has been implemented leveraging DAPHNE technology. The architecture of this pipeline is shown in Figure 1. The pipeline has been substantially reworked during the DAPHNE project. In the following, we explain the pipeline in more detail.

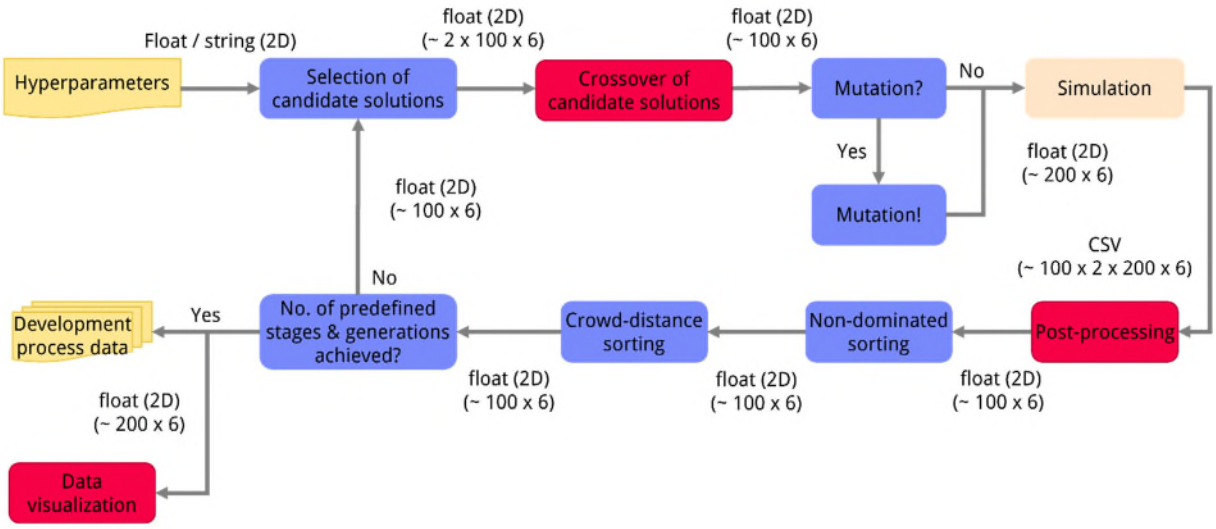


Figure 6.1: Architecture of the pipeline for development process data generation

The first pipeline step is the manual specification of pipeline hyperparameters, i.e., the number of generations for the genetic algorithm, number of individuals per generation, and development process structure. The development process structure captures the constraints of real-world development processes and ensures that the synthetic data also adheres to these constraints, e.g., that a process consists of several subsequent development phases, and different aspects of the product are elaborated in different process phases.

Based on these hyperparameters, a set of (initially random) parent candidate solutions are selected from a pool of candidate solutions. A new generation of child solutions is created from this set of parents by means of crossover, i.e., merging parameter vector subsets of different parent candidates. To enhance diversity among child solutions, additional solution candidates are created randomly using mutation, i.e., by randomly changing individual parameter values. Specifically for methods of genetic algorithms (e.g. cross-over; mutation), the `pymoo` framework[17] is applied. Each candidate solution in this generation is fully characterized by a unique parameter vector (cf. D8.2 section 6.2.2), which encodes a specific product design.

The KPIs (Key Performance Indicators) of each candidate solution are computed by parameterizing a simulation model with the candidate's parameter vector, executing, and post-processing the simulation. Based on the KPI results, the fitness of each candidate solution is evaluated by means of sorting (both non-dominated and crowd-distance sorting, cf. [4] for details) with respect to the KPI results. The best candidate solution in each generation is stored in the development process data file as an individual data point, i.e., a parameter-KPI-tuple.

In the last step of the pipeline, the generated data is visualized so that domain experts can assess and judge data quality (cf. next section for examples of data visualization). If the number of generations as specified by a hyperparameter is reached, the pipeline stops, otherwise the best candidate solutions are used as parents in the next generation of the optimization.

[17] https://pymoo.org

Crucially, this pipeline does not need manual intervention, but the expected patterns in the time series data of parameter values over time and KPI values over time emerge from the optimization process based on genetic algorithms.

6.2 Benchmarking Setup

For benchmarking the pipeline, UMLAUT[18] and the Python library timeit[19] are used. The focus of benchmarking is put specifically on three sections in the pipeline (see Figure 6.1), where DaphneLib is utilized: Crossover, post-processing and data visualization. The crossover function is predefined in the opensource framework pymoo and requires additional tools and libraries. The Docker image[20] provided by DAPHNE is running on a Windows Subsystem for Linux (WSL[21]). For benchmarking, the DAPHNE commit 4e96943[22] is used. Besides the DaphneLib integration, the pipeline also includes a third-party tool for simulation: As mentioned in section 6.1, the evaluation of each proposed candidate solution by means of simulation is essential. For executing these simulations, the AVL co-simulation tool Model.CONNECT™ [23] is used. In Table 6.1, the software tools and libraries used for running and benchmarking the pipeline are listed. In Table 6.2, the high-level specifications of the hardware system used for benchmarking are listed.

Table 6.1: List of tools and libraries used for executing and benchmarking the pipeline

Tool / Library	Version
Cython	3.0.10
Docker	24.0.5
Linux	Ubuntu 22.04.4 LTS
Model.CONNECT™	R2022.2
Numpy	1.24.4
Pandas	2.0.3
pymongo	2.6.1
pymoo	0.6.1.1
UMLAUT	0.1.0
WSL2	2.2.4.0

Table 6.2: Hardware system for running pipeline benchmarks

Attribute	Version
Processor	Intel(R) Xeon(R) Gold 6136 CPU @ 3.00GHz (2 cores)
RAM	176 GB
System type	64-bit operating system, x64-based processor

[18] https://github.com/daphne-eu/umlaut
[19] https://docs.python.org/3/library/timeit
[20] https://hub.docker.com/r/daphneeu/daphne-dev
[21] https://learn.microsoft.com/en-us/windows/wsl
[22] https://github.com/daphne-eu/daphne/commit/4e96943
[23] https://www.avl.com/en-de/simulation-solutions/software-offering/simulation-tools-a-z/model-connect

6.3 Productivity & Performance Improvements

As mentioned in section 6.1, the pipeline existing at the beginning of the DAPHNE project (referred to as "baseline0" in the remainder of this section) has been improved in terms of architecture and algorithmic approach ("baseline1", using genetic algorithms) and in terms of implementation ("baseline2", using DAPHNE technology) in the DAPHNE project. Thus, there are 2 different comparisons to be analyzed:

- Baseline0 vs. baseline1: Productivity and performance improvements due to changes in architecture and algorithms, as addressed in section 6.3.1
- Baseline1 vs. baseline2: Productivity and performance improvements due to leveraging of DAPHNE technology, as addressed in section 6.3.2.

6.3.1 Pipeline Results: Development Process Data

Since the baseline1 pipeline has been substantially reworked compared to the baseline0 pipeline, the development process data generated by the baseline1 pipeline shall be presented and discussed.

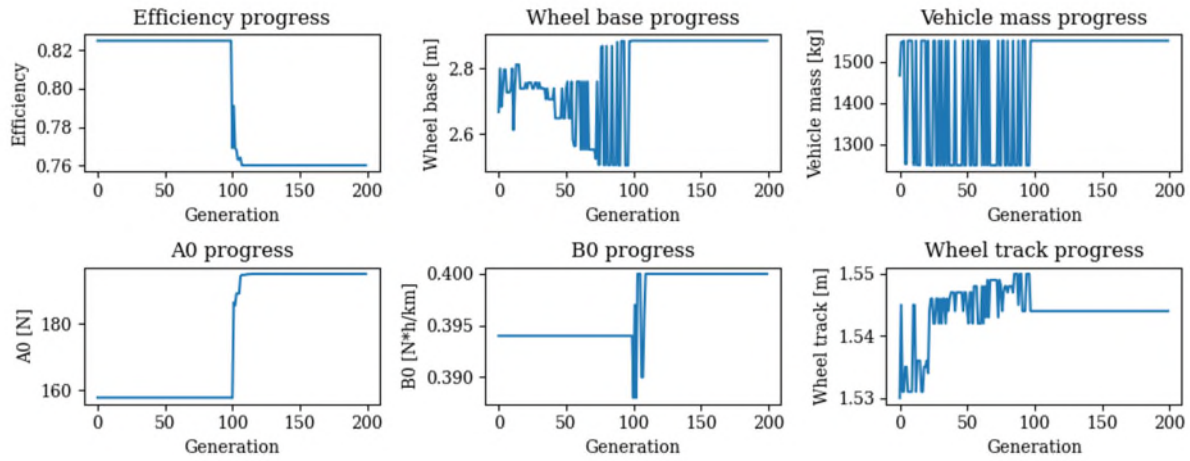


Figure 6.2: Generated development process data: Parameter values over generations. Parameter values of the best candidate solution in each generation are plotted

Figure 6.2 shows the visualization of a sample development process dataset. On the x-axis, the consecutive generations of the genetic algorithm optimization are plotted, which is semantically equivalent to time passing in the development process. On the y-axis of each subplot, one specific parameter (i.e., component of the parameter vector) is plotted. The blue curves consist of individual data points: One data point (i.e., the parameter-KPI-tuple) per generation. Each data point is the best / fittest candidate solution in its generation. In this example (simplified for illustrative purposes), each candidate solution is specified by 6 different parameter values: The product under development in this case is a car, the design of which is encoded by these 6 parameters. Considering the parameter "Wheel track" (bottom right subplot), one can see that the parameter value changes from generation to generation, showing an oscillating behavior and converging to a final optimal value, which is also the behavior of parameter values over development time typically observed in real-world development projects. One can also observe in these 6 subplots that shortly after Generation 100, an optimal parameter vector has been found. Finally, the development process structure as specified in the hyperparameters is reflected in these subplots: For this simplified example development process, the process is

defined to consist of 2 phases: The 1st phase optimizes fundamental hardware-related parameters (wheelbase, wheel track, vehicle mass) while the rest of the parameters are kept constant (at best-guess values). The 2nd phase optimizes the remaining parameters while retaining the optimal values found in the 1st process phase.

Figure 6.3 shows the visualization of the same sample development process dataset as Figure 6.2, but now product KPIs over generations (i.e., the KPI part of the parameter-KPI-tuple). In this simplified example, two different KPIs of the product under development are considered. The parameter values are the input of the simulation (cf. Figure 6.1), while the KPIs are the output of the simulation. The KPIs are relevant for evaluating the fitness of each candidate solution: The closer a candidate's KPIs are to the pre-defined target values as specified in the product's requirements (not shown here for sake of simplicity), the better the candidate solution is. As for the parameters, also the generated KPI time series show the oscillating and converging behavior, as observed in real-world projects.

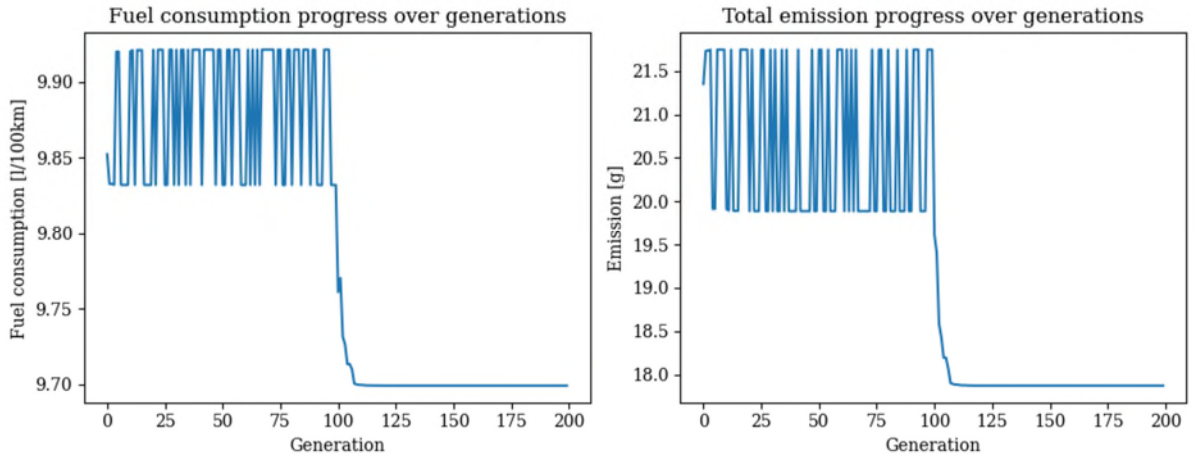


Figure 6.3: Generated development process data: KPI values over generations. Specifically, KPI values of the best candidate solution in each generation are plotted.

In summary, the development process data created by the baseline1 pipeline well addresses the two measurables defined in deliverable D8.1 (section 6.3): The first measurable is the similarity of the synthetic data to those data and patterns observed in real-world projects. As analyzed in Figure 6.2 and Figure 6.3, the created data exhibits patterns also typically observed in real-world projects, while not showing any patterns deemed unrealistic by domain experts. The second measurable is the pipeline's scalability. As laid out in this chapter, the revised pipeline creates realistic synthetic training data autonomously, without manual intervention during pipeline execution. Thus, this autonomy provides for scalability of data generation, since the main bottleneck of manual intervention (as laid out in deliverable D8.2, section 6.2) is resolved, which means that synthetic data can be created in significantly larger volume using the revised pipeline.

6.3.2 Pipeline Benchmark: Results

In this section, the effect on performance of DaphneLib utilization in the baseline2 pipeline investigated. Comparing Figure 6.4 and Figure 6.5, one can observe that the performance of the pipeline is decreasing overall. We attribute the longer runtime to intense usage of parallel-

ized matrix computations in the baseline1 pipeline: In the data visualization pipeline step (Figure 6.1), matrices for each of the six parameters are computed in parallel. Since DaphneLib is (at the time of benchmarking) not capable of parallelized matrix computations, the observation of increased runtime is to be expected. In the CPU usage the baseline2 pipeline is peaking twice and the memory behavior is similar to baseline1. Since the data visualization step is not executed repeatedly but only once per pipeline execution, any potential memory leak cannot be observed in the results.

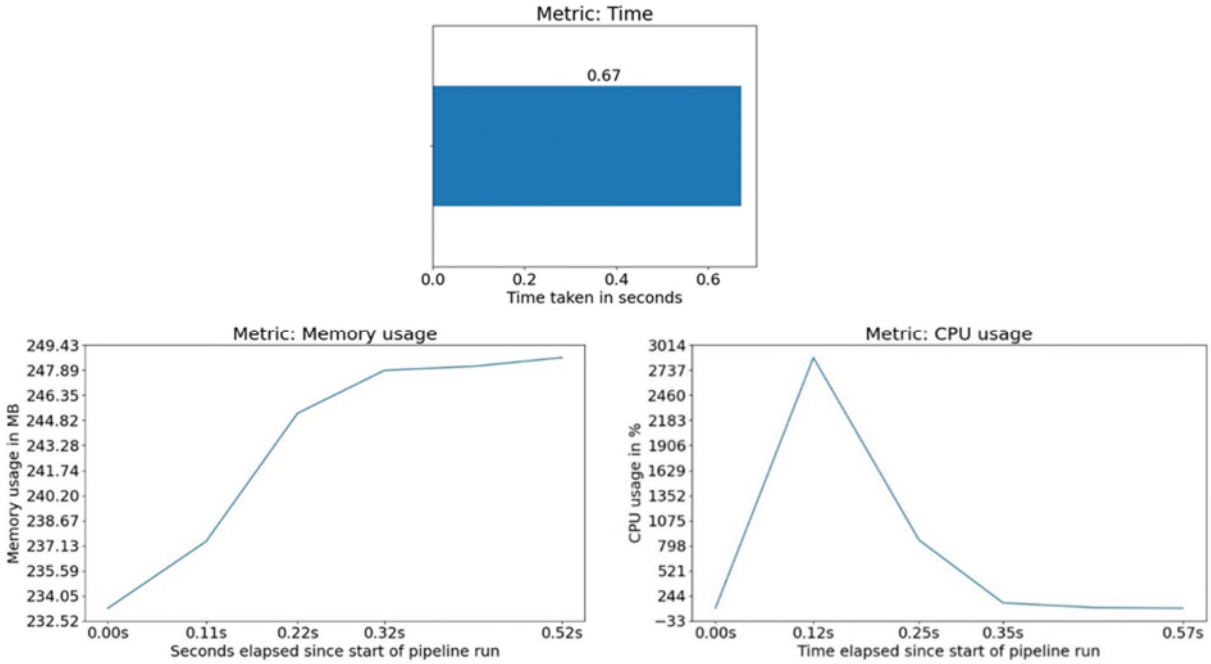


Figure 6.4: Benchmarking the data visualization pipeline step without DaphneLib (baseline1 pipeline)

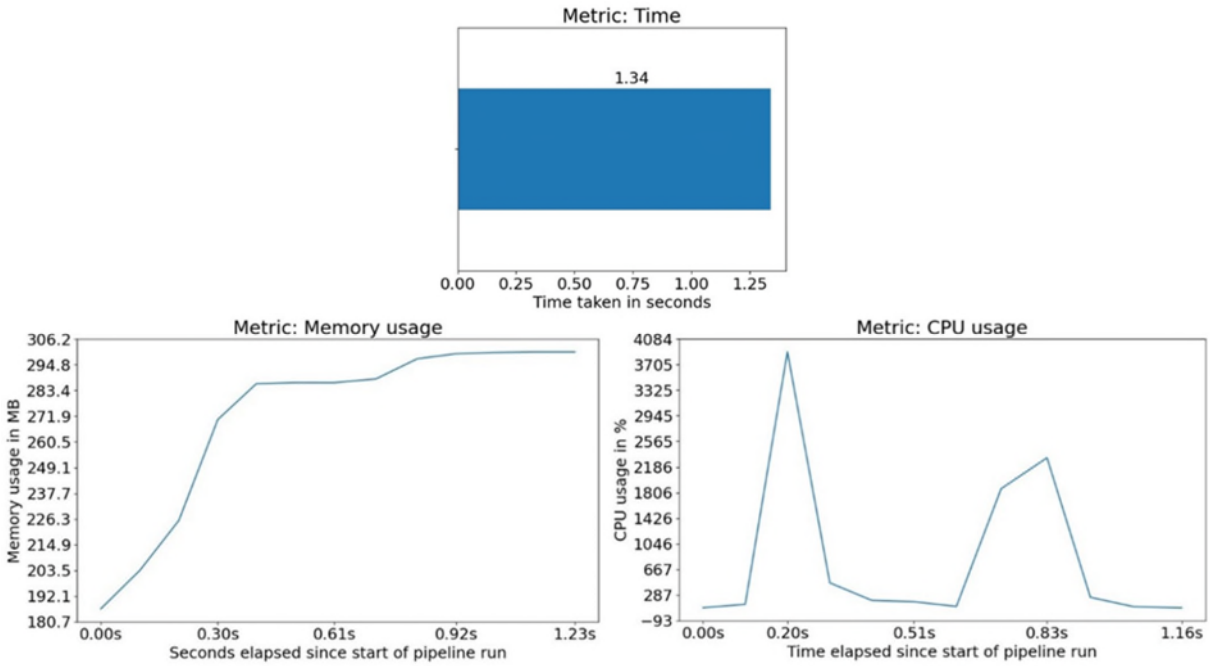


Figure 6.5: Benchmarking the data visualization pipeline step with DaphneLib (baseline2 pipeline)

For the postprocessing pipeline step, we observe a similar memory usage behavior for both baselines, as shown Figure 6.6 und Figure 6.7. Differences can be seen in the CPU usage which plateaus at approximately 100% for baseline1 and has a sharp peak for baseline2. As for the

visualization pipeline step, the worse pipeline performance is caused by parallel computations: The postprocessing computations are executed in parallel for each candidate solution in a given generation (for benchmarking, 100 candidate solutions are evaluated per generation). With parallelized computing capabilities in DaphneLib these timing issues can not only be prevented, but it is likely that the performance will improve.

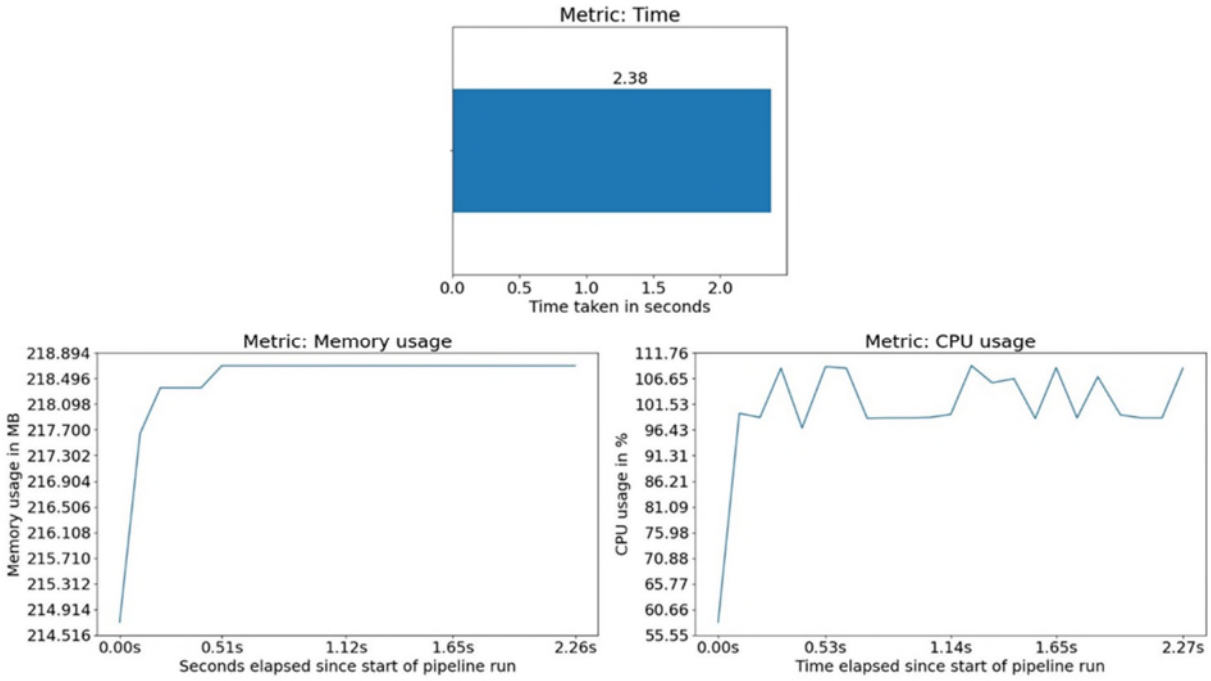


Figure 6.6: Benchmarking the postprocessing pipeline step without DaphneLib (baseline1 pipeline)

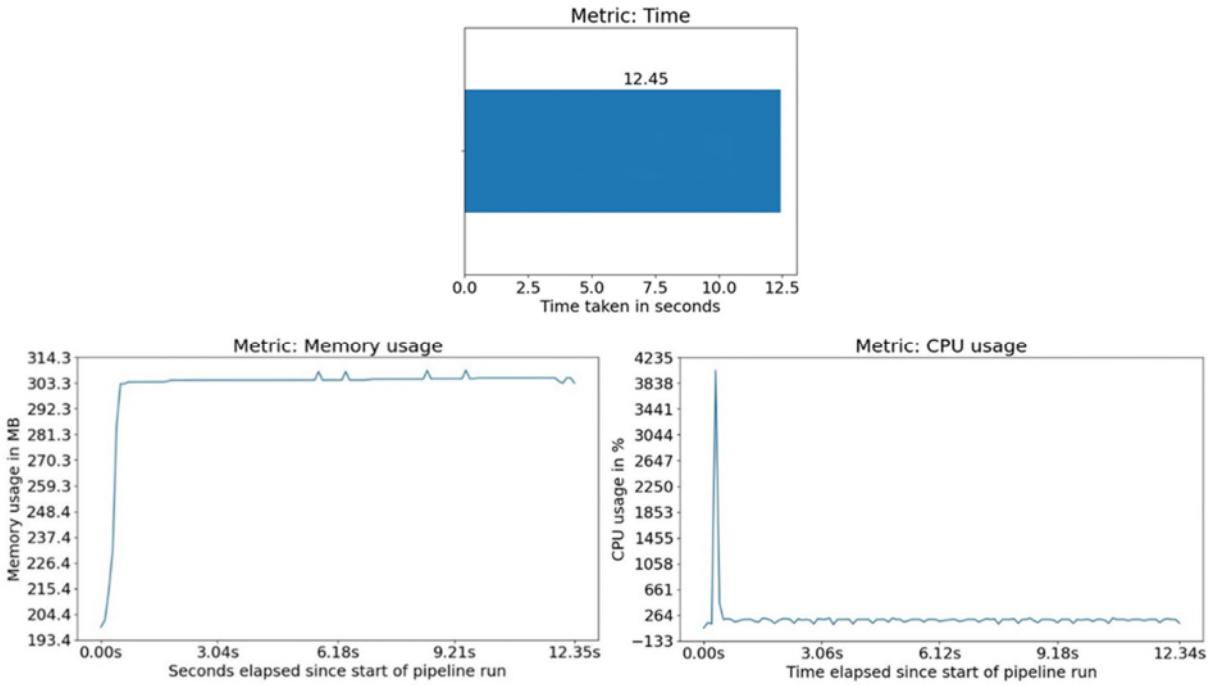


Figure 6.7: Benchmarking the postprocessing pipeline step with DaphneLib (baseline2 pipeline)

Overall, the pipeline performance (baseline2 vs. baseline0) has improved significantly within the DAPHNE project: A full simulation run for baseline0 took approximately four days. With the genetic algorithm approach of the revised pipeline, the pipeline execution is now fully automatically and significantly faster. One execution of the revised pipeline (both baseline1 and

baseline2) can be done (using the same hardware setup) within one day. These improvements can substantially overcompensate the DaphneLib performance drawbacks, which are exclusively caused by the issue parallelized computation. Prospective parallelized computation capabilities and genetic algorithm methods provided by DAPHNE would likely further improve the pipeline performance substantially.

7 Additional Integrated Pipelines

In this section, we present two additional pipelines using the DAPHNE system. These pipelines have been developed by our partners during the DAPHNE project.

7.1 Randomized Optimization Algorithms

This additional pipeline presents Randomized Optimization Algorithms (ROA) in DAPHNE run in Singularity container on EuroHPC Vega using Slurm [5].

First, the DAPHNE system is downloaded as source code, cloning DAPHNE main repository as from GitHub: the git command is invoked, then the remote code is cloned into the local file system. Then, a singularity image is compiled locally and transferred to the Vega so that it can be used later for compilation of DAPHNE: the singularity command is invoked, then the build proceeds and completes by creating the daphneeu.sif image.

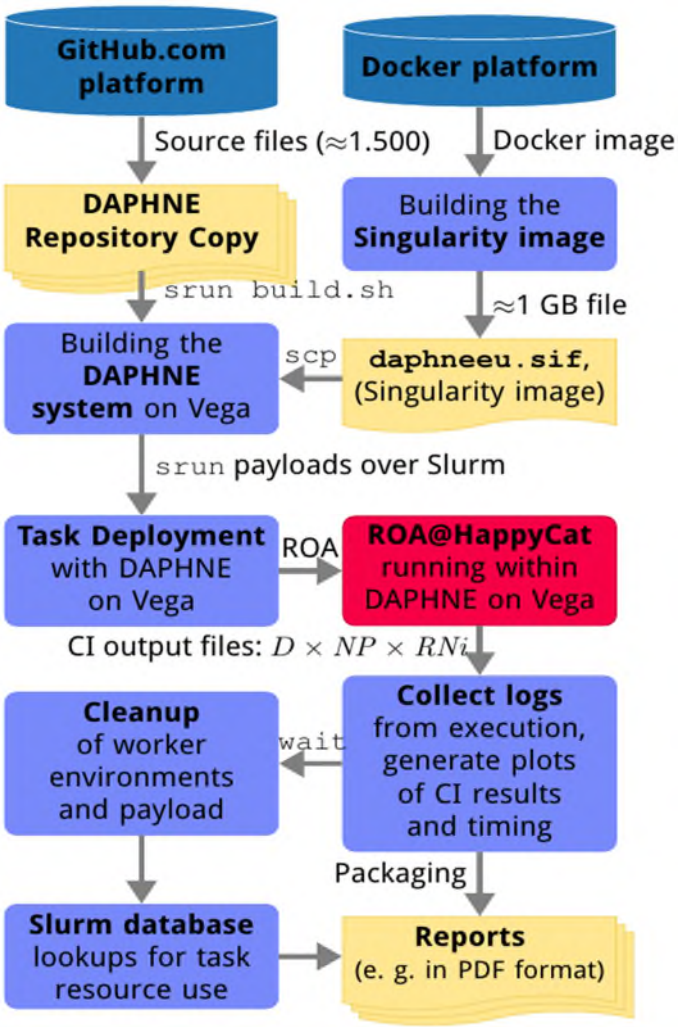


Figure 7.1: Pipeline definition flowchart for DAPHNE ROA CI additional pipeline

In order to validate DAPHNE in an EuroHPC infrastructure, the container image is copied to Vega, where the two-factor authentication and checking of the user access certificate take place: the image file is specified, then authentication takes place, then copying proceeds. With the singularity image and source code in-place, the DAPHNE system is then compiled on the target system (Vega) from source code: the compilation is started, and the build then finishes. After the images are prepared, a ROA [6] is implemented in roa.d and run with different configurations (sample configurations with for...do and the srun command) and saving of outputs and timings. Each task contains an independent run that start with a fixed seed (RNi) for random generator in ROA. For each task, up to 1 GB memory and 10 minutes node use are requested to run the workload using the container.

To further explain the deployment and benchmarking of ROA in DAPHNE for our use case, we also provide a flowchart which is seen in above Figure 7.1 and shows how the data flows in the ROA use case. The ROA@HappyCat center part in red color is completely executed by the DAPHNE system, as the Differential Evolution (DE) [7] parameters D (dimension), G (number of

generations), and NP (population size), and function for fitness evaluation (HappyCat) are provided within the core of the use case. The pipeline generates data analysis reports (e.g. in PDF format, in the bottom of the flowchart), after it builds the Singularity image from Docker platform and DAPHNE from GitHub source and runs the ROA tasks over Slurm (flowchart top). The collection of logs and cleanup after waiting of the tasks completion and lookup into the Slurm database to see resource use, then contribute to generating the reports.

7.1.1 Benchmarking Setup

A sample setup run set for ROA in single objective case is presented in this paper, using a challenging optimization function HappyCat [8], where the fitness evaluation of a numerical input vector **x** is computed by the function

$$f = ((\sum(x^2) - 10)^2)^{0.125} + \frac{\frac{\sum(x^2)}{2} + \sum(x)}{10} + 0.5$$

The CPU partition of Vega was used and prepared as described in [5], using Singularity and source code of DAPHNE with underlying software packages. As DAPHNE supports LLVM lowering and also implements Multi-Level Intermediate Representation (MLIR) kernels [6], the generated code for the fitness function in LLVM was presented in [5], for which the function definition includes the constants definition, allocations, and the specific calls to the implemented kernels (e.g. for matrix operations like addition/subtraction or multiplication/division). The algorithm is configured with combinations of D at 10, 100, 1000 and NP at 10, 100, 1000.

As D and NP are increased by tenfolding (each increase is ten times larger), their impact to result quality and workload time is observed, while testing if the system computes the workload successfully. The setup script is run on Vega and is listed in Figure 7.2: the run-daphne.sh is supplied with the roa.d file that is parameterized with D, NP, and RNi. The run-daphne.sh executes within the daphneeu.sif image, which itself is executed over Slurm.

```
for D in 10 100 1000; do
      for NP in 10 100 1000; do
            echo D=$D NP=$NP
            for RNi in {1..10}; do
                  echo -n .
                  { time srun --mpi=none \
                        --time 10 --mem=1G \
                        ../daphneeu.sif \
                        ./run-daphne.sh roa.d \
                        D=$D NP=$NP RNi=$RNi \
                        > results-D-$D-NP-$NP-RNi-$RNi-out.txt &
                  } 2> time-D-$D-NP-$NP-RNi-$RNi-out.txt
            done #RNi
      done #NP
done #D
```

Figure 7.2: Deployment of ROAs with DAPHNE using Slurm on Vega

Although the prepared independent randomization runs output the same computational results for the same configuration, their runtime varies. Considering that in the previous deliverable D8.1 the runtime was the identified crucial measurable, this benchmarking setup focuses

on measuring runtime when allocating Slurm tasks on HPC and executing the randomization runs as well. The allocation time is important for HPC users so that they do not wait for their results longer than running sequentially, as allocation times might exceed the combined time rendering the speed up nonsignificant from the user perspective, which should be alleviated and is hence benchmarked here as well within the measured time.

7.1.2 Productivity & Performance Improvements

As identified in the previous deliverable D8.1 that measuring time is the crucial measurable and as in deliverable D8.2 it has been identified, that e.g. automotive Product Development Processes includes finding near-optimal solutions by iterating and optimizing product design, productivity and performance are addressed in this subsection with regard to improving a framework that creates an optimization technique sub-pipeline being mapped to some specific case study. Therefore, to check that the optimization technique works, the generated ROA computational results are gathered and checked if the fitness is converging toward the minimum values and hence the main functionality works.

The optimization results from the ROA runs (fitness convergence through generations) as explained in the above deployment preparation, are presented in Figure 7.3 as a set of convergence graphs, in configurations with dimensions D of 10, 100, 1000 and population sizes NP of 10, 100, 1000.

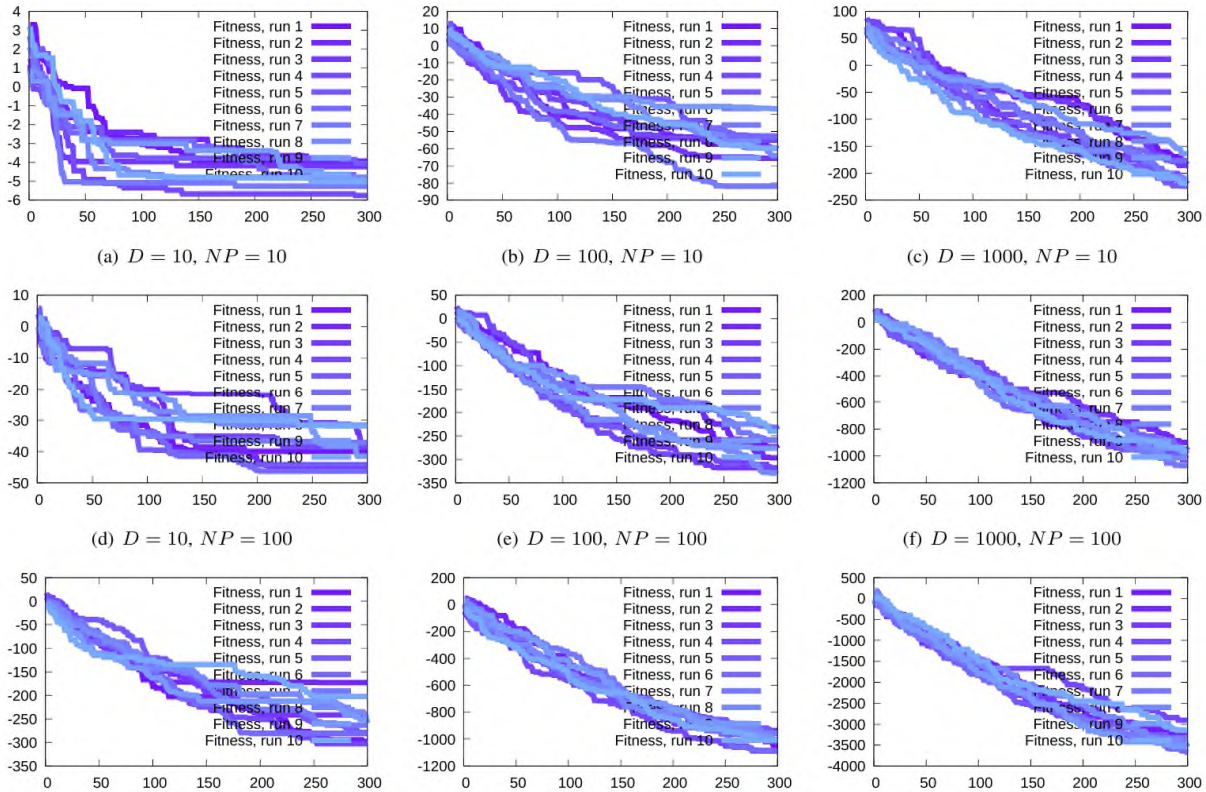
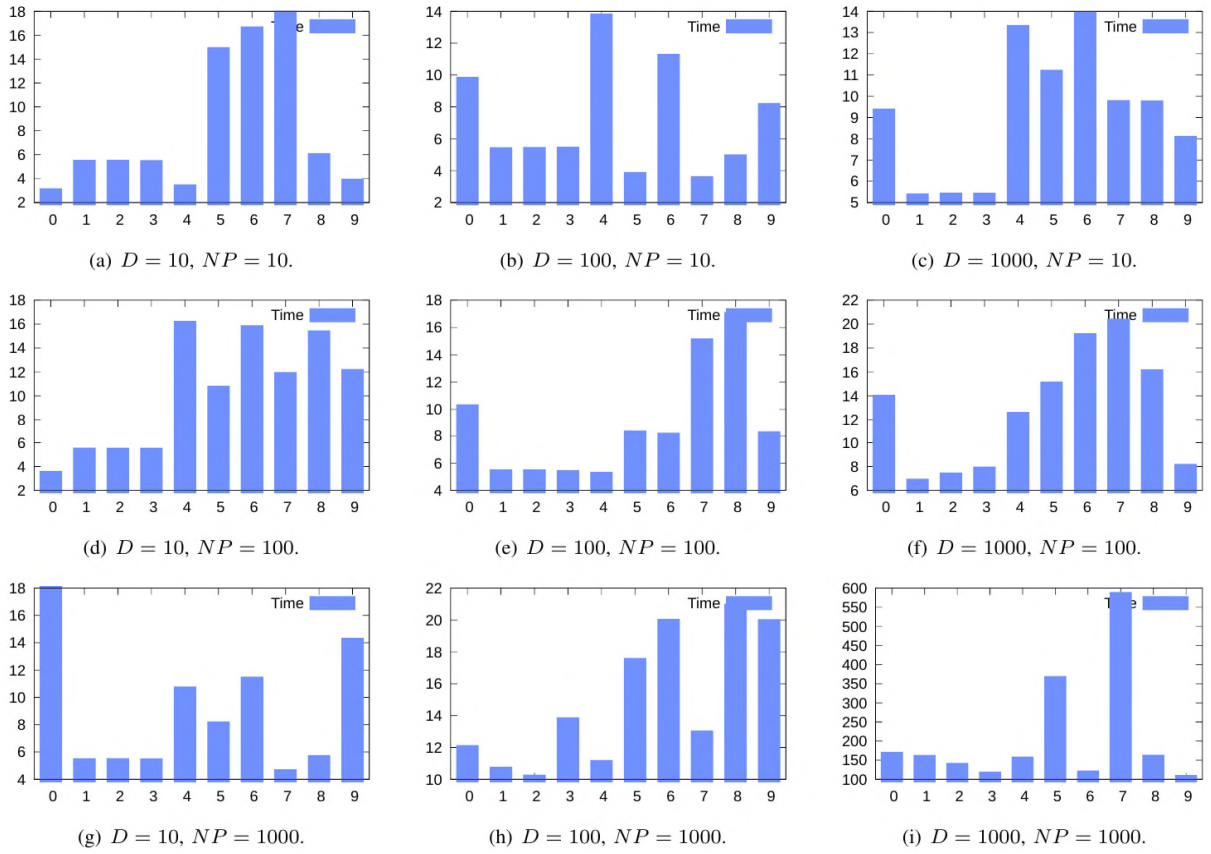


Figure 7.3: Convergent optimization runs (fitness on vertical axis, generations on horizontal axis) using DAPHNE ROA on function HappyCat, for different independent seeds

For each of the runs plotted, we observe that the fitness function optimized by the ROA is successfully improving, hence, the ROA CI is performing its main functionality of optimization.

The respective timings of the real time to allocate and execute different job variations are shown in Figure 7.4 as reported by time command. We can observe that the configuration of D=1000 and NP=1000 in case (i) has the far highest time requirements overall for these cases. When observing each of the subfigures separately, we see some limited degree of variation in job allocation and execution waiting time from 2 to 22 seconds, but these are much less than the case (i) that always reported timings above 100 seconds (with only run 5 and 7 above 200 seconds, but still below maximum requested allocation of 10 minutes). We also further inspected the Slurm database to profile run 7 and see that while it consumed 92.139 Wh in 582 s, the task has spent only 8 seconds waiting to be allocated, on empty current user queue, which further demonstrates fast Vega task allocations, practical in this use case.



(a) $D = 10$, $NP = 10$. (b) $D = 100$, $NP = 10$. (c) $D = 1000$, $NP = 10$.

(d) $D = 10$, $NP = 100$. (e) $D = 100$, $NP = 100$. (f) $D = 1000$, $NP = 100$.

(g) $D = 10$, $NP = 1000$. (h) $D = 100$, $NP = 1000$. (i) $D = 1000$, $NP = 1000$.

Figure 7.4: Times (in seconds, on vertical axis) of running optimization runs (runs 0 to 9, on horizontal axis), for different configurations (a)-(i)

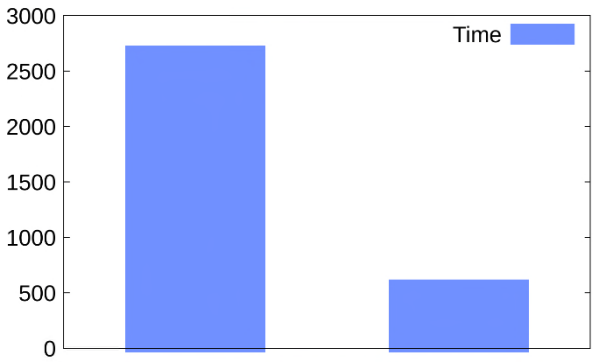


Figure 7.5: Combined time (left bar in the plot) vs. batched time (right bar in the plot)

In Figure 7.5 on the right, the combined time of processing all batched jobs, compared to running them with Slurm, is presented, demonstrating the speedup of real time needed by

running the tasks in parallel, and hence, scaling. Also, when running just a subset of the jobs with much more similar timings (e.g. jobs with D=100, NP=100) for much more independent runs, the speed up is mostly capped by the longest running job.

While the responsiveness of the Slurm scheduler varies slightly due to HPC workload of all running jobs, the batching of the set of jobs however greatly reduces the crucial measurable of time required to execute a batch, compared to just sequentially running each job. Also, as allocation time is important for HPC users considering their productivity so that they do not wait for their results longer than running sequentially, the allocation times by far did not exceed the combined time, i.e. the speed up was significant also from the user perspective. The main observed advantage is hence the potential for scaling the ROA that was successfully benchmarked and scaled through tasks in Slurm.

Additionally, to the main observed advantage, Very Large Scale Global Optimization (VLS-GO) in context of Randomized Optimization Algorithms (ROA) in DAPHNE is run for dimension sizes beyond ten thousand and up to hundred thousand. The latest software from DAPHNE repository daphne at the GitHub account daphne-eu, using the last July 2024 commit to the main branch (548ea01), is compiled and deployed on Euro-HPC Vega supercomputer in Maribor, Slovenia. The compilation is then deployed using Slurm to execute a set of ROA runs with different configuration classes for VLSGO, increasing the optimized parameter sizes [9]. Then, the newly obtained computational results from ROA runs are reported and analyzed below.

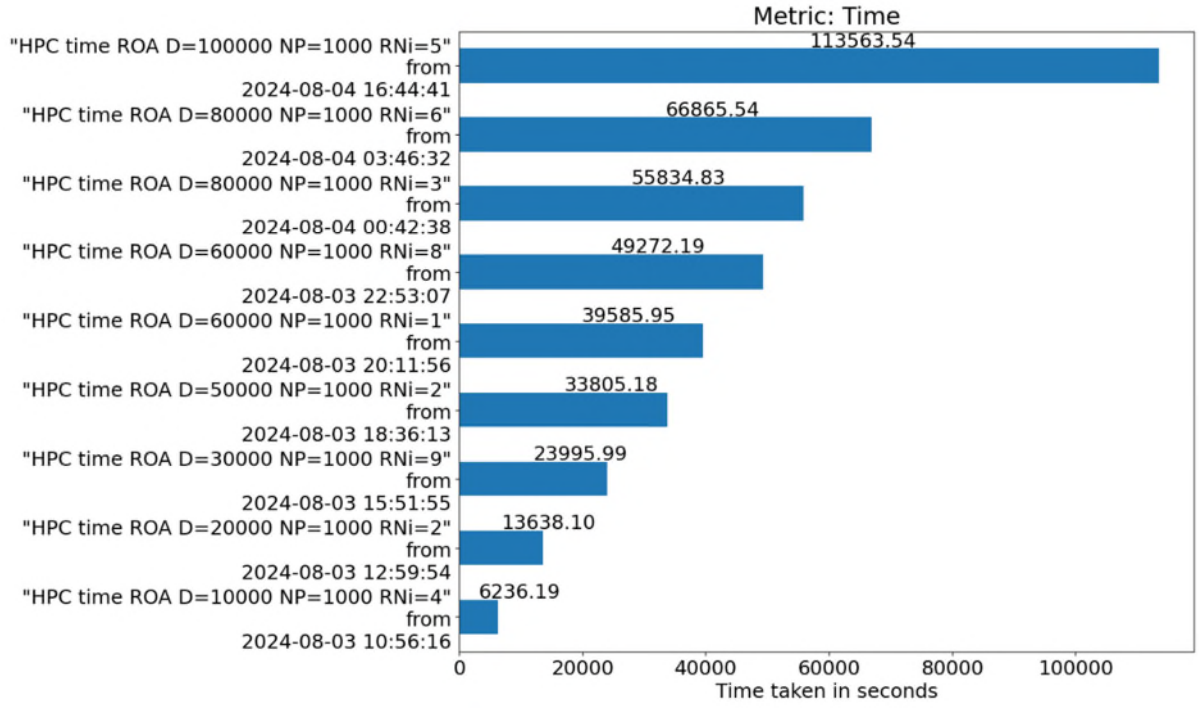


Figure 7.6: ROA timing using Umlaut

Further, using Universal Machine Learning Analysis Utility (Umlaut) from GitHub[24], the resource usage is tracked during execution. As an example, for configuration D=10000, NP=1000, the total runtime over some of these independent runs is seen in Figure 7.6. The aggregated runs

[24] https://github.com/daphne-eu/umlaut

displayed in Figure 7.7 demonstrate the linear increase of runtime with dimension, suitable for scaling. Furthermore, Figure 7.8 provides usage of memory and CPU resources, respectively. The memory usage plot peeks at approximately 105.87 MB for these runs, by initially rising to roughly 89 MB and then slightly increasing. It very likely also shows the remaining memory leaks indicated by the other use-case pipelines.

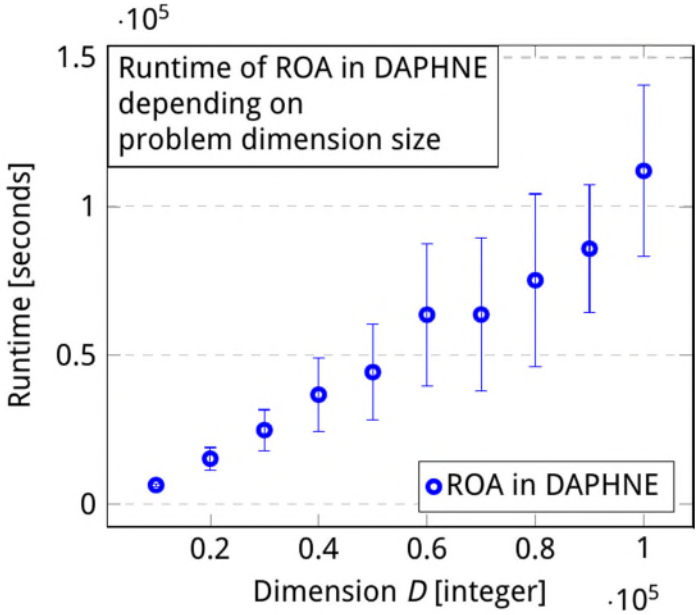


Figure 7.7: Average and standard deviation aggregated plot for runtimes

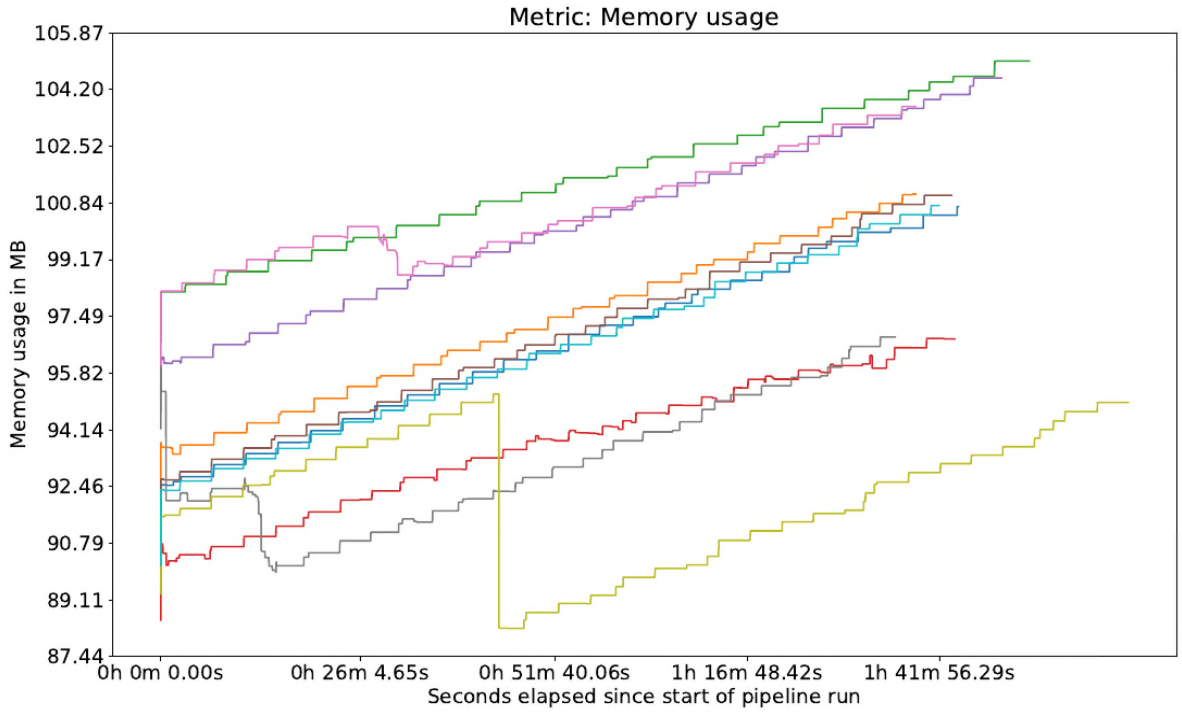


Figure 7.8: Memory usage for ROA tracked using Umlaut

7.2 Surface High-Density Electromyogram (HDEMG) Processing

High-density surface electromyography (HDEMG) is one of the most popular methods for non-invasive acquisition of muscle activity. Two decades of significant improvements in acquisition systems brought along also the development of decomposition algorithms that decompose

the EMG signals into contributions of individual muscle motor units, enabling identification of motor unit firing times. However, such data processing pipelines present a significant computing challenge and require substantial computing power. This makes it a good candidate for high performance parallel data processing of HDEMG signals on modern CPUs, GPUs, or HPC clusters.

High-density surface EMG signals are recorded with special flexible electrodes consisting of multiple channels, typically, 32, 64 or more channels are used. Each channel is stored as one row in a large data matrix, so 64 channels result in 64 rows. The number of columns depends on the recording length and sampling frequency: e.g., a 4096 Hz sampling frequency which results in 245,760 samples (columns) per one minute of recording, stored in double format. This results in fairly large matrices, which can easily reach 10 GB to 30 GB in size.

A simple data processing pipeline was prepared and implemented in DaphneDSL that demonstrates realistic matrix operations that are often used in EMG processing, such as concatenation, extension, and calculation of correlation matrix [10]. The benchmarking setup and results are provided below.

7.2.1 Benchmarking Setup

To demonstrate a typical sEMG processing operation we prepared the following pipeline [10]:

- Input matrices A and B are loaded from CSV files.
- Matrices A and B are horizontally concatenated.
- The resulting matrix is extended by a factor R: R-1 time-delayed (right-shifted) copies of each row are added to the matrix.
- Calculation of correlation matrix.
- Calculation of pseudoinverse of the correlation matrix.

This pipeline was implemented as a script in DaphneDSL language using the latest libraries from the daphne-eu/ daphne/main GitHub repository daphne-rep, commit with hash 4e96943 from 2024-07-18. The DAPHNE binary was executed with additional parameters to print the MLIR-based intermediate representation (DaphneIR): `daphne --explain parsing_simplified,property_inference ckc_simple_pipeline2.daph`. The code includes a main function and two sub-functions: for matrix extension and for matrix pseudoinverse.

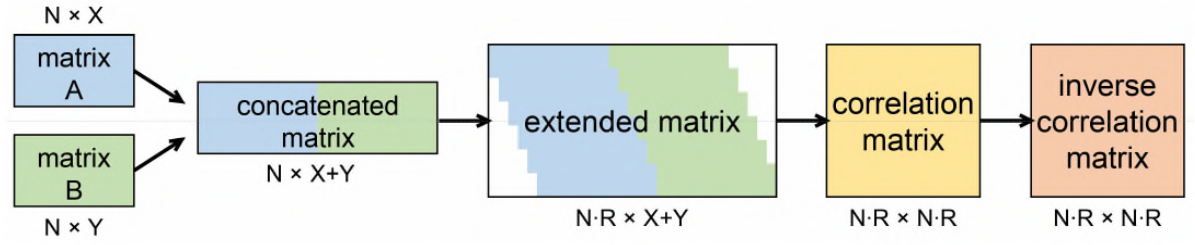


Figure 7.9: Overview of HDEMG pipeline. Boxes represent matrices. N = number of rows, X,Y = number of columns, R = matrix extension factor

For realistic input HDEMG data several sets were used for simulated HDEMG signals for generated signals for the Biceps Brachii muscle, with the following parameters [10]: up to 500 motor units, electrode grid with 10 rows and 9 columns (90 channels), 5 mm interelectrode distance, constant force of 10 %, 30 %, 50 % and 70 % of maximum voluntary contraction, 4096 Hz

sampling rate, 600 seconds in length, no added noise, monopolar recording mode. Data is stored as a 2D matrix of raw EMG values in CSV format. For the initial tests, presented in this paper, only 10 different subsets of the available data were used, with 90 rows and 10,000, 20,000, … and up to 100,000 columns.

A dedicated DAPHNE runtime environment was prepared in a virtual machine created with the QEMU/KVM hypervisor software ver. 4.0.0 on a Linux host computer with Kubuntu 22.04.4 LTS operating system, 64 GB RAM, AMD Ryzen Threadripper 1920X 12-core CPU and Samsung 980 PRO 2 TB NVME SSD. The virtual machine was with Kubuntu 24.04 operating system installed, 16 GB RAM, 4 virtual CPUs and 100 GB of available disk. All the required libraries and dependencies were installed and the DaphneDSL executables built from the GitHub repository source code.

7.2.2 Productivity & Performance Improvements

To measure performance, the pipeline was run 5 times for each of the 10 selected matrix sizes (90 x 10,000, 90 x 20,000, 90 x 30,000, …, 90 x 100,000) and run times measured, as seen in Figure 7.10 and these results show that the run time increases linearly with the size of the input matrices, which is the correct and expected behavior.

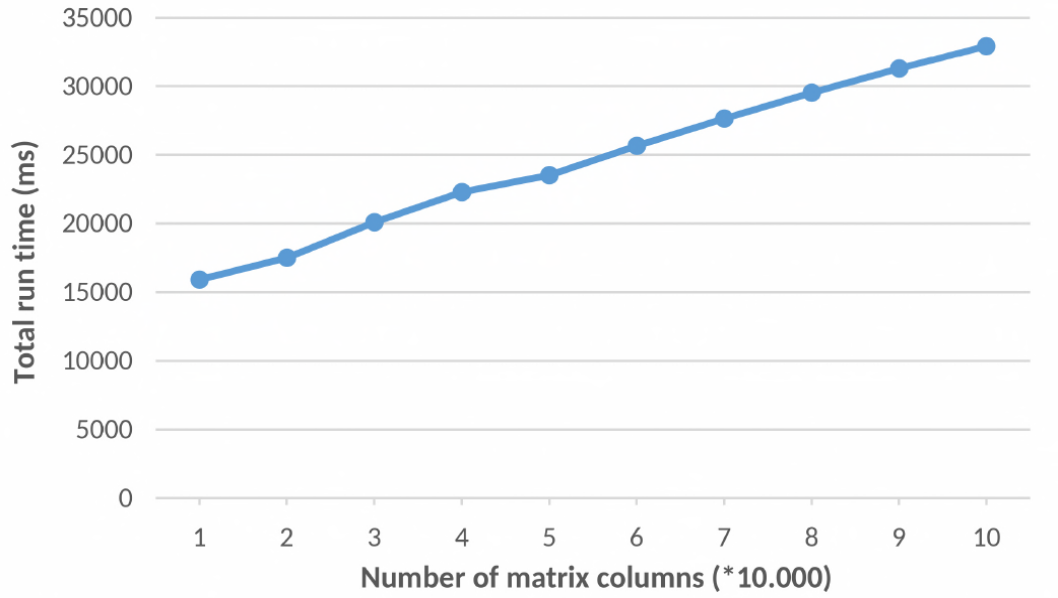


Figure 7.10: Mean total run time of HDEMG pipeline, estimated over 5 consecutive runs. The input matrices consisted of 90 rows and from 10,000 to 100,000 columns. Time is measured in milliseconds

HDEMG signal processing pipeline productivity improvements are hence a viable option and can be used for creating data processing pipelines. As the DaphneDSL language is still in development larger amounts of data and more complex pipelines are also expected in future, further development and testing is also needed to provide an even more reliable and convenient solution for parallel and HPC-ready processing of HDEMG data.

8 References

[1] A. Laber , Ion Implantation Sensor and Process Target Data for Predicting Ion Beam Tuning in Semiconductor Manufacturing, Zenodo, 2024.

[2] M. Visvalingam and J. D. Whyatt, "Line generalisation by repeated elimination of the smallest area," 1992.

[3] D. H. Douglas and T. K. Peucker, "Algorithms for the Reduction of the Number of Points Required to Represent a Digitised Line or its Caricature," *The Canadian Cartographer,* vol. 10, pp. 112-122, 12 1973.

[4] K. Deb, "Multi-objective Optimisation Using Evolutionary Algorithms: An Introduction," in *Multi-objective Evolutionary Optimisation for Product Design and Manufacturing*, London, Springer, 2011, pp. 3-34.

[5] A. Zamuda and M. Dokter, "Deploying DAPHNE Computational Intelligence on EuroHPC Vega for Benchmarking Randomised Optimisation Algorithms," in *International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom)*, Graz, 2024.

[6] S. Silva and L. Paquete, "GECCO '23: Proceedings of the Genetic and Evolutionary Computation Conference," Lisbon, 2023.

[7] R. Storn and K. Price, "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization,* vol. 11, p. 341–359, 1997.

[8] H.-G. Beyer and S. Finck, "HappyCat – A Simple Function Class Where Well-Known Direct Search Algorithms Do Fail," *Lecture Notes in Computer Science,* vol. 7491, pp. 367-376, 2012.

[9] A. Zamuda, "Very Large Scale Global Optimization with Randomised Optimisation Algorithms in DAPHNE," in *33rd International Electrotechnical and Computer Science Conference*, Portorož, Slovenia, 2024.

[10] M. Divjak and A. Zamuda, "Experimental pipeline definition for surface high-density electromyogram (HDEMG) processing," in *33rd International Electrotechnical and Computer Science Conference*, Portorož, Slovenia, 2024.

[11] A. Laber, M. Gebser, K. Schekotihin and Y. Yang, "Predicting Ion Beam Tuning Success in Semiconductor Manufacturing," in *2022 14th International Conference on Advanced Semiconductor Devices and Microsystems (ASDAM)*, Smolenice, Slovakia, 2022.

[12] M. Harman, "Search-based software engineering," *Information and Sofware Technology,* vol. 43, no. 14, 2001.