Supercomputing Frontiers and Innovations

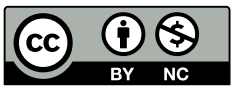
2021, Vol. 8, No. 1

Scope

- Future generation supercomputer architectures
- Exascale computing
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Novel approaches to computing targeted to solve intractable problems
- Convergence of high performance computing, machine learning and big data technologies
- Distributed operating systems and virtualization for highly scalable computing
- Management, administration, and monitoring of supercomputer systems
- Mass storage systems, protocols, and allocation
- Power consumption minimization for supercomputing systems
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Scientific visualization in supercomputing environments
- Education in high performance computing and computational science

Editorial Board

Contents



DOI: 10.14529/jsfi210101

When Sally Met Harry or When AI Met HPC

Ulises Cortés[1,3], *Ulises Moya*[2], *Mateo Valero*[3,1]

Introduction

The Artificial Intelligence (AI) explosion which we are witnessing today can also be, at least in part, credited to the current advances in computing power, in particular to High-Performance Computing. This is not a brand new relation as it can be traced from the very beginning of the hardware and AI developments. Maybe the first encounter between AI and hardware dates back to 1958. The perceptron – a more general computational model than McCullochPitts units – was intended to be a programable machine rather than a software program. While its first implementation was in software for the IBM 704, perceptron subfsequently implemented it in custom-built hardware as the *Mark 1 perceptron* [1, 6]. The perceptron was designed for image recognition: it was an array of 400 photocells, randomly connected to units called *neurons*. Weights were encoded in potentiometers, and electric engines performed weight updates during the learning phase. A seminal interaction between AI and the hardware design was the use of a perceptron for efficient branch prediction to boost instruction-level parallelism [4]. After the years, the evolution of those *neurons* brought the inception of the so-called Neural Networks (NN) as software that gave birth to Deep Learning (DL). In turn, to accelerate DL, nowadays the use of GPU's and more specialized hardware architectures has become the norm (*e.g.* Cerebras CS-2, SambaNova, INTEL's Habana, *etc.*).

In the 80's Thomas Knight, an AI researcher at the Massachusetts Institute of Technology (MIT) said: "*The bigger we make our programs, the 'smarter' they get and the slower they get ... We're in the embarrassing position that we give a program more information, and it gets worse* [9]." The explosion of cheap sensors, the wide use of the Internet and the smart telephones augmented the amount of available data, AI-based technologies request more and more computational power.

The convergence between AI & HPC is seriously and consistently pursued throughout the HPC ecosystem. This includes, as said before, Deep Learning (DL) as the main engine, DL is a greedy approach. It is clear that this AL and HPC union is an excellent opportunity to obtain better and faster scientific results and translate them into industrial applications.

These accomplishments all share a single common thread. Namely, the algorithms developed to accelerate Deep Learning models' training on HPC platforms have a strong experimental component (see [8]). To the date, there is no accurate framework to narrow down the ideal set of hyper-parameters to guarantee rapid convergence and optimal performance levels of AI models as the number of processor or GPU nodes increases to speed up the training stage. Furthermore, it is common, in this community, to compare distributed training algorithms on HPC platforms using idealised neural network models and data sets, *e.g.*, training a ResNet model [10] using the ImageNet dataset [2].

On the other hand, many researchers use AI-based technologies, mostly DL and Reinforcement Learning (RL), to transform how computer systems and chips are designed and opti-

[1]Universitat Politècnica de Catalunya, Catalunya, Spain
[2]Gobierno del Estado de Jalisco, Jalisco, México
[3]Barcelona Supercomputing Center, Catalunya, Spain

mised. Many core problems in systems and hardware design are combinatorial optimisation tasks [3, 5, 7]. AI development has been tightly interlinked with progress in chip design. This cooperation may speed up the achievements in both fields and, it is evident, in terms of scientific development, in general. Today, AI methods are available to solve various complex problems in the design and development of the HPC systems, such as predicting running times, resource utilisation, optimisation of load balancing, job scheduling, resource discovery, and process migration. Recent accomplishments of this program brought the idea of having a selection of those achievements under the official name *Advances on Parallel and High-Performance Computing for Artificial Intelligence.*

Exploring the Selection

The present selection is meant to provide a snapshot of some of the latest work done in the confluence of AI and HPC. The collection we have made covers a small but illustrative choice of papers that clearly shows the importance of HPC in developing faster and powerful applications of Al. The papers in this special issue address the following topics:

- **Benchmarking**. Recent years witness a trend of applying large-scale distributed deep learning algorithms (HPC & AI) in both industry and scientific computing areas, which aim to speed up the training time to achieve a state-of-the-art quality. The HPC & AI benchmarks accelerate the process. Benchmarking HPC AI systems at scale raise serious challenges. The paper by Parés-Pont et al. touches on this topic.
- **Deep Learning Applications**. HPC & AI give the power to deal with ever growing complex models. More accurate predictions need to be fed by large amounts of data and they require large computing power, in return those systems create better applications. The paper by Manero and Béjar brings an application that may impact on the production of green energy.
- **Training Deep Neural Networks**. The paper by Torres et al. touches on the performed characterisation using a convolutional neural network implemented in TensorFlow and Pytorch. Likewise, the behaviour of the component interactions is discussed by varying the batch size for two sets of synthetic data.
- **An Exascale platform and AI**. The exascale platforms offer unique challenges to enabling functional hardware and software toolchains to manage vast volumes of data. In this regard, in their contribution, Fell et al. introduce the vision of MareNostrum Experimental Exascale Platform (MEEP), an open-source platform enabling HPC experimentation. One of the most exciting topics of this paper is the Accelerated Compute and Memory Engine (ACME) proposal. This tool facilitates the exploration of separating the computation from the memory operations and optimising the accelerator for dense (compute-bound) and sparse (memory-bandwidth bound) workloads that are very useful to AI workflows.

Conclusions

HPC and, in general, Parallel and Distributed Computing has become a pervasive utility, from supercomputer facilities and server farms containing multicore CPUs, GPUs and TensorFlow, to individual PCs, laptops, and new powerful mobile devices. AI research and industrial application heavily depend on parallel processing.

Today, as AI & HPC continue to transform an ever-increasing number of scientific disciplines and successful industrial applications at an expeditious pace, we can only imagine what the future holds once AI is powered with a rigorous mathematical framework. This is a promising innovation space for developers and hardware engineers to build the application, compiler, libraries and the necessary hardware to solve future challenging endeavours in the HPC, AI, ML, and DL domains.

The fast confluence of AI & HPC gives the means to address science, engineering, and industry challenges. It enables the creation of disruptive approaches for data-driven discovery and innovation. Realising these goals demands a combined effort between AI practitioners, HPC and specific domain experts.

Acknowledgements

The editors of this special issue would like to acknowledge and thank the external reviewers. They carried out a very professional, scientific and altruistic task in very short times and with very short deadlines.
Prof. U. Cortés (SNI-III) and Dr. U. Moya (SNI-I) are members of the Sistema Nacional de Investigadores, CONACyT (México).

References

1. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)

2. Deng, J., Dong, W., Socher, R., et al.: ImageNet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition, 20-25 June 2009, Miami, FL, USA. pp. 248–255. IEEE (2009), DOI: 10.1109/cvprw.2009.5206848

3. Goldie, A., Mirhoseini, A.: Reinforcement Learning for Placement Optimization. In: Proceedings of the 2021 International Symposium on Physical Design, 22-24 March 2021, Virtual Event, USA. pp. 5–5 (2021), DOI: 10.1145/3439706.3446883

4. Jimenez, D., Lin, C.: Dynamic branch prediction with perceptrons. In: Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, 19-24 Jan. 2001, Monterrey, Mexico. pp. 197–206. IEEE (2001), DOI: 10.1109/HPCA.2001.903263

5. Khailany, B., Ren, H., Dai, S., et al.: Accelerating chip design with machine learning. IEEE Micro 40(6), 23–32 (2020), DOI: 10.1109/mm.2020.3026231

6. Minsky, M., Papert, S.A.: Perceptrons: An introduction to computational geometry. MIT press (2017), DOI: 10.7551/mitpress/11301.001.0001

7. Nemirovsky, D., Arkose, T., Markovic, N., et al.: A general guide to applying machine learning to computer architecture. Supercomput. Front. Innov. 5(1), 95–115 (2018), DOI: 10.14529/jsfi180106

8. Sejnowski, T.J.: The unreasonable effectiveness of Deep Learning in Artificial Intelligence. In: Proceedings of the National Academy of Sciences. vol. 117, pp. 30033–30038. National Acad. Sciences (2020), DOI: 10.1073/pnas.1907373117

9. Waldrop, M.M.: Artificial intelligence in parallel. Science 225(4662), 608–610 (1984), DOI: 10.1126/science.225.4662.608

10. Wu, Z., Shen, C., Van Den Hengel, A.: Wider or deeper: Revisiting the ResNet model for visual recognition. Pattern Recognition 90, 119–133 (2019), DOI: 10.1016/j.patcog.2019.01.006

DOI: 10.14529/jsfi210102

Forecastability Measures that Describe the Complexity of a Site for Deep Learning Wind Predictions

Jaume Manero[1], *Javier Béjar*[1,2]

The application of deep learning to wind time series for multi-step prediction obtains good results at short horizons. The accuracy of a wind forecast is highly dependent on the specific structure of wind in the specific location, as many local features influence wind behaviour. The characterization of the complexity of a site for wind prediction is defined as forecastability or predictability and can be obtained from the inner structure of the meteorological time series observations from a site. We analyze the time series structure searching for properties that have a high correlation with the prediction result, properties that can create measures that have the potential to describe the forecastability of a site. The best measures will show a high correlation with the accuracy of the predictions. In this work, we analyze wind time series from 126,692 wind locations in the US, where we apply several deep learning methods first, and then we verify several forecastability descriptors with the accuracy deep learning results. We require High-Performance Computing (HPC) resources for this task as the deep learning algorithms have sensible resource requirements and are applied to a large set of data. The measures defined and explored in this work are based on several techniques that decompose or transform the wind time-series. By combining several of these measures, we can obtain better predictors of the site complexity, which will allow us to evaluate the future error of a prediction on this site. Forecastability measures can contribute to a wind site multi-dimensional description, becoming a valuable tool for wind resource analysts and wind forecasters.

Keywords: wind forecasting, time series, wind time series, deep learning, CNN, convolutional networks, forecastability.

Introduction

Wind-generated electricity is becoming a relevant component of the generation mix. While it keeps growing, some projections of future CO_2 *free* generation mix estimate a high wind generation dependency with percentages close to 40% in most countries [10]. These projections, which are a couple of decades away, are ambitious, but we can find countries which operate electricity systems with over 20% of wind-generated electricity, like Denmark (41%), Ireland (28%), Portugal (24%), Germany (21%) or Spain (19%) [22]. The largest global economies, the United States and China, have an installed capacity of 105 GW and 236 GW, respectively, accounting for over 50% of the total worldwide wind energy generation capacity, which has been 650 GW by the end of 2019.

Renewable energy has an intrinsic property, its intermittency, that challenges the stability of electrical systems. In a stable grid, demand must balance generation, which requires continuous forecasting to orchestrate the multiple energy generation origins to match the predicted demand. To predict wind energy output, we must predict the intensity of wind in the future, being wind a complex weather feature to model. Its formation depends on many local features, requiring meteorological models that work with a very high resolution not widely available today. When evaluating the potential of a wind generation location, we need to assess many dimensions, like the wind strength, the wind seasonality, the site accessibility, or the connectivity to the existing grid.

[1]Technical University of Catalonia, Barcelona, Spain
[2]Barcelona Supercomputing Center, Barcelona, Spain

The basic and possibly, essential feature is the wind intensity as it defines the potential amount of energy that can be generated. However, for the energy generated to become economically viable, it needs to be predictable. Otherwise, its value decreases. In a scenario where the balancing of demand and production is complex, and demand-response markets are appearing, the prediction requirements for a site are critical. There is noticeable economic value from accurately assessing the predictability of a geographical location for wind electricity generation [1].

(a) Wind Speed Mean (m/s) (b) Wind Speed Variance (c) Site Elevations (m)

Figure 1. Geographical representation of wind speed mean, wind speed variance and site elevation for the NREL dataset

Previous Works in Forecastabilty

Wind generators must commit their electricity generation in advance, and for this reason, they predict their park outputs several time-points in the future (like 6, 12, 24 hours). If the real output is not aligned with the prediction, the park owner is economically penalised by the electricity system operator. A wind park easy to forecast will obtain better returns than one that is difficult, and in this sense, we can affirm that good forecastability increases the returns of a wind site.

Forecastability has appeared in the literature on wind prediction literature recently. The definition of this novel term can be found in an article from Rogers [19], where he defines several measures that correlate with the wind prediction. In this article, the measures are defined as Measure-correlate-predict (MCP) algorithms, a definition that evolves in subsequent papers as forecastability. We can find more recent works in this area like:

Girard et al. analyzed in [9] how the forecast error economically impacts wind parks in Denmark, using as source information the penalties incurred by the different wind farm owners provided by the Danish transmission system operator (TSO). In this work, they conclude that, in the assessment phase, forecastability has low value for the producer in an isolated wind park but increases its value if several locations are grouped together or if the wind park is already in production. This work is focused on the producer side and does not analyze the impact that forecastability may have on the overall electricity system. The market imbalance cost reduction impacts the site assessment and the operation phases. This research found that it can also impact maintenance and downtime for off-shore wind farms, as a low prediction quality increases the complexity to find episodes of good weather to access the wind turbines at sea.

Sanz Rodrigo et al. in [11] go more in-depth in applying forecastability properties to the assessment phase of a wind park finding that the capacity factor is significantly more relevant than predictability. However, it has value to determine the stability of the system and thus its resilience. The study uses Denmark and Ireland data but proposes to include sites located in a much broader area creating a virtual clustering. In the example, by adding sites from France and Spain, the prediction error is reduced 10%.

A site can be described by a time series generated from past meteorological measures. As predictability depends on the local wind features and wind can be described by a time series, some inner properties of the time series can describe the forecastability, forming the basis for the approach followed in the next two works.

Feng et al. in [7] analyse the characterisation of the time series structure by applying decomposition, linearity analysis of entropy. This approach is used in a subsequent work [8] where the defined characterisations are applied to wind sites in North America, analysing the relationship of the uncertainty in forecastability to spectral entropy using regression approaches.

Article Structure and Objectives

This article explores and analyses some novel forecastability measures and analyses how their combination improves the characterisation of the prediction error. The prediction is obtained using several deep learning approaches at different time horizons.

The defined forecastability measures are defined from different time series properties, like series decomposition, spectral analysis or aggregation methods. We work with 103 individual measures, and we validate their forecastability by comparing them with deep learning predictions on the US geography. The data used is from the NREL Wind dataset, which has 126,692 wind sites in North America [6]. The amount of data used for predictions and the multiple validations have required the use of High-Performance Computing infrastructure.

This article offers several contributions, as follows:

1. A proposal of forecastability measures that can have a practical use to determine the forecasting complexity of a site.
2. Verify the accuracy of the proposed measures using 126,692 representative sites from all different typologies of wind locations.
3. Obtain combinations of measures with higher correlation and potential regression ability to each time series.

The remainder of the paper is organised as follows. Section 1 describes the National Renewable Laboratory (NREL) dataset. Section 2 reveals the deep learning prediction models applied to the data. Section 3 develops different forecastability measures and their mathematical foundation, and in Section 4 we describe the experimental organisation with the measure exploration approach, regression strategy and individual measure contribution to the predictions. The article closes with some conclusions and with some insights on future work.

1. The Wind Toolkit Dataset from the National Renewable Laboratory (NREL)

The NREL dataset is a wind dataset that contains wind time series, composed of several weather variables and created by the NREL laboratory in the US. Comprises 126,692 sites distributed evenly in the US geography, which are located in a grid of 2x2 km. Each time series is seven years long, from January 1st 2007 to December 31st 2013, sampled every 5 minutes (over 730,000 steps long) and contains the following observations for each site: wind speed, wind direction, humidity, pressure and temperature [6].

The dataset contains a high diversity of wind patterns as they cross many climatic regions, ranging from the seas (Atlantic and Pacific) to the mountain peaks in the Rockies. The highest site is number 125,659, located at the North Star Mountain in Colorado, at latitude:

39.38160216, longitude: –106.101776, above 3,000 m of altitude. Figure 1 illustrates a geographical representation of the wind-speed mean, wind-speed variance, and site elevation for the sites in the dataset, and we can see how different boundaries are drawn on each map, observing variability of winds or altitudes in the geography.

The NREL is a synthetic dataset, with data obtained from global meteorological models with some post-processing and cross-verified with real observed data. Its value lies in the large number of sites that allow experimentation on a wide diversity of wind time series. It is a large dataset and requires sizeable amounts of computing resources, but it provides the validation of results using a wide diversity of wind locations.

2. Deep Learning Methods Applied to Wind Prediction

Wind prediction can refer to the wind speed or electricity generated forecast (if applied to wind electricity generation turbines). There are many methods and approaches used for wind prediction. We can group them into two major categories, one set of methods based on weather prediction modelling and another set based on time series. For short-term prediction, time series are more accurate, and for longer-term ones, weather modelling is preferred.

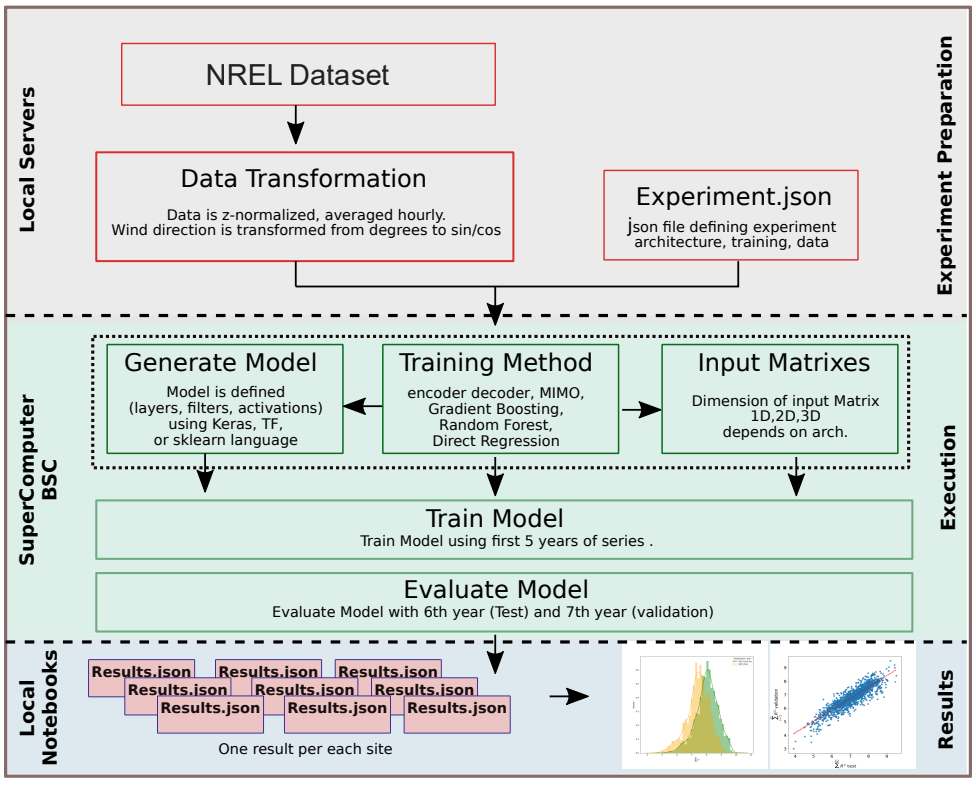


Figure 2. Experimental framework designed to apply several deep learning architectures to the NREL dataset

For this work, we will use deep learning prediction trained on wind time series. As wind is a complex phenomenon, the time series reflect this complexity, showing non-linearity and non-stationarity properties, which imply that the best performing forecasting algorithms must cope with both properties. Deep learning neural networks can model non-linear functions becoming a good candidate for the wind modelling.

There are several published results on the application of neural networks methods to wind forecasting using Multi-Layer Perceptron models (MLP) [12, 16, 20], Recurrent Neural Networks (RNN) [3, 13], and some are using Convolutional Networks (CNN) [23] (for a complete literary review see [15]).

In this work, we use a multi-step prediction approach. Multi-step consists of predicting a set of results in the future in one algorithm execution. The alternative, and more common, is the single-step that predicts a single point in the future. Multi-step prediction is not as widely researched as single-step but has some practical advantages over a single step [2].

We define an experimental methodology and a framework to design and validate the different deep learning architectures on the NREL dataset illustrated in Fig. 2. This methodology is based on three main steps or phases. First we transform the data for better neural network ingestion (z-normalisation, averaged) and we define an input model (based on .json structure) to define the architecture description (layer structure, kernels, regularisation strategies, neurons, training data structure).

Then we develop the implementation of each deep learning architecture using open-source deep learning frameworks (see Section 2.1) and then we train each one with five years of data and validate with two. The evaluation is made using different methodologies like MSE, R^2 or RMSE for each wind site and each predicted step (for example for 12 hours ahead we obtain 12 error measures), for an experiment, we obtain 126,692 values that combined are considered as a probability distribution.

In the last phase we analyse and validate each experiment for a site. For an horizon H is the sum of the individual error values for each prediction step, $\sum_1^H \hat{Y}_i$, and the result of an experiment across all the dataset is the average of the values from all sites (see Tab. 1)[3].

Table 1. R^2 (cumulative for all steps) for deep learning models multi-step prediction at different horizons (1 hour, 6 hours and 12 hours). Models abbr. are: CNN: Convolutional Neural Network, CNN-sep: Convolutional separable, MLP: Multilayer perceptron, RNN: Recurrent neural network

Model	Description	1h	6h	12h
MLP	Multi Layer Perceptron	0.82	4.39	7.25
RNN	Recurrent Neural Network	0.81	4.35	7.15
CNN	Convolutional Neural Network	0.81	4.40	7.23
CNN-sep	Convolutional non-separable Network	0.82	4.43	7.32

The deep learning architectures belong to one of the three main categories, MLP, CNN and RNN. The CNN combines convolutional layers with fully connected layers to generate the output sequence. RNN have several typologies, using encoder-decoder constructions, or just combining the RNN with fully connected layers (similar to the CNN).

The results and conclusions for the experimentation are the following (for a detailed description of the experiments see [16]).

[3]The code and some evaluation notebooks are available at http://github.com/castorgit/Articles-2020 and http://github.com/castorgit/Wind_code

- **RNN**: Recurrent Neural Networks are the architecture of choice for sequences, however, the best models use small input sequence windows of fixed size, and for this problem the CNN and MLP obtain better results.
- **MLP**: Multi-layer perceptrons obtain good results with architectures 2 or 3 layers depth with 512–1024 neurons each. Deeper architectures do not obtain better results.
- **CNN**: The convolutional 1-dimensional standard operation obtains better performance than the MLP, and the separable convolution shows the best results (see [4]), with filters (3x3, 9x9) and 2 or 3 layers depth.

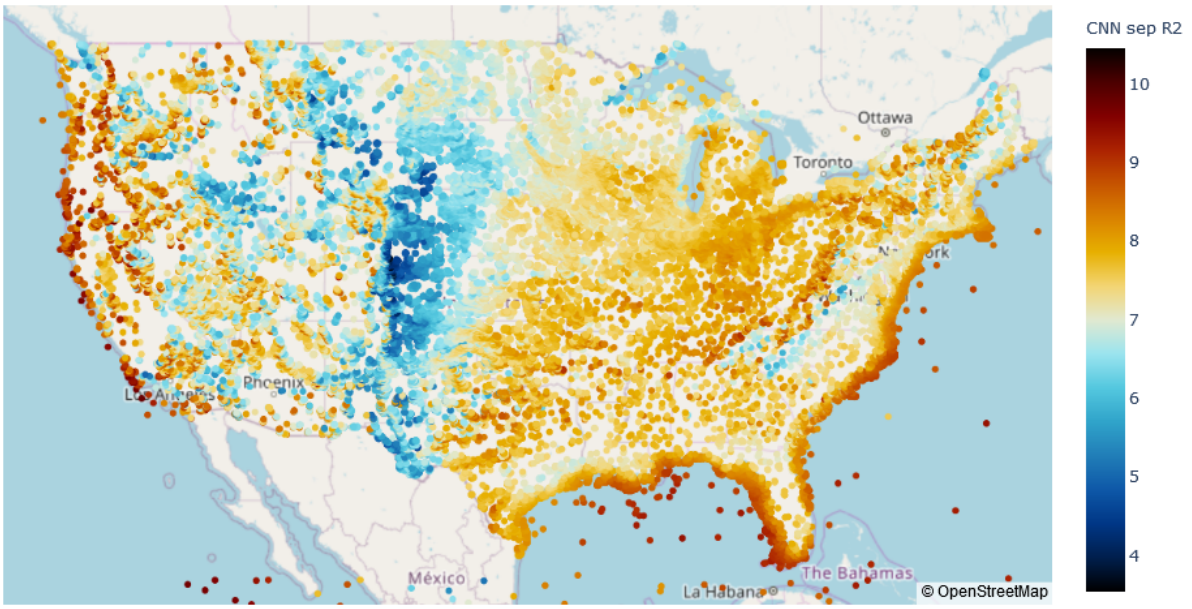


Figure 3. Accuracy of convolutional separable networks applied to the NREL dataset - blue less accurate, dark red more accurate

Table 1 shows the results of 4 main (RNN, MLP, CNN and CNN separable) architectures, that have been validated for prediction on 3 different horizons (1, 6 and 12h ahead).

The results, in this article, are always represented using R^2, which is the coefficient of determination, which is a good measure to rate the accuracy of a regression, it is a number in the interval $[-\infty..1]$ which is the ratio between the explained variation and the total variation from the regression.

$$R^2 = 1 - \frac{\sum_i^N (y_i - \widehat{y})^2}{\sum_i^N (y_i - \bar{y})^2} \tag{1}$$

An important conclusion from the experiments with the DL experiments is the high correlation between the accuracy results from the different architectures. The Pearson correlation between the deep learning methods is > 0.95, which shows that the site wind pattern structure is more important than the DL specific approach to determine the complexity of a site or, formulated in a different way, that the DL architectures have better results in easy sites and worse results in difficult places. In this research we will try to understand what makes wind-time series complex for prediction.

To improve clarity in the analysis in the next sections we use as representative the results from the best performing architecture, which correspond with the CNN separable architecture that uses a 1-dimensional separable convolution in 2 layers.

2.1. Technical Assessment and HPC Requirements

From the software point of view the deep learning experimental architectures have been developed from scratch using Python 3.6, and using some add-on standard packages, being the most relevant:

- machine learning platform: Tensorflow 1.14.0;
- deep learning library: Keras 2.2.4;
- Scikit-learn machine learning library: 0.21.4;
- statistical support: statsmodels 0.11.

The number of sites (126,692) and the length of the time series (7 years of data) are optimal for an extensive experimentation with deep learning, however, the computer requirements recommended the use of an HPC resource. Thanks to the availability of such infrastructure we performed the experimentation on almost one-hundred alternative DL architectures.

The experiments have been supported by a NVIDIA GPU based computer at the Super Computing Centre [17]. The HPC resource used is the Minotauro cluster which is build with BULL and it has 39 bullx R421-E4 servers, each composed of:

- 2 Intel Xeon E5-2630 v3 (Haswell) 8-core processors (each core at 2.4 GHz, and with 20 MB L3 cache);
- 2 K80 NVIDIA GPU Cards;
- 128 GB of Main memory, distributed in 8 DIMMs of 16 GB – DDR4 @ 2133 MHz - ECC SDRAM;
- 1 PCIe 3.0 x8 8GT/s, Mellanox ConnectX-3FDR 56 Gbit;
- 4 Gigabit Ethernet ports.

The full HPC machine provides a Peak Performance of 250.94 TERAFLOPS distributed as 226.98 TERAFLOPS (K80) + 23.96 TERAFLOPS (Haswell).

3. Basic Forecastability Measures

In this section we describe forecastability measures that come from four major categories, basic statistics Section 3.1.1, time series decomposition Section 3.1.2, aggregation and stability Section 3.1.3 and spectral analysis Section 3.1.4. First we describe the theoretical foundation in Section 3.1 of each one and then we describe the experimentation in Section 3.2.

3.1. Theoretical Foundation of the Basic Forecastability Measures

If we plot the prediction error (measured in R^2) from the convolutional separable architecture across all US sites, we obtain Fig. 3. In this illustration, we can see how the accuracy results clearly define different geographical wind regions. We can observe how the rocky mountains split the map in two, with a turbulent area in the plains and more stationary wind regions by the seas (either in the Pacific or in the Atlantic oceans). With this geographical mapping in mind, we can see how prediction defines groups of sites, pointing to a relationship between the site time series structure and the complexity (or accuracy) of the prediction. This idea guides this work as we try to find the relationships between the internal structural elements in the time series and the prediction error obtained with the DL algorithms.

We perform an exploration of measures in four steps, first we use some basic statistic characterisation measures, like mean or variance, then we verify the use of time series decomposition,

we design some new approaches using aggregation and combination models and we finalise with entropy analysis.

We split this chapter in two sections, the first section contains the theoretical descriptions for each measure, and the second section describes the experimentation using these methods and the results obtained on the NREL dataset.

3.1.1. Basic statistic descriptive indexes

We consider basic descriptive indexes as the ones that have a simple statistical formulation, like mean, median, variance or standard deviation σ. We include elevation in this group, to see how the location altitude influences the prediction result.

In Fig. 1 we can see a geographical plot of Mean, Variance and Elevation of wind speed. In this map we can see how the measures define boundaries between different wind regions, but we need to understand how these measures are related with the prediction accuracy.

3.1.2. Measures based on time series decomposition techniques

If a weather phenomena repeats in cycles, it generates some kind of seasonal pattern in the wind time series, like a day-night flow, or heat-cold changes, or seasonal modifications of wind over the years. A good forecastability predictor will identify these patterns and, if for locations with high seasonality, the predictability measure can be quite effective.

In a time series, seasonality is identified by finding sub-sequences with a large auto-correlation coefficient. To find if a series has seasonal patterns, the best way is to decompose the series in components, using methods that split the series into different components.

Time series decomposition can be additive or multiplicative, depending on the operation (sum or multiplication) used to combine the individual elements of the series as can be seen in (2, 3).

Additive decomposition is the best choice if the time series does not hide a growth or decrease trend over time. When the variation of the pattern is proportional to the mean or level of the series then the multiplicative can be more appropriate, this happens with economic time series which show increases and decreases proportional to time.

Additive decomposition is more aligned with wind time series, as the mean of a wind time series is usually constant over time. For this reason the model applied in this work is the additive approach.

The objective in the decomposition is to split the series W_t in the three components, seasonal (S_t), trend (T_t) and residual (E_t). Depending on the type, the series can be formulated like:

$$W_t = S_t + T_t + E_t \qquad \text{additive} \tag{2}$$

$$W_t = S_t \times T_t \times E_t \qquad \text{multiplicative} \tag{3}$$

There are several decomposition methods. The classical method is the moving average, based on using the moving average on the series to isolate the trend-cycle, after removing the seasonality in the series (by applying filters or other approaches). In this work we name this method as the 'classical' approach W_{tc}, S_{tc}, T_{tc}, E_{tc}.

The STL (Seasonal and Trend decomposition using LOESS) is another widely used method for time series decomposition [5]. The LOESS method (locally estimated scatterplot smoothing)

uses the LOESS smoothing algorithm to extract smooths estimates of the three components, this method requires to specify the seasonal period.

A refinement of the decomposition consists in calculating the strength of the trend f_T or seasonality f_S in the time series, following the framework proposed by Wang et al. in [24].

For data with strong trend, the seasonality adjusted data (deseasonalized data) $W_t - S_t$ has more variation than the residual component, for data without trend the two variances would be the same. In this sense we define the strength of trend as:

$$f_T = 1 - \frac{var(E_t)}{var(W_t - S_t)}. \tag{4}$$

Strength of seasonality is defined in the same way, but in this case the variance used is the detrended data, $Wt - T$

$$f_S = 1 - \frac{var(E_t)}{var(W_t - T_t)}. \tag{5}$$

A time series with seasonality strength f_S equal to 0 has no seasonality, and when f_S is close to 1 it is an indicator for strong seasonality.

3.1.3. Measures based on aggregation and stability properties

Other measures that can be defined on fub-series combination features. In this section we describe methods that are based on tiled (non-overlapping) windows.

The first one is stability *Stab*, consisting of the variance of the means, The second one is lumpiness *Lump* or the variance of all the windows variances.

Stab and *Lump* are calculated obtaining sub-series of length *period* from the complete series (usually 12h, 24h, 3m, 6m), where *Lump* is the mean of each sub-series and *Stab* is the standard deviation.

$$Lump_{(P)} = \frac{1}{(N-P)} \sum_{j=1}^{N-P} \left(\frac{1}{P} \sum_{i=j}^{j+P} x_i \right), \tag{6}$$

$$Stab_{(P)} = \frac{1}{(N-P)} \sum_{j=1}^{N-P} \left(\sqrt{\frac{1}{P} \sum_{i=j}^{j+P} (x_j - \bar{x})^2} \right), \tag{7}$$

$$\text{where } N = \frac{\text{Series length}}{Period} \tag{8}$$

In this experimentation we calculate Lump and Stab for periods of 12, 24 hours, one week, one, three months and six months.

3.1.4. Spectral entropy analysis measures for wind time series

Entropy is a measure of the uncertainty of a random variable. In time series it can be used as a property to quantify the series, and it has been used in wind time series to decompose the signal and improve forecasting [21].

In this work we have used two different entropy calculations approaches, sample entropy and spectral entropy.

Sample entropy (named *SampEnt* in this work for convenience) assesses the complexity of the time series, a large value indicates high complexity while a small one indicates low complexity or more regular series. It was initially created for physiological time series signals [18].

$$H(x, m, r) = -log\left(\frac{C(m+1, r)}{C(m, r)}\right), \tag{9}$$

where m is the embedding dimension (= order), r is the radius of the neighbourhood (default = $0.2\text{std}(x)$), $C(m+1, r)$ is the number of embedded vectors of length $m+1$ having a Chebyshev distance inferior to r and $C(m, r)$ is the number of embedded vectors of length m having a Chebyshev distance inferior to r.

Spectral entropy (*SpecEnt* in this article) is defined to be the Shannon Entropy of the Power Spectral Density (PSD) of the data:

$$H(x, sf) = -\sum_{f=0}^{f_s/2} PSD(f) log_2[PSD(f)], \tag{10}$$

where PS is the normalised PSD, and f_s is the sampling frequency.

3.2. Experimentation with the Basic Forecastability Measures

We have performed a set of experiments.

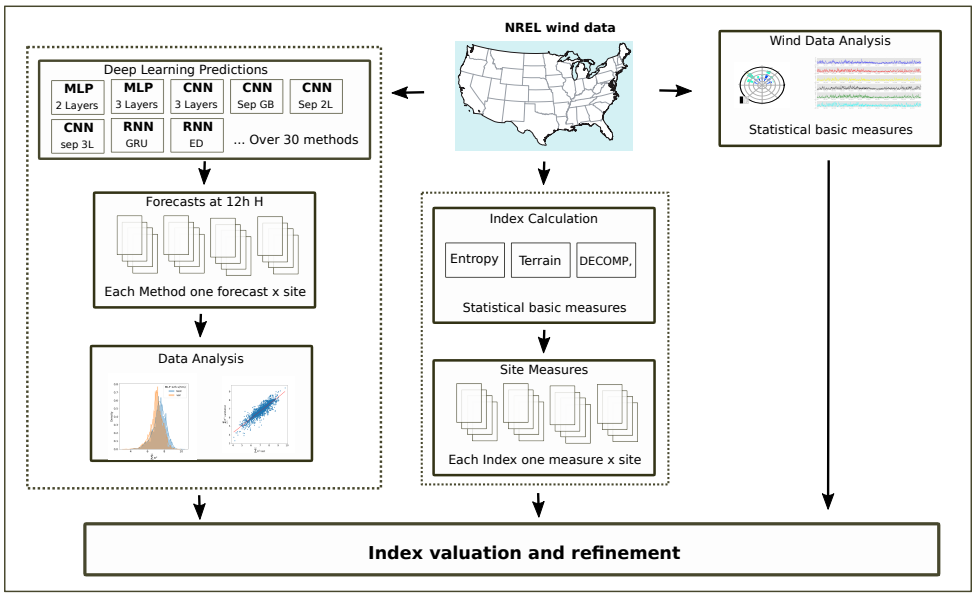


Figure 4. Experimental framework for forecastability measures validation

3.2.1. Correlations with the basic statistical measures

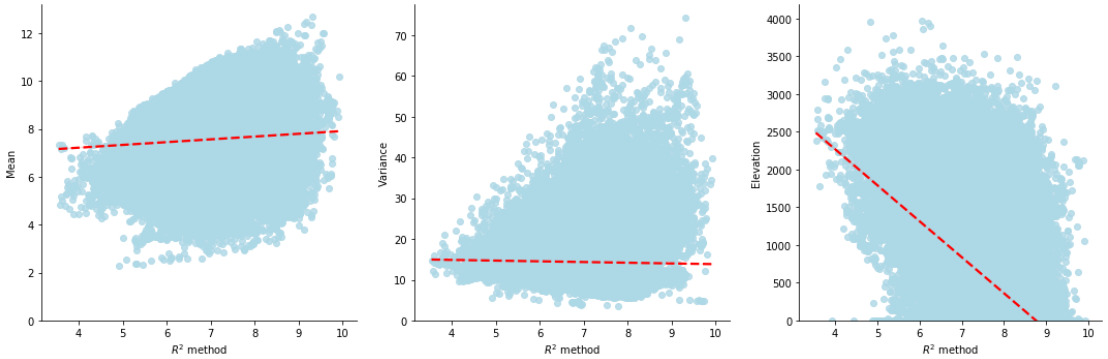
In Tab. 2 we show the correlations between three basic measures (wind-speed mean, wind-speed variance and site elevation) and the DL prediction.

The more significative correlation is with the wind-speed mean, and at lower level with the variance and the site elevation.

Table 2. Correlations between basic statistical measures and a DL prediction with a 12h ahead horizon

Measure	Pearson Correlation with DL prediction
Wind-speed variance	0.3113
Wind-speed mean	–0.5913
Terrain elevation (site altitude)	–0.2759

Representing the relationship between the measures and the predictions we obtain three scatterplots (see Fig. 5) where we can observe visually how the different measures adjust to the prediction.



(a) Mean vs DL prediction (b) Variance - DL prediction (c) Site elevation - DL

Figure 5. Scatterplots of Wind-speed mean, Wind-speed variance and site elevation with DL (best model convolutional separable network)

3.2.2. Correlation results between decomposition aggregations and spectral measures and DL prediction

We apply the different method approaches to each site, and then we calculate the Pearson correlation between each measure and the deep learning prediction. The results from all the different approaches are presented in Tab. 3.

We use two different decomposition methods, the STL (using LOESS) or the Least Squares. Both methods obtain quite similar results with some better correlations using the STL, because the smoothing method is more sophisticated and fits better the specific characteristics of wind data. We can see the comparison between the two decomposition methods in Fig. 6, where the scatterplots for f_T and f_S are presented side by side. In both cases using the STL method obtains better adjustment to the prediction, with higher correlation. The best correlations are obtained with 12 and 24 hours periods.

It is remarkable how the strength of Trend obtains a correlation of 0.888 which is extremely high, which points very good forecastability properties.

In Fig. 6 we illustrate with a scatterplot the relationship between the reference Deep Learning prediction and the f_S and f_T calculated with the LOESS and Least Squares methods, and in Tab. 3 the values are calculated on the NREL dataset.

Table 3. Pearson Correlations between Time Series decomposition measures (f_t strength trend and f_s strength seasonality), time horizons and DL Prediction across 126,692 sites

Decomposition						
	STL			Least Squares		
	12h	**24h**	**1m**	**12h**	**24h**	**1m**
Strength Trend f_T	0.834	0.888	0.473	0.812	0.841	0.420
Strength Seasonal f_S	–0.199	0.153	–0.042	–0.151	0.110	–0.012

Aggregation / Combination						
	12h	**24h**	**1w**	**1m**	**3m**	**6m**
Lump	0.247	0.282	0.324	0.352	0.346	0.117
Stab	–0.282	–0.058	0.110	0.075	0.038	–0.086

Spectral Analysis	
Spectral Entropy *SpecEnt*	–0.835
Sample Entropy *SampEnt*	–0.641

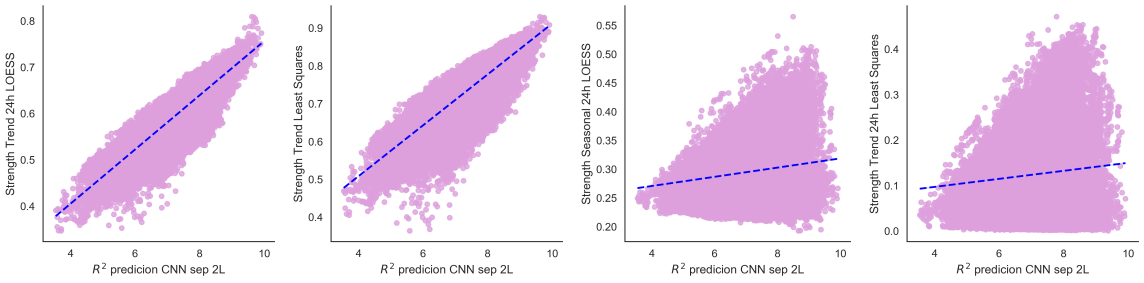
In the scatterplot figures we can observe how the LOESS model generates a cloud of ponts slightly better adjusted than the Least squares, and how the strength trend S_t obtains a higher correlation using the LOESS method versus the strength seasonality in a 24 hours period using the least squares method (Pearson correlation 0.888 vs 0.841) (see Fig. 7).

When it comes to aggregation measures we find different values depending on the period, being the most effective periods 1 week, 1 month and 3 months.

With Spectral analysis we obtain a high correlation with Spectral Entropy *SpecEnt* where the observed correlation is –0.835 and for *SampEnt* (sample entropy) is –0.641 (see Tab. 3 and Fig. 7).

3.2.3. Discussion on the basic forecastability measures results

We can observe the existence of correlations between basic measures and prediction, the strongest correlations are found in f_T and *SpecEnt*. The correlations are high and close to 0.8. However, we can see that both of them have a set of sites that seem not to be inside the cloud

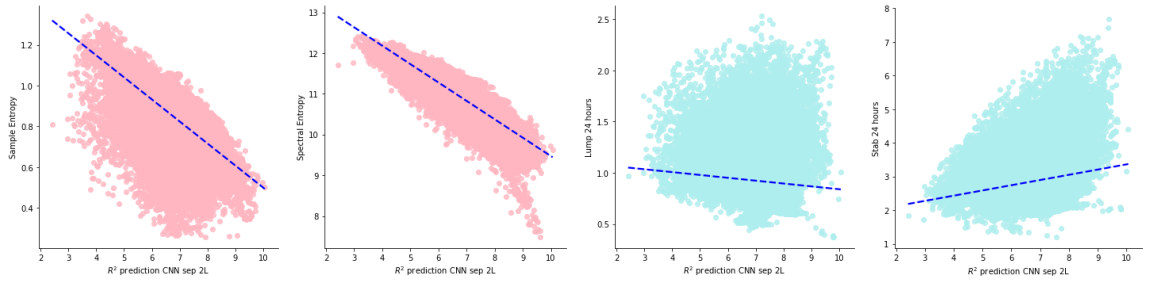


(a) f_t STL vs DL (b) f_t L.Squares vs DL (c) STL f_s vs DL (d) L.squares f_s vs DL

Figure 6. Scatterplot of decomposition measures for f_T and f_S using two alternative methods, the STL and Least Squares. The X axis shows the DL prediction

points, that can be seen as a tail in the scatterplot figures (see Fig. 6 and Fig. 7). Analysing the outlier points we can see that the points do not belong to a particular area in the map, but are randomly distributed over the map. We propose some further analysis on these points to identify what makes them to have low correlation.

Aggregating a combination of measures obtains results with lower correlations, contributing to the defintion of better forecastability measures. The following sections analyse this contribution and how the combination by regression can obtain better predictability.



(a) SampEnt vs DL (b) SpecEnt vs DL (c) Lump24h vs DL (d) Stab24h vs DL

Figure 7. Spectral Analysis measures compared to deep learning prediction (CNN separable 2 layers) with horizon 12 hours ahead

4. Defining Complex Forecastability Measures

In this section we combine elementary measures to improve the forecastability of the results. To identify the best combinations we use the following methodology. In Section 3.2 we analyse the individual correlations, discussed in Section 3.2.3, after this preliminary analysis, in this section, we explore all the possible individual correlations in Section 4.1 and explore the measure importance using Principal Component Analysis (PCA) in Section 4.2. Then we build combinations using regression that obtain the highest predictive value in Section 4.3.

4.1. Analysis of Correlations with All the Individual Measures

In Section 3.2.1 we have analysed the correlation between the wind-speed variable and its transformation with the prediction. Now in the analysis we include all the possible defined measures which are the result of combining the measures *Lump*, *Stab*, f_T, *SpecEnt* with different periods (24h, 12h, 1 month, 3 months, 6 months), for each variable (wind_direction_sin, wind_direction_cos, temperature, pressure, density, wind_speed). The correlation matrix or confussion matrix can be seen in Fig. 8. We visualise high correlation clusters in this figure, The wind speed measures show a cluster, and the rest of measures (wind_direction, pressure, temperature and density in another one. Dark green areas show negative correlation,

Our conclusion, after analysing this figure, is that we can complement the wind_speed measures with other, but due to high correlation between pressure, density, direction or temperature, we do not need to choose the whole set, with a reduced representation we can obtain all the information required for the combined measure exercise.

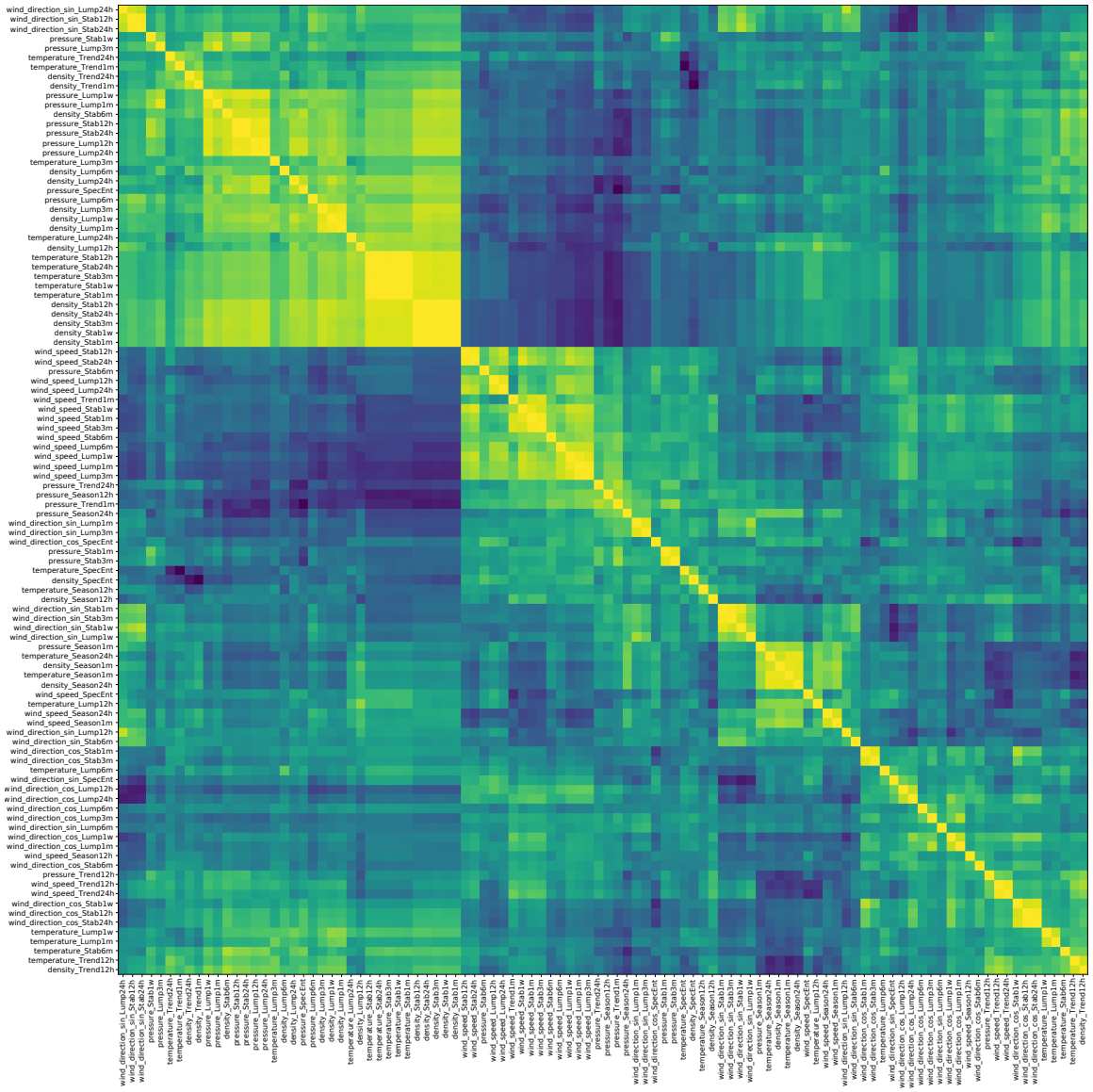
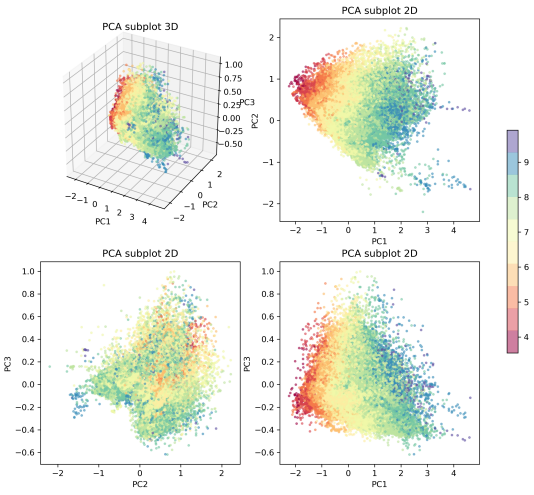


Figure 8. Correlations between measures

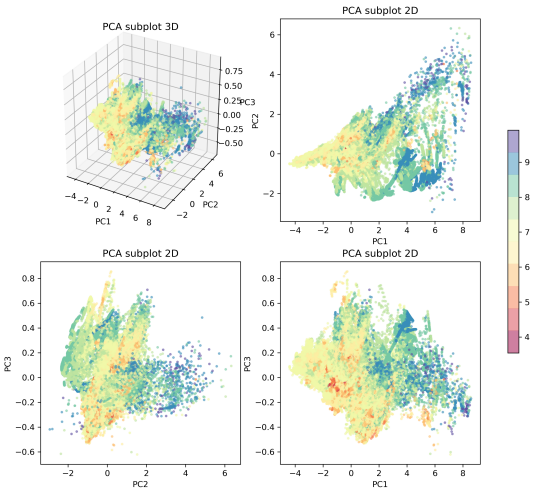
4.2. PCA Analysis and Measure Importance

To identify the variance contribution we perform a PCA analysis using the dimensions wind-speed, temperature and density, and using the prediction at 12h ahead horizon for the graphical representation (see Fig. 9). As a further exploration for the measure combination we perform a Principal Component Analysis (PCA), this method allows to reduce the dimensionality of the measures (considered as features). In this exploration we use measures based on wind speed, temperature and wind density. The PCA component decomposition in 3 dimensions reduces the dimensionality of the measures into 3, the PCA components show the underlying structure in the data by finding a set of axis (2-dimensional or 3-dimensional) that when applied to the data we maximize the data variance.

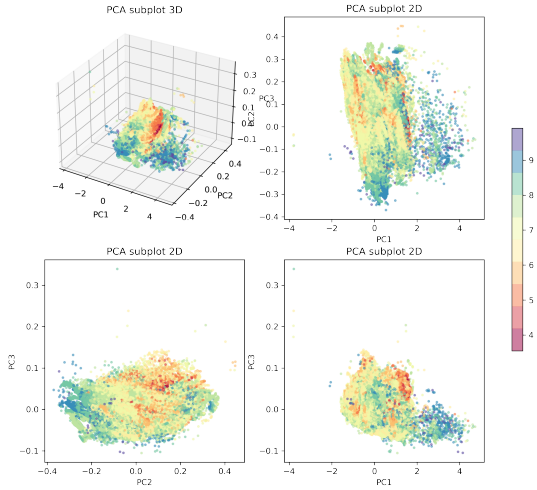
The amount of variance explained by the PCA on the three dimensions is [0.68504753 0.26162715 0.03701629], which amounts for a 94.67% with the first two components and 98.37% with the three components, showing that just with the two dimensions PC1 and PC2 we cover most of the variance.



(a) Wind speed & prediction 12h



(b) Temperature & prediction 12h



(c) Density & prediction 12h

Figure 9. 3D PCA scatterplots using wind_speed, temperature, density, with several measurs (*Lump*, *Stab*, f_T, *SpecEnt*) to identify the variance contribution on each dimension

We will use the PC1, and PC2 components in some of our regressions, discarding PC3 as it has low informative value.

4.3. Regression based on Multiple Measures

The next step consists of building a predictive measure, by combining the best features, that will have the highest forecastability capability of the wind prediction result.

This predictor allows to characterise the time series 'forecastability', where a high value determines that the series has potential for an accurate prediction using deep learning and a low result points to a time series that will have low accuracy prediction.

We consider the predictor as a function built using individual features or characterisation measures (based on decomposition or spectral analysis). This function can be considered a regression that combines each one of the individual indexes. To develop this function we try several regression models (see Tab. 4) using different techniques. Firstly we try regressions using several measures as regression variables, these models use one or several measures. Secondly we

we introduce some non-linearity by applying a support vector machine approach. We test all the models with the deep learning representative (CNN convolutional with separable convolutions) crating several models, each one with more information and variables.

The first regression model is based on the two measures that show the highest correlation with the prediction ($SapEnt$ and f_T) (see Fig. 10a).

We can consider a multi-variable x_i regression model to be formulated as:

$$y = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_n x_{in} + \varepsilon, \tag{11}$$

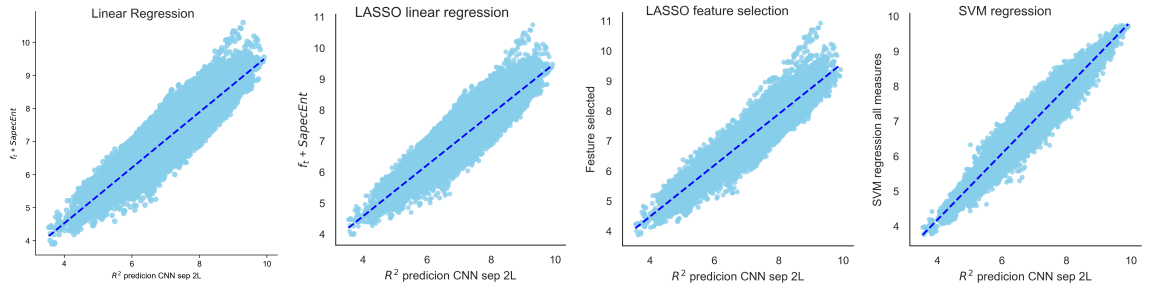
where the measures correspond to the x_i variables. To obtain the coefficients β_i, we transform the the objective into an optimisation problem where the goal is to reduce the residual ε term (sum of squares between the target and the regression). The specific optimisation technique defines the nature of the regression method, and it may influence the accuracy of the result. For this work we try two methods, the OLS (Ordinary Least Squares) and the LOESS (Locally Estimated Scatterplot Smoothing). The first optimisation algorithm applied is the OLS (see Fig. 10a). Using this approach and combining the two most relevant measures f_t and $SpecEnt$ we obtain results that are between 0.82 and 0.84 R^2 on the different horizons (see Tab. 4). Using only the two best measures we obtain a prediction result with a correlation over 0.9 and $R2$ of 0.84.

Correlation is not a good measure to validate a regression, and other measures are better tailored for this task. In this case we are using the R^2 or coefficient of determination (described in 1. We keep regression to have a reference with the individual measures, but the quality of the regression must be analysed using the R^2 in this section and in Tab. 4.

An R^2 result of 0.84 is very high, and can be interpreted as a relevant approximation. We illustrate this result in Fig. 10a with a scatterplot where the x axis is the regression result, and the y axis is the original prediction. We observe that the obtained cloud of points is quite adjusted to the optimal line, showing a better fit than the plot of the individual measures (see Fig. 7).

The second model is obtained using an alternative regression approach, the LASSO approach with two variables, in this case the result is very similar, with a correlation and R^2 equal to the least squares. LASSO is widely used and performs the regression by penalizing the sum of the absolute values of the weights β_j, it has the property to reduce the weight of some features, simplifying the model, however in this application as we use only two the results are similar to the OLS. As we can see in Fig. 10a and in Fig. 10b the scatterplots are quite similar.

For the third approach we use the LASSO properties for feature selection, to use all the measures and restrict the less relevant. We include in this regression measures with lower correlation



(a) Linear regression with SpecEnt and f_T

(b) Loess Regression with only SpecEnt and F_T

(c) Lasso Regression with feature selection

(d) SVM regression with all measures

Figure 10. Comparison of regressions using the best individual features

(like *SampEnt*, *Lump* or *Stab*) and the LASSO models the regression with all the features, the final model assigns low weights to the less relevant features, ($Lump1w$, $Trend12h$, $Stab1w$) but extracts information from the rest.

This result obtains a correlation of 0.94 with an R^2 of 0.88, showing results that have very high predictability character. The scatterplot in this case (see Fig. 10c) shows a slightly shrinked cloud of points.

As the regression does not improve by linear methods, we use a SVM Support Vector Machine. SVM are well known models proposed by Vapnik able to represent non-linearity by representing the data into a higher dimensional space. To do so we use a kernel function that performs the transformation. The kernel used for this regression is a Gaussian RBF (Radial Basis Function). We have very good results obtaining a coefficient R^2 of 0.950 and a correlation of 0.975 (for 12 hours ahead horizon), being the best regression by far. In Fig. 10d we illustrate the scatterplot of this regression were we can see how the cloud of points adjusts closer compared to the linear regressions.

Table 4. Pearson Correlations and R^2 values for regressions and measures (only wind speed)

Prediction	**1h**		**6h**		**12h**	
Evaluation	**corr**	R^2	**corr**	R^2	**corr**	R^2
Strength Trend f_t	0.92	-	0.90	-	0.83	-
SpecEnt Spectral Entropy	–0.71	-	–0.77	-	–0.83	-
Regression 2 measures	0.90	0.82	0.92	0.85	0.92	0.84
LASSO 2 measures	0.91	0.82	0.92	0.85	0.92	0.84
LASSO all measures	0.96	0.92	0.94	0.89	0.94	0.88
SVM	0.98	0.95	0.98	0.97	0.97	0.95

Conclusions and Future Work

In this article we propose several forecastability measures based on time series properties. We propose these measures as predictors of the accuracy of a method used on this site (as wind time series are originated on a single location). We propose the different measures and we build a regression model combining them that shows an almost perfect fit with the prediction (R^2 of 0.95).

We consider predictions performed using deep learning architectures and the comparisons are made with the best performing approach the convolutional separable models (see [14]). The predictions and measures are performed in the largest wind dataset available, the NREL dataset (126,691 sites), which allows to process a wide set of wind examples, but requiring sizeable resources that have been provided by an HPC infrastructure.

The conclusions from this experimentation are:

- Time series decomposition is a powerful tool to create measures that are correlated with prediction, in this way we find the f_T index based on the series trend that has a very high correlation with the predictions.

- Applying Signal Analysis decomposition we obtain a measure *SpecEnt* that has a very high correlation, and another measure *SampEnt* which has shown some valuable correlation with prediction.
- Two new measures defined using entropy analysis, *Lump* and *Stab*, have some correlation with the predictions.
- The combination of several measures using a linear regression approach, or a non-linear SVM obtains correlations very close to 1 with R^2 results over 0.95.

With these results, we can conclude that there are forecastability measures highly adjusted to deep learning predictions. In this way, we can obtain a prior evaluation of a time series predictability, that allows us to understand the complexity of the prediction for this site and therefore the economic value for energy commitment to the grid. Wind forecasters, wind farm developers and power system operators can benefit from the use of these predictor measures that characterise time series from a specific location. Adding these measures to the tool-set offers a new powerful characterisation tool. We can rate how easy or complex is a site before any prediction is performed.

As future work we propose to evaluate new combinations of individual indexes using linear and non-linear modelling and to apply finance modelling to the indexes to convert forecastability measures into economic values that allow the wind speed industry to take decisions based on site evaluation wind predictability.

Another possibility is to work in the definition of deep learning architectures that adapt themselves depending on the forecastability results for a specific location, in this way the prediction algorithm self-adapts to the data characteristics thus improving the result accuracy.

Acknowledgements

The authors would like to thank the Barcelona Supercomputing Center (BSC) for the usage of their resources and the United States National Renewable Laboratory (NREL) for the use of its Wind Toolkit (wind datasets). We would also like to thank the anonymous reviewers for providing valuable comments that helped to improve the quality of this paper.

References

1. Bastian, J., Jinxiang Zhu, Banunarayanan, V., Mukerji, R.: Forecasting energy prices in a competitive market. IEEE Computer Applications in Power 12(3), 40–45 (1999), DOI: 10.1109/67.773811

2. Ben Taieb, S., Bontempi, G., Atiya, A.F., Sorjamaa, A.: A review and comparison of strategies for multi-step ahead time series forecasting based on the NN5 forecasting competition. Expert Systems with Applications 39(8), 7067–7083 (2012), DOI: 10.1016/j.eswa.2012.01.039

3. Cao, Q., Ewing, B.T., Thompson, M.A.: Forecasting wind speed with recurrent neural networks. European Journal of Operational Research 221(1), 148–154 (2012),

DOI: 10.1016/j.ejor.2012.02.042

4. Chollet, F.: Xception: Deep learning with depthwise separable convolutions. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 21-26 July 2017, Honolulu, HI, USA. pp. 1800–1807. IEEE Computer Society, Los Alamitos, CA, USA (2017), DOI: 10.1109/CVPR.2017.195

5. Cleveland, W.S.: Robust locally weighted regression and smoothing scatterplots. Journal of the American Statistical Association 74(368), 829–836 (1979), DOI: 10.1080/01621459.1979.10481038

6. Draxl, C., Clifton, A., Hodge, B.M., McCaa, J.: The Wind Integration National Dataset (WIND) Toolkit. Applied Energy 151, 355–366 (2015), DOI: 10.1016/j.apenergy.2015.03.121

7. Feng, C., Chartan, E.K., Hodge, B.M., Zhang, J.: Characterizing time series data diversity for wind forecasting. In: Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, 5-8 Dec. 2017, Austin, Texas, USA. pp. 113–119. Association for Computing Machinery, New York, NY, USA (2017), DOI: 10.1145/3148055.3148065

8. Feng, C., Sun, M., Cui, M., Chartan, E.K., Hodge, B.M., Zhang, J.: Characterizing forecastability of wind sites in the United States. Renewable Energy 133, 1352–1365 (2019), DOI: 10.1016/j.renene.2018.08.085

9. Girard, R., Laquaine, K., Kariniotakis, G.: Assessment of wind power predictability as a decision factor in the investment phase of wind farms. Applied Energy 101, 609–617 (2013), DOI: 10.1016/j.apenergy.2012.06.064

10. Jacobson, M.Z., Delucchi, M.A., Bauer, Z.A., et al.: 100% clean and renewable wind, water, and sunlight all-sector energy roadmaps for 139 countries of the world. Joule 1(1), 108–121 (2017), DOI: 10.1016/j.joule.2017.07.005

11. Javier, S.R., Frías Paredes, L., Girard, R., et al.: The role of predictability in the investment phase of wind farms. In: Renewable Energy Forecasting: From Models to Applications, chap. 14, pp. 341–357. Woodhead Publishing Series in Energy, Elsevier - Woodhead Publishing (2017), DOI: 10.1016/B978-0-08-100504-0.00014-7

12. Li, G., Shi, J.: On comparing three artificial neural networks for wind speed forecasting. Applied Energy 87(7), 2313–2320 (2010), DOI: 10.1016/j.apenergy.2009.12.013

13. Liu, Z., Gao, W., Wan, Y.H., Muljadi, E.: Wind power plant prediction by using neural networks. In: IEEE Energy Conversion Congress and Exposition (ECCE), 15-20 Sept. 2012, Raleigh, NC, USA. pp. 3154–3160. IEEE (2012), DOI: 10.1109/ECCE.2012.6342351

14. Manero, J.: Deep learning architectures applied to wind time series multi-step forecasting. Ph.D. thesis, Technical University of Catalonia UPC. Department of Computer Science (2020), http://hdl.handle.net/2117/328183

15. Manero, J., Béjar, J., Cortés, U.: Wind energy forecasting with neural networks. a literature review. Computación y Sistemas 22, 1085–1098 (2018), DOI: 10.13053/CyS-22-4-3081

16. Manero, J., Béjar, J., Cortés, U.: "Dust in the Wind...", Deep Learning application to Wind Energy time series forecasting. Energies 12(12), 2385 (2019), DOI: 10.3390/en12122385

17. Martorell, J.M.: Barcelona Supercomputing Center: Science accelerator and producer of innovation. Contributions to Science 12(1), 5–11 (2016), DOI: 10:2436/20.7010.01.238

18. Richman, J.S., Moorman, J.R.: Physiological time-series analysis using approximate entropy and sample entropy. American Journal of Physiology-Heart and Circulatory Physiology 278(6), H2039–H2049 (2000), DOI: 10.1152/ajpheart.2000.278.6.H2039

19. Rogers, A.L., Rogers, J.W., Manwell, J.F.: Comparison of the performance of four measure-correlate-predict algorithms. Journal of Wind Engineering and Industrial Aerodynamics 93(3), 243–264 (2005), DOI: 10.1016/j.jweia.2004.12.002

20. Shi, J., Guo, J., Zheng, S.: Evaluation of hybrid forecasting approaches for wind speed and power generation time series. Renewable and Sustainable Energy Reviews 16(5), 3471–3480 (2012), DOI: 10.1016/j.rser.2012.02.044

21. Sun, W., Wang, Y.: Short-term wind speed forecasting based on fast ensemble empirical mode decomposition, phase space reconstruction, sample entropy and improved back-propagation neural network. Energy Conversion and Management 157, 1–12 (2018), DOI: 10.1016/j.enconman.2017.11.067

22. Walsh, C., Pineda, I.: Wind energy in Europe in 2018 trends and statistics. http://windeurope.org/about-wind/statistics/european/wind-energy-in-europe-in-2018/, accessed: 2021-01-09

23. Wang, J., Zong, Y., You, S., Trholt, C.: A review of Danish integrated multi-energy system flexibility options for high wind power penetration. Clean Energy 1(1), 23–35 (2017), DOI: 10.1093/ce/zkx002

24. Wang, X., Smith, K., Hyndman, R.: Characteristic-based clustering for time series data. Data Mining and Knowledge Discovery 13(3), 335–364 (2006), DOI: 10.1007/s10618-005-0039-x

DOI: 10.14529/jsfi210103

Size & Shape Matters: The Need of HPC Benchmarks of High Resolution Image Training for Deep Learning

Ferran Parés Pont[1], *Pedro Megias*[1], *Dario Garcia-Gasulla*[1], *Marta Garcia-Gasulla*[1], *Eduard Ayguadé*[1,2], *Jesús Labarta*[1,2]

One of the purposes of HPC benchmarks is to identify limitations and bottlenecks in hardware. This functionality is particularly influential when assessing performance on emerging tasks, the nature and requirements of which may not yet be fully understood. In this setting, a proper benchmark can steer the design of next generation hardware by properly identifying said requirements, and quicken the deployment of novel solutions. With the increasing popularity of deep learning workloads, benchmarks for this family of tasks have been gaining popularity. Particularly for image based tasks, which rely on the most well established family of deep learning models: Convolutional Neural Networks. Significantly, most benchmarks for CNN use low-resolution and fixed-shape (LR&FS) images. While this sort of inputs have been very successful for certain purposes, they are insufficient for some domains of special interest (*e.g.*, medical image diagnosis or autonomous driving) where one requires higher resolutions and variable-shape (HR&VS) images to avoid loss of information and deformation. As of today, it is still unclear how does image resolution and shape variability affect the nature of the problem from a computational perspective. In this paper we assess the differences between training with LR&FS and HR&VS, as means to justify the importance of building benchmarks specific for the latter. Our results on three different HPC clusters show significant variations in time, resources and memory management, highlighting the differences between LR&FS and HR&VS image deep learning.

Keywords: deep learning, convolutional neural networks, high-resolution images, variable-shape images, HPC benchmarks.

Introduction

Nowadays, on the race for exascale, the leading architectures panorama is as heterogeneous as ever. Looking at the Top500 list (November 2020), we find five different architectures in the first six positions of the list. At the same time, the software is more heterogeneous than ever as almost all scientific fields use high-performance environments. This heterogeneity entails significant differences in terms of computing needs and patterns, software's maturity, and data demands. And motivates an individualized analysis for all applications of interest.

In this scenario, HPC benchmarks are of paramount importance. On the one hand, to set a common ground to evaluate different architectures and systems. On the other hand, to provide tools for users or developers to understand the most suited architecture for their specific needs.

With the recent rise of Artificial Intelligence (AI) applications, particularly through Deep Neural Network models (DNNs), benchmarks for this family of tasks have gained relevance. Meanwhile, the establishment of Convolutional Neural Networks (CNNs) as the de facto solution for most image related tasks has turned them into the spearhead of DNNs models for benchmarking purposes.

A common (but not always good) practice when training DNN models for images is to down-sample their size before computation. AI practitioners typically do this to reduce the memory requirements and training time of the model. Unfortunately, such down-sampling process entails deformation of visual patterns and generalizes information loss. While many applications are

[1]Barcelona Supercomputing Center (BSC), Barcelona, Spain
[2]Universitat Politecnica de Catalunya (UPC), Barcelona, Spain

not particularly affected by these drawbacks because they do not rely on details of the input (*e.g.*, telling cats from dogs), on other key domains where images are naturally of high-resolution and variable shape (HR&VS) down-sampling can entail a significant performance reductions. This includes strategic and critical applications like autonomous driving [8, 30], satellite data analysis [28], and medical diagnosis [12, 21].

The popularity of image-related tasks in AI, that can be solved successfully by down-sampling the data, has motivated the definition of several HPC benchmarks for low-resolution (LR) images. As the AI community looks for new challenges, the applications which benefit from HR&VS properties are gaining attention. This is highlighting the limitations of current systems and the need for HR&VS specific benchmarks, influencing the design of the next generation of hardware.

High-resolution (HR) data entails large memory requirements, which limits the amount of images that can be processed together (*i.e.*, in the same batch). Batch size limitations have a significant effect on the efficiency of the computation, while its impact on model performance remains under study by the AI community. At the same time, current accelerators (*e.g.*, GPUs, TPUs) are rather limited in terms of memory capacity, although workarounds to load larger memories than the one offered by the device have already been proposed (as discussed in Section 1). These workarounds include model parallelism [3, 10], activations re-computation [7] and offloading [6], enabling greater memory loads at the cost of computation efficiency. In this high-memory load context, avoiding accelerators and using CPU computation must be considered as a feasible alternative. The two main arguments in favour of CPUs being the lack of added components and functionalities for their execution, and the access to large memory devices, enabling larger, and therefore more efficient, batch sizes.

Variable shape is challenging to deal with, particularly in the context of batch training. All images in a batch need to have the exact same shape to be computed, and when your data is of VS, the easiest way to achieve a uniform shape without deformation or loss of information is through padding. Padding is a technique used to extend the size of an image by adding fixed-valued pixels (*e.g.*, zeros), and which can be used to fill the gaps between images of different shape found in the same batch. However, padding is something to be minimized for several reasons [29]. First, it introduces noise (*i.e.*, non-informative pixels) which can affect the learning process. Second, it increases the computational cost of the task, as padding pixels are also computed by the CNN. And third, padding increases the memory requirement of the task, as these values still need to be kept in memory. To complicate things even further, random batching in a VS problem can result in landscape and portrait images batched together. This entails huge amounts of padding, to the point where more padding than informative pixels may remain in the batch.

To motivate research towards hardware that can deal with HR&VS data, in this paper we evaluate the computational differences between a LR problem and a HR&VS problem. We use a public dataset composed by HR&VS images (MAMe, with an average of 6.6 megapixels per images), and explore the behavior of HPC clusters on different versions of it (low, mid and high resolution). Our results illustrate very distinct behaviors among clusters and problems, motivating the idea that better hardware for LR may not be better hardware for HR, and the other way around.

The remaining of the document is organized as follows. In Section 1 we review the related work, in Section 2 the details of the proposed Deep Learning Benchmark based on High Resolution images are explained. Section 3 describes the environment employed for the evaluation and the

complete performance comparison of the benchmark. Finally, we summarize the findings of the paper in the conclusions Section and the future work.

1. Related Work

After Linpack, the most popular benchmarks in the HPC community target specific architecture features: HPCG [11], NAS parallel benchmarks [4] or IO500 [24]. For the purpose of evaluating new architectures, these benchmarks are too specific [22], which calls for a bottom up approach. This means developing benchmarks, kernels and micro-applications that mimic some of the features or phases of scientific workloads: CORAL [2], Graph500 [25]. Even using real workloads to evaluate emerging systems [5, 9, 27].

In this context and with the growing popularity of AI applicaions, several HPC benchmarks for DNNs have been released. This includes AI500, [15], where high resolution images are used on weather predictions; HPL-AI, where mixed-precision operations are tested through solving a system of linear equations, [17, 18] or MLBench, on which different datasets from diverse fields can be processed, [20].

The most relevant for our work (due to the dimensionality of inputs) is MLPerf. In November 2020, this organization released an HPC benchmark for DNNs trainings based on two different tasks [1]. First, a 3D CNN (CosmoFlow [23]) trained with N-body cosmological simulation data to predict cosmological parameters. In this benchmark, data is composed by 128^3 voxels, which in standard 3-channel RGB form would correspond to images of 836x836x3, that is, 2.1 MP. Although the size of these data samples is larger than popular datasets, it still falls short of MAMe [26], with an average size of 6.6 MP. Furthermore, in CosmoFlow all samples have exactly the same size. That is, the data is of fixed size (FS).

The second task proposed in the MLPerf HPC benchmark is a 2D CNN model (DeepCam [19]). That is a convolutional encoder-decoder architecture based on ResNet-50, and trained to process climate related data and identify extreme weather phenomena. This weather data consists of a set of images all of fixed shape 768x1152x16. In standard 3-channel RGB form this would correspond to 2172x2172x3 images, that is, 4.7 MP. This is the same resolution used by another HR benchmark, AI500.

The aforementioned benchmarks are considered of high-resolution, and they are righfully so when compared to alternative tasks [26]. However, it is relatively easy to find data sources of relevance which already exceed these sizes (*e.g.*, mammographies [12]). Significantly, most benchmarking tasks (including both of MLPerf) have fixed size inputs, which simplifies memory management. Again, datasets of variable shape are easy to find in fields of relevance, including the majority of medical imaging sources [31].

The main difference between the MLPerf benchmarks and our work is both the purpose and scope. Regarding the former, MLPerf works with standard low-resolution and fixed-shape images (LR&FS) and its purpose is to overcome benchmarking variability inherent to DL systems due to hyperparameters, stochasticity, software and hardware differences. Instead, the main focus of our work is the use of novel HR&VS images that produce a training execution with different characteristics and, on the other side, we avoid focusing on benchmarking variablity by using partial deterministic training processes. HR&VS data produces a different training execution because this particular type of data requires to modify the training setting, hence producing an execution with different characteristics in comparison to regular LR&FS trainings. Regarding the scope, MLPerf benchmarks are executed on typical accelerators (GPUs & TPUs) to check the

efficiency on current state-of-the-art DL processes, while our work introduces a novel training execution that focus on benchmark hardware based on a different set of hardware characteristics, characteristics required to efficiently execute novel HR&VS trainings.

1.1. Memory Workarounds

In this work we consider a dataset of higher resolution based on MAMe. We use a ResNet18 architecture, which is smaller than the one used by DeepCam, and which requires less memory space for activations and gradients. Yet, in our HR&VS setting, roughly 3% of the training samples already require more than 16 GB of memory. This means the HR&VS task we evaluate in this paper does not fit in memory for accelerators with standard 16 GB memories, even when using batch size one. In these cases, some workarounds enable the use of larger memory loads at the cost of reducing the computation efficiency. Some of these alternatives being:

- **Model parallelism** [3, 10]: Split the model and assign each part to a set of accelerator devices. This technique splits the memory load between devices, effectively increasing the amount of memory available proportional to the number of accelerators. However, this comes at the cost of computation efficiency due to the need of multiple synchronization points and dependencies between such devices.
- **Re-computation** [7]: Re-computing the network activations every time the back-propagation process requires them, instead of keeping them in memory. This considerably reduces the memory from network activations at the cost of some repeated computation. For each step, instead of computing a single inference process, re-computation will compute up to n inference processes, where n is the number of layers in the network.
- **Offloading** [6]: Offloading network activations from accelerator to system memory. Whenever the back-propagation process requires a set of activations, they are transferred back from system to accelerator memory. This technique allows to alleviate memory requirements on the accelerator but at the cost of higher memory access times due to data movement between accelerator and system memory.

All these accelerator workarounds entail a reduction in computational efficiency, the scale of which has not been properly assessed so far. In this context, it is impossible to discard CPU computation as a competitive alternative, until proven otherwise. Particularly, since CPU computation has access to larger memory capacities (and thus, larger batch sizes), while avoiding the overhead of additional components and operations. If anything, CPU computation should be considered as the baseline for all workarounds based on accelerators.

2. High Resolution and Variable Shape (HR&VS) Benchmark

Training DL models with images of high-resolution and variable shape (HR&VS) is an active area of research within the AI field. Using HR&VS has desirable properties, such as keeping all the original information without any loss or deformation. Additionally, there is a set of problems that can only be computed under a HR regime because they require both attention to detail and understanding the overall structure.

In this work we analyze the computational differences between CNN training processes using low-resolution and fixed-shape images (LR&FS trainings) and HR&VS trainings. The goal is to assess the suitability of current hardware for both LR&FS trainings and HR&VS trainings, while highlighting the disparities between both. For such purpose, we use the MAMe dataset

(Museum Art Medium dataset [26]) because of the extreme HR&VS properties of its samples. MAMe contains 37,407 HR photographies of artworks hosted in three different museums. From this dataset, we produce three different input sets: low resolution (LR&FS), mid-resolution (MR&VS) and high resolution (HR&VS), as illustrated in Fig. 1. Notice the LR also implements a fixed shape for all data (FS), while the MR and HR enable variable shape inputs (VS) by adding padding when batching.

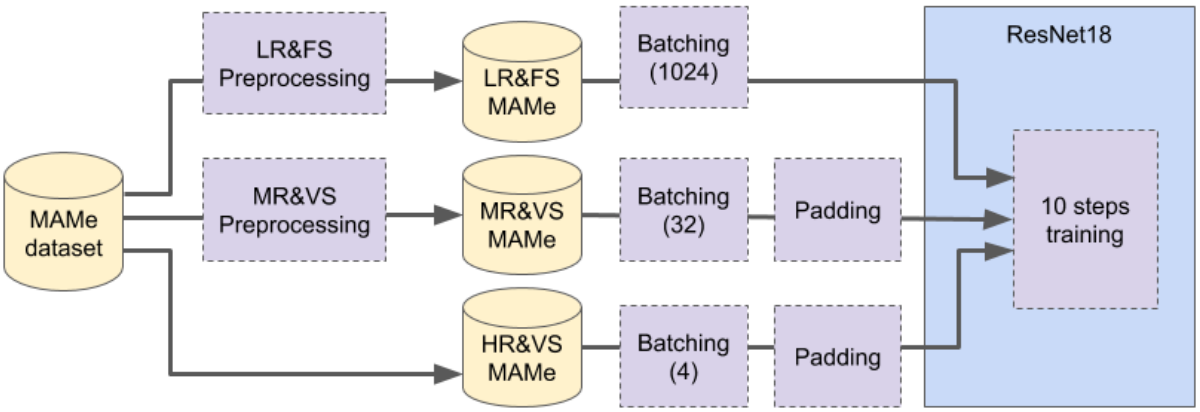


Figure 1. Diagram of the three tasks evaluated

The LR&FS pre-processing consists of down-sampling of all images to a fixed resolution of 256x256 pixels. For the MR&VS, the pre-processing consists of detecting which dimension (*i.e.*, width or height) is smaller, forcing it to 500 pixels long, while keeping the other dimension proportional *w.r.t.* the original aspect ratio of the image (*e.g.*, a 1000x2000 image becomes 500x1000). Finally, HR&VS requires no pre-processing, maintaining the original shape of images as in the MAMe dataset. Table 1 shows the megapixels of a batch for all three settings. Notice the LR has a bigger megapixel size than MR because of the bigger batch size. Also notice the significant variations in megapixels for the experiments that include VS (MR & HR), and how the corresponding padding is close to the amount of informative pixels.

Table 1. Megapixels of each experiment, per batch size for 10 randomly seeded batches. Informative corresponds to megapixels of original data, while padding corresponds to megapixels added after batching to obtain a homogeneous shape among batched samples

		Informative			Padding			Total		
Input	**Batch Size**	**Min.**	**Avg.**	**Max.**	**Min.**	**Avg.**	**Max.**	**Min.**	**Avg.**	**Max.**
LR&FS	1024	67.1	67.1	67.1	0	0	0	67.1	67.1	67.1
MR&VS	32	10.9	11.5	12.1	14.5	35.3	87.3	25.4	46.8	99.4
HR&VS	4	16.8	34.6	73.8	10.4	26.8	46.8	27.2	61.5	120.6

To obtain maximum comparability among all training processes, it is important that all of them use the same CNN architecture for the training process. The one which can process both *fixed shape* and *variable shape* data. In other words, the architecture has to be input-agnostic, capable of dealing with different input shapes (although channels dimension will always have a size of 3, Red-Green-Blue channels). We use a Resnet18 architecture [14] because of its popularity among AI practitioners, adding an extra adaptive pooling layer [13] right before the fully-connected layer, as this enables this particular architecture to be input-agnostic.

The batching policy is also shared among input sets, producing batches of samples randomly. While this produces homogeneous tensors for LR&VS, the variable shape nature of MR&VS and HR&VS requires the addition of padding. In these two scenarios, input processing pads the images in the same batch to fill the gaps between images, ensuring the same shape. Batch sizes also differ when training on each input set, to process a similar amount of pixels. By doing so, each executionl allocates nearly the same memory on each batch, ensuring that workloads are not heavily distant among cases. In detail, we use a batch size of 1024 for LR&FS, 32 for MR&VS and 4 for HR&VS corresponding to approximate average memory loads of 28 GB, 20 GB and 26 GB, respectively (see Tab. 2 for further details).

Table 2. Main memory allocation for each task in GB, per batch size for 10 randomly seeded batches

Input sets	Batch Size	Minimum	Average	Maximum
LR&FS	1024	27.9	28.3	28.655
MR&VS	32	11.0	19.8	41.518
HR&VS	4	12.2	25.7	38.003

The training process uses an Adam optimizer [16] with a learning rate of 0.0001, β_1 of 0.9, β_2 of 0.999 and no weight decay. For reproducibility purposes, random seeds are fixed to ensure that the process always join the same images in every batch. All the code necessary for reproducing our experiments is publicly available at our GitHub repository[3].

3. Experiments

In this section, we present the performance results when training the CNN ResNet18 with our proposed setup using a HR&VS input, and a comparison to training the same model using LR&FS and MR&VS. Our goal is to illustrate the computational particularities of a HR&VS setting, the practical relevance of which will continue to grow in the near future.

To that end we follow a top-down approach, analyzing first the elapsed time per megapixel for the training, and entering into detail in the later sections by looking at low level metrics such as: IPC, number of instructions executed per Megapixel, differentiating between counting all the pixels or just the informative ones (*i.e.*, not including the padding pixels), or last level cache misses per 1000 instructions.

3.1. Environment

We use three clusters for the performance evaluation: MareNostrum4, Minotauro and CTE-AMD. In Tab. 3 we can see the different characteristics of each cluster.

One of the main differences between these architectures and that is not highlighted in the previous table is the memory hierarchy. The Zen2 architecture, on which the CTE-AMD processor is based, forms groups of 4 cores, called CCX, and each one of those can directly access its own slice of 16 MB L3 cache. These CCX are paired inside CCDs, and our processor contains a total of 8 CCDs. This means that L3 cache in CTE-AMD is a 256 MB shared resource, but not all this cache is accessible by all cores.

[3]https://github.com/HPAI-BSC/SizeMatters

Table 3. Hardware configuration of the nodes used in the experiments

	MareNostrum4	Minotauro	CTE-AMD
CPU name	Intel Xeon Platinum 8160	Intel Xeon E5-2630 v3	AMD EPYC ROME 7742
Sockets	2	2	1
Cores/socket	24	8	64
Threads/core	1	1	2
Frequency	2.1 GHz	2.4 GHz	2.25 GHz
L1 Cache	private 32 KiB	private 32 KiB	private 32 KiB
L2 Cache	private 1024 KiB	private 256 KiB	private 512 KiB
L3 Cache	shared 33 MiB	shared 20 MiB	shared 256 MiB
Memory/node	96 GiB	128 GiB	1024 GiB
Memory tech	DDR4-2666	DDR4-2133	DDR4-3200

To obtain further insights, we compare the experiments at three levels: using four cores, using one socket and using a full node. Only one node of each cluster is used to avoid the effect of network in the results and focus on architectural differences. We also collect hardware counters using a linux embedded tool, Perf, which allows us to record the events listed in the following table.

Table 4. Hardware counters obtained per cluster

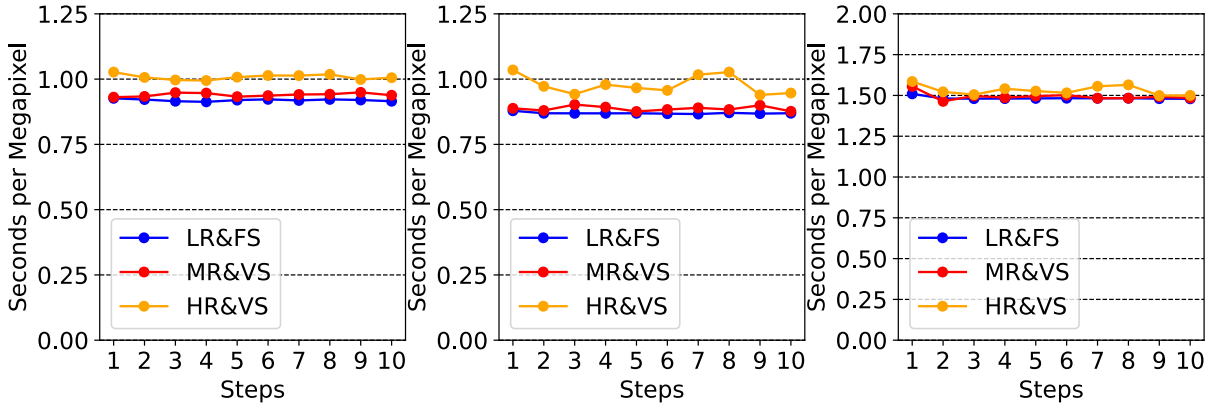
Marenostrum4 and Minotauro	CTE AMD
CPU-CYCLES	CPU-CYCLES
INSTRUCTIONS	INSTRUCTIONS
MEM_LOAD_RETIRED.L3_MISS	l3_misses

We use Singularity (version 3.6.4 on CTE-AMD and 3.5.2 on Marenostrum4 and Minotauro) to containerize the experiments and ensure reproducibility. The same container is used in the three clusters, it includes pytorch 1.6.0, torchvision 0.7.0, numpy 1.17.4, pandas 0.25.3, pillow 6.2.1, python-dateutil 2.8.1, pytz 2019.3 and six 1.13.0.

As said before, the three data-sets explained in Section 2 are used to train the ResNet18 model. In order to homogenize the memory consumption of the different input sets we use a different batch size for each data-set. For LR&FS batch size is 1024, for MR&VS batch size is 32 and for HR&VS the batch size is 4. This will allocate roughly the same amount of memory for all the data-sets. In our results we show the average of three independent runs performing 10 training steps. We have verified that the variability between different runs is below 6%.

3.2. Execution Time

In this subsection we show the differences in timing when processing the different input sets in each cluster. In the context of random batching, LR&FS task produces batches with a constant shape because all images have same dimensions. However, in those cases where images have variable shape (MR&VS and HR&VS), the amount of pixels to compute in a batch may

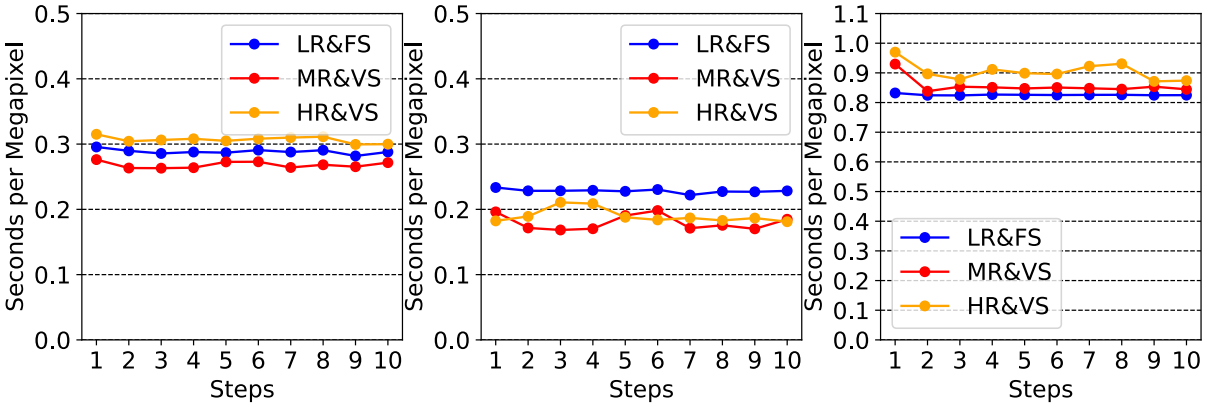


(a) MareNostrum4 (b) CTE-AMD (c) Minotauro

Figure 2. Execution time per megapixel using 4 cores

vary significantly from step to step hence. For this reason we use the "Seconds per Megapixel" metric as a common ground.

In Fig. 2 we can see performance obtained on the different clusters when using 4 cores to process the different data sets. We observe not only that there is a difference of performance achieved by the different clusters, but also that the performance between the different data sets is different. In Minotauro the performance variation between the different data sets is minimum, while in CTE-AMD, the HR&VS data set performs worse than the others, presenting also a high variability between the different steps. In Marenostrum the HR&VS also shows a worse performance but without the variability seen in CTE-AMD.

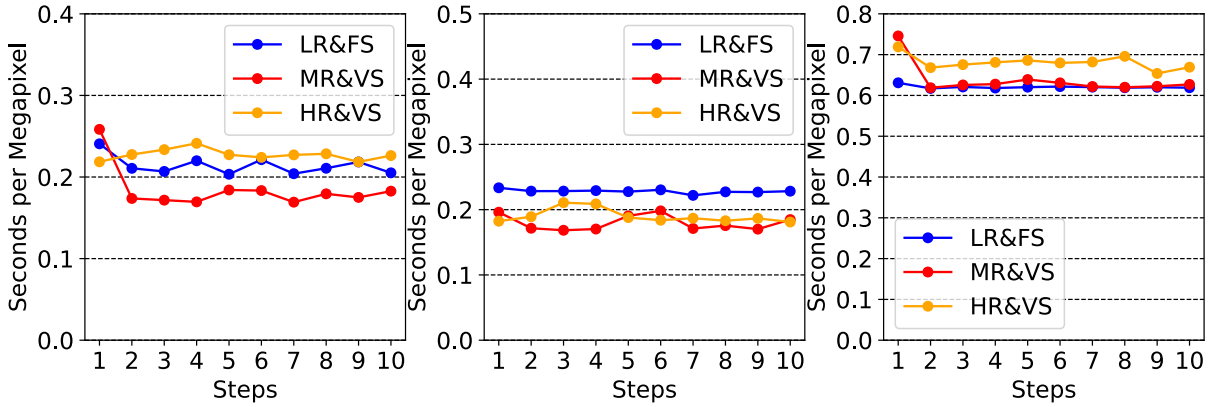


(a) MareNostrum4 (b) CTE-AMD (c) Minotauro

Figure 3. Execution time per megapixel using 1 socket

In Fig. 3 we show the seconds per megapixel achieved when using one full socket in each cluster. In MareNostrum4 and Minotauro the HR&VS is the data set with worse performance but not in CTE-AMD. The LR&FS is the best performing one in Minotauro and the worse one in CTE-AMD. In CTE-AMD both HR&VS and MR&VS present a high variability and it is not clear which one performs better. This illustrates the differences in computational behaviour caused by a change in the input resolution.

The performance obtained when using a full node of each cluster can be seen in Fig. 4. We observe that, in MareNostrum4 the MR&VS performs clearly worse than the other data sets,



(a) MareNostrum4 (b) CTE-AMD (c) Minotauro

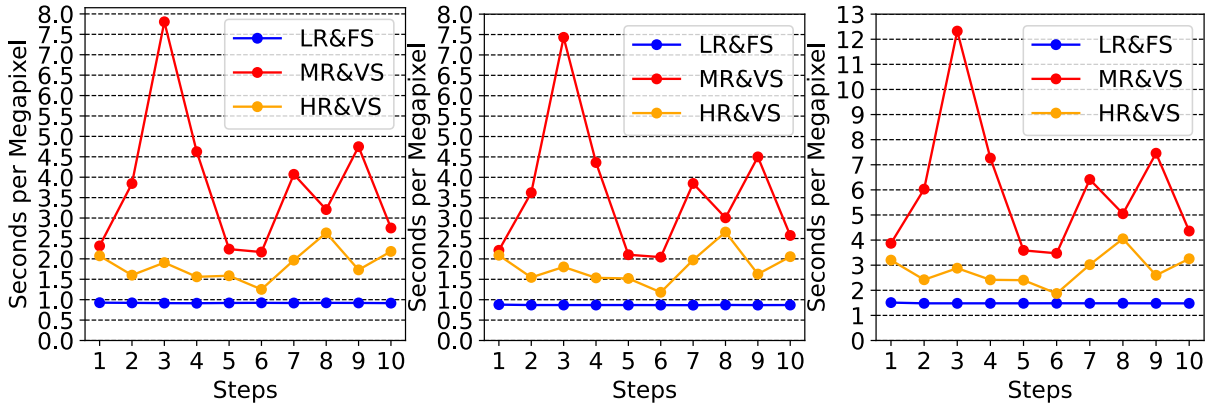
Figure 4. Execution time per mega pixel using 1 node

and also that the HR&VS is slightly better than the LR&FS data set. In Minotauro, on the other hand, the worse performing data set is the one using HR&VS images, without relevant difference between using LR&FS or MR&VS.

With this first analysis, we have demonstrated that the HR&VS data set presents a different performance from the LR&FS when running on different architectures. This alone illustrates the particular nature of HR&VS when compared to other DNN tasks, and justifies specific benchmarks for it. To understand where the differences come from we expand the analysis in the following sections with additional metrics.

3.3. Padding Effect

One of the main differences between the different data sets is the padding, added to the HR&VS and MR&VS data sets, in order to keep a homogeneous shape within each batch. We refer to these pixels as non-informative, as they do not carry any information but they are computed by the network nevertheless. In this section, we study the performance of the different experiments in each cluster taking into account only the informative pixels.

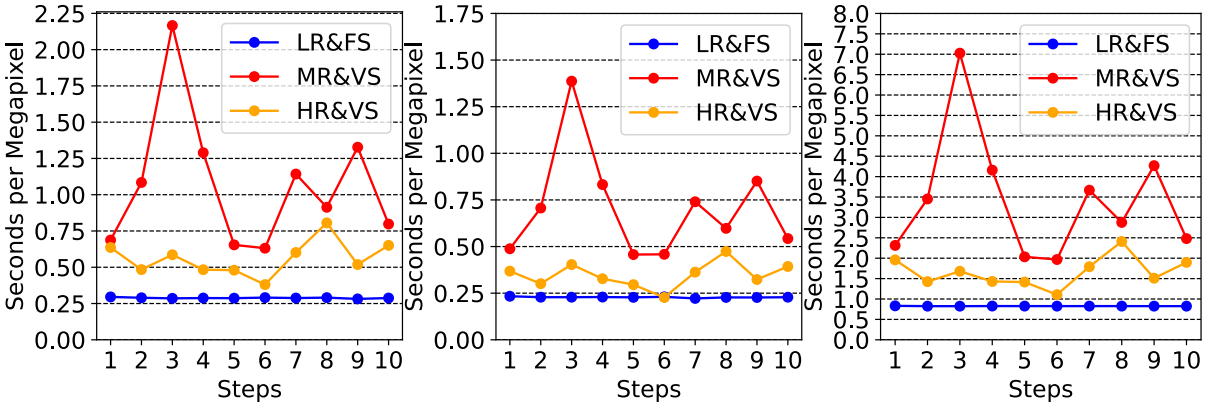


(a) MareNostrum4 (b) CTE-AMD (c) Minotauro

Figure 5. Execution time per informative megapixel using 4 cores

In Fig. 5 we can see the seconds per informative Megapixel achieved by the different data sets when running in 4 cores of each cluster. We observe that, although, the performance obtained

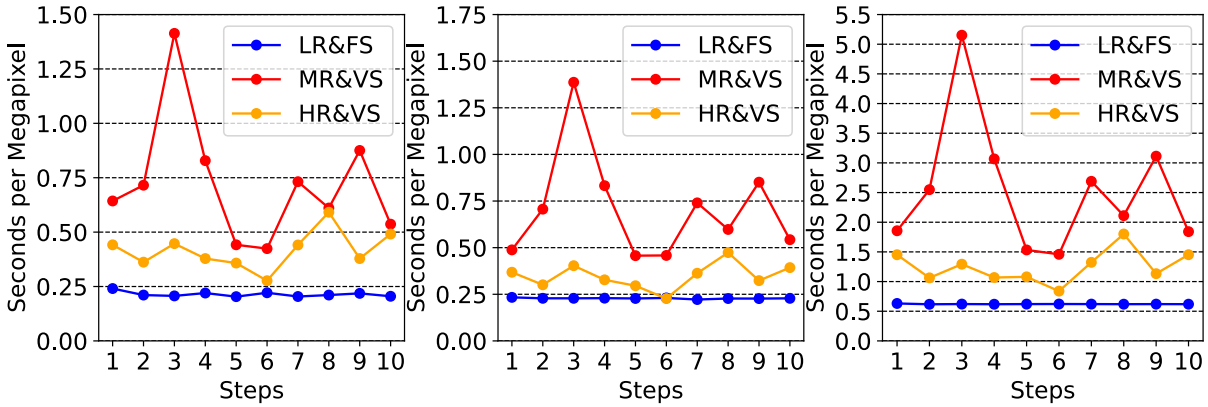
in each cluster is different, in all the clusters the best performing input set is the the LR&FS. Significantly, the LR&FS data set does not have padding pixels therefore, all the computation is done on informative pixels. This indicates that padding is adding a significant overhead to the computation. The worse performing data set is the MR&VS with a high variability between different time steps, this is explained because the MR&VS data set has a higher batch size than the HR&VS which means a higher variability in image shapes within the same batch and therefore, more padding pixels.



(a) MareNostrum4 (b) CTE-AMD (c) Minotauro

Figure 6. Execution time per informative megapixel using 1 socket

The performance results when using one socket of each cluster can be seen in Fig. 6. In this case the results look similar to the ones obtained when using 4 cores, the order and shape of the lines are the same. The most relevant difference between the clusters is the difference in performance obtained between MareNostrum4 and CTE-AMD, compared when using 4 cores. While using 4 cores the performance of both clusters is very similar, when using a full socket CTE-AMD clearly outperforms MareNostrum4. This can be explained by the difference in the number of cores between the two clusters, 24 for MareNostrum4 and 64 for CTE-AMD.



(a) MareNostrum4 (b) CTE-AMD (c) Minotauro

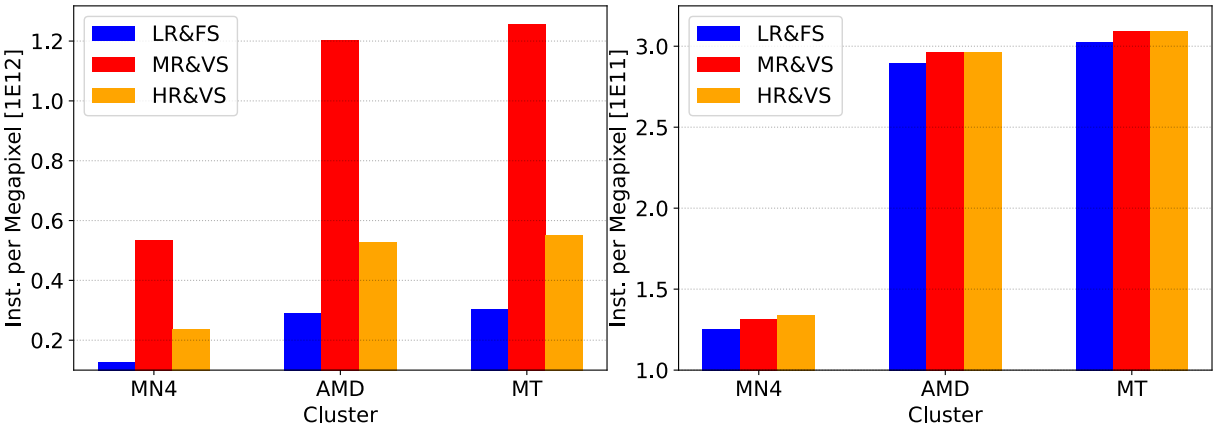
Figure 7. Execution time per informative megapixel using 1 node

In Fig. 7 we show the performance obtained taking into account the informative pixels when using the full node. While the performance of the different data sets remains the same for the different clusters, we can observe that one node of Marenostrum4 achieves the same performance

as the CTE-AMD with less cores. With these results we can conclude that the padding pixels (which have always zero value) add overhead to the training, but less than informative pixels.

3.4. Executed Instructions

In this section, we study the amount of instructions required to train the model with the different inputs using the two metrics: with and without non-informative pixels. Note that in this case the number of instructions executed are obtained from hardware counters at the end of the execution, therefore, metrics are not detailed per time step.



(a) Informative pixels

(b) All pixels

Figure 8. Number of instructions per MP in 4 cores

In Fig. 8 we can see the Instructions per Megapixel executed in each experiment and cluster. On the right hand the plot depicts the Instructions per Megapixel taking into account all the pixels processed (including padding), on the left hand side we can see the same metric but taking into account only the informative pixels.

The most straightforward conclusion from this experiment is that the number of instructions necessary to process a pixel is almost the same if the pixel is informative or not. This can be seen in Fig. 8b, where LR&FS (which holds no padding) executes almost the same number of instructions as the MR&VS or HR&VS per pixel. From Fig. 8a we can also see that there are important differences in the number of instructions executed per informative Megapixels in the different input sets. The pattern of this difference (higher for MR&VS, lower for LR&FS) confirms that the difference comes from the padding pixels that are being executed but not accounted.

We can observe the same pattern across the different architectures: MR&VS always presents a higher number of instructions per MP than the other inputs and LR&FS the lowest one. This means that the kind and number of instructions executed do not depend on the resolution of the image or the batch size used.

Finally the last observation is the difference in the number of instructions executed per Megapixel between the different clusters. While Minotauro and CTE-AMD show a similar number of instructions executed, Marenostrum4 roughly uses half the number of instructions that the other two clusters. In order to explain the lesser number of instructions executed by MareNostrum4 cluster, we have to point at its CPU capabilities. In particular, we can see that the Intel Xeon Platinum 8160 is capable of executing AVX512 instructions, while CTE-AMD and Minotauro can only execute AVX2 and the vector length of AVX512 is twice the size of AVX2 vector.

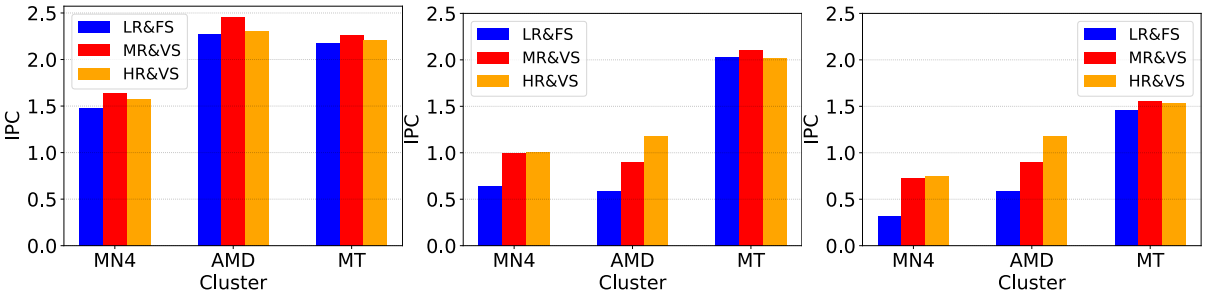
This observation also indicates that all input sets are making an intensive use of the vector units of the different processors.

We do not show the corresponding plots for the execution on one socket and one node because they show the same pattern and there is no difference in the number of instructions executed.

After these results, we can conclude that the number of instructions depends on the total amount of pixels, and non-informative pixels require the same number of instructions as the informative ones. Also, there is no difference in the cost in terms of instructions when processing LR, MR or HR pixels.

3.5. Instructions per Cycle (IPC)

In the previous sections we have seen that there is a difference in the execution time between the different input sets and that this difference does not come from the total number of instructions executed. Here we analyze the IPC of each cluster when facing each of the input sets for the different configurations and clusters. In Fig. 9 we can see the average IPC obtained by each input set in each cluster when using four cores, one socket or one node.



(a) 4 cores (b) 1 socket (c) 1 node

Figure 9. Instructions per cycle (IPC)

Looking at Fig. 9a we observe that all the clusters share the same pattern when training the network with the different image input sets. The MR&VS input set achieves the highest IPC in all the cases and LR&FS is the lowest one. Knowing that the MR&VS input set is the one that includes more padding pixels and that the LR&FS does not include padding pixels, we can assume that, although the informative and non-informative pixels need the same number of instructions, the instructions used for non-informative pixels are faster to execute (*i.e.*, use less cycles to complete).

We also observe a notable difference between the IPC achieved by the different clusters where Marenostrum4 shows a lower IPC than the other clusters. In the previous section we have seen that Marenostrum4 executes less instructions for the same input than the other clusters but these instructions take more cycles than the ones executed by CTE-AMD and Minotauro.

In Section 3.2 Fig. 2a Marenostrum4 and CTE-AMD has showed a similar performance in terms of execution time per Megapixel. It is interesting to see that this is achieved by both architectures using different vector units but delivering the same performance. In Marenostrum4 executing less instructions at a lower IPC and in CTE-AMD executing more instructions at a higher IPC.

Looking at Fig. 9b we find the IPC when using a whole socket of each cluster. We can notice important variations on shapes. In this case Minotauro gets the best IPC having almost

no difference with the one obtained when using 4 cores, this is due to the fact that Minotauro is the one with the fewer cores per socket (8, versus 24 in Marenostrum4 and 64 in CTE-AMD) with 8 cores the shared resources of the socket are not being saturated.

On the other hand, CTE-AMD shows an important drop in the IPC when using one socket (64 cores) with respect to using 4 cores. This could indicate a saturation on the memory bandwidth but we will verify this assumption in the following section looking at memory access hardware counters. Marenostrum4 also shows a drop in the IPC when using a full socket (24 cores) but not as drastic as the one on CTE-AMD.

Looking at the different input sets we also see relevant differences. The LR&FS input set has a lower IPC when running in one socket than when running in 4 cores. We know that it is not related to the padding pixels as MR&VS and HR&VS do not show the same trend (MR&VS having more padding pixels than HR&VS). We could relate it to the batch size, this means that although the amount of memory accessed is roughly the same as shown in Section 2 Tab. 2 there is a difference depending if it is organized in more images or less.

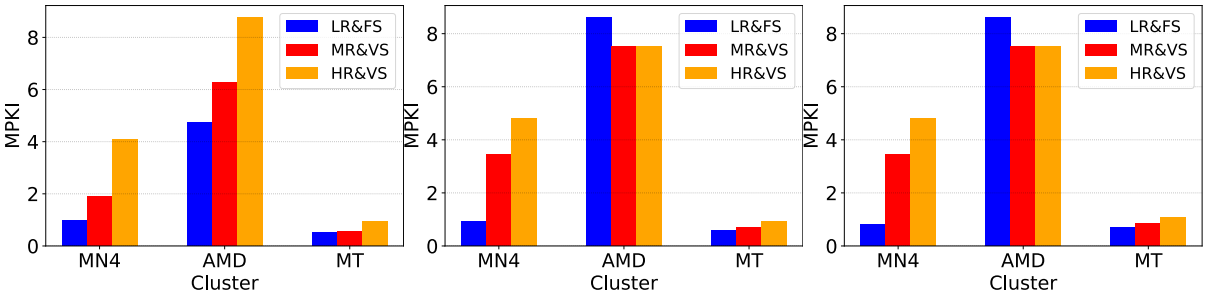
In Fig. 9c we can see the IPC achieved by each cluster when using a full node. Both Minotauro and Marenostrum 4 show a lower IPC than when using one socket, meaning that using more cores some of the shared resources of the node are being saturated. It is also interesting to notice the different behavior of the different input sets. In Minotauro there seems to be not a very important difference in the IPC obtained by the different inputs. Therefore, there is a difference in the execution time shown in Fig. 4c that is not explained by the IPC nor by the number of instructions executed by the different input sizes. At this point this difference can only come from a difference in the frequency (which is fixed for the HPC systems being evaluated but could come from low cycles per microsecond due to I/O operations or OS preemption) or the useful execution of instructions, meaning that with the current approach we account for all the instructions executed, but there could be phases with busy waiting processes or threads that are not performing useful work. A further detailed analysis is needed to unveil this difference.

Between Marenostrum4 and CTE-AMD there is also an important difference, in Marenostrum4 MR&VS achieves the same IPC as HR&VS while in CTE-AMD the IPC obtained when training with HR&VS is higher than when using MR&VS. We will try to understand this differences in the following section looking at memory HWC in detail.

3.6. Memory Access

In this section, we analyze the data access when using the different input sets for the training of the network. In Fig. 10 we can see the misses of the last level cache (LLC), that in all the clusters correspond to L3, per 1000 instructions (MPKI). We assume every miss on L3 implies a data transfer from memory, therefore, we use this metric as a measure of the pressure on the memory system.

Looking at Fig. 10a, that corresponds to the execution using 4 cores, we see a common behaviour in all the clusters: the HR&VS input set has a higher MPKI than the other input sets, and LR&FS has the lower MPKI. The CTE-AMD cluster shows a much higher MPKI for all the input sets than the other clusters, this can be explained by its architecture, as the 4 cores that are being used belong to the same CCX and share a 16 MB L3. This is quite small compared to the 33 MB available in the L3 of Marenostrum4.



(a) 4 cores (b) 1 socket (c) 1 node

Figure 10. LLC Misses per 1000 instructions (MPKI)

We have to take into account that in CTE-AMD when a core misses an access to its L3 cache slice, the data can come from another CCX cache or from main memory and with the given hardware counters we cannot differentiate this two cases.

We can see the MPKI for the execution using one socket in Fig. 10b. In this case we observe an important change in the behaviour of the CTE-AMD. The LR&VS set is the one showing a higher MPKI, almost twice the one observed when using 4 cores. MR&VS is also higher but HR&VS shows a lower MPKI than when using 4 cores. The increase in MPKI for the LR&FS and MR&VS can be explained because the L3 is shared and the different processes are removing each others data from L3. And the decrease of HR&VS is an effect of having more capacity of L3 available, as now using the whole socket it has 256 MB of L3 available.

It is clear that the different input sets present a different behaviour in terms of memory accesses. Probably the LR&FS can reuse less data because of the higher batch size, while HR&VS can reuse more data from the caches.

Conclusions

Motivated by the need of processing high resolution and variable shape images, we have uncovered relevant performance differences and needs. These are visible between the different input sets when running in clusters with different system configurations and CPU architectures.

The use of padding pixels, needed to create a batch of variable shape images, has a significant impact in the performance. As shown in Figs. 8 and 9, padding pixels require the same amount of instructions as informative pixels but they can be processed faster than informative ones.

We have not been able to explain the differences in performance between the different input sets when running in Minotauro using the aggregated hardware counters. A more detailed analysis based on tracing is necessary.

Clearly, the use of vector units such as AVX512 is beneficial for this kind of workloads, but it is interesting to notice also that smaller vector units can deliver the same performance if they can run at a higher IPC.

We have also demonstrated that the memory access pattern is different between regular LR&FS and novel HR&VS sets, even though all the input sets used roughly the same amount of memory the MPKI measured are different. This means that the configuration of the batch has an impact in the memory access pattern, by batch configuration we mean number of images, size of the images and amount of padding pixels.

Future Work

The work presented here analyzes a problem (training CNNs with HR&VS data) of relevance for the next generation of AI services (*e.g.*, medical diagnosis, autonomous driving), which is at the limit of what can be computed efficiently by current HPC infrastructure (due to memory requirements). For assessing the relevance of this work, the definition of future work is of paramount importance.

Following the problem introduction of this paper, we foresee two main milestones in the future. First, gaining further insights into the problem at hand, since the analysis here presented is of limited depth. And second, implementing and releasing a closed benchmark to facilitate adoption by the community.

The analysis of this work is of limited scope because we have only access to hardware counters and a limited list of them. The next step in the analysis will be to do a detailed performance analysis using tracing tools to understand the parallel and computational behaviour of the different executions.

Finally, although all codes and environments needed to reproduce our results are publicly available, further work is needed towards transforming the work of this paper into an HPC benchmark. The best way to do so is to integrate our work with a well established benchmarking organization, like MLPerf. At that point, we expect researchers all over the world to fully tackle the proposed problem, eventually solving it through the next generation of HPC hardware and software.

References

1. Press Release 11/18/20: MLPerf Releases Inaugural Results for Leading High-Performance ML Training Systems (2020), `https://mlperf.org/press#mlperf-hpc-v0.7-results`

2. Advanced Simulation and Computing: CORAL Benchmarks. `https://asc.llnl.gov/coral-benchmarks`, accessed: 2021-04-06

3. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. CoRR abs/1409.0473 (2014), `https://arxiv.org/abs/1409.0473`

4. Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. Tech. rep., Technical Report NAS-95-020, NASA Ames Research Center (1995)

5. Banchelli, F., Garcia-Gasulla, M., Houzeaux, G., Mantovani, F.: Benchmarking of state-of-the-art HPC Clusters with a Production CFD Code. In: Proceedings of the Platform for Advanced Scientific Computing Conference, 29 June-1 July 2020, Geneva, Switzerland. pp. 1–11 (2020), DOI: 10.1145/3394277.3401847

6. Beaumont, O., Eyraud-Dubois, L., Shilova, A.: Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training, pp. 151–166 (2020), DOI: 10.1007/978-3-030-57675-2_10

7. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost. CoRR abs/1604.06174 (2016), https://arxiv.org/abs/1604.06174

8. Chen, X., Ma, H., Wan, J., Li, B., Xia, T.: Multi-view 3d object detection network for autonomous driving. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 21-26 July 2017, Honolulu, HI, USA. pp. 1907–1915. IEEE (2017), DOI: 10.1109/cvpr.2017.691

9. Criado, J., Garcia-Gasulla, M., Kumbhar, P., Awile, O., Magkanaris, I., Mantovani, F.: CoreNEURON: Performance and Energy Efficiency Evaluation on Intel and Arm CPUs. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER), 14-17 Sept. 2020, Kobe, Japan. pp. 540–548. IEEE (2020), DOI: 10.1109/cluster49012.2020.00077

10. Dean, J., Corrado, G.S., Monga, R., et al.: Large scale distributed deep networks. In: Advances in Neural Information Processing Systems. vol. 25. Curran Associates, Inc. (2012)

11. Dongarra, J., Heroux, M.A., Luszczek, P.: HPCG benchmark: a new metric for ranking high performance computing systems. The International Journal of High Performance Computing Applications 30(1), 3–10 (2016), DOI: 10.1177/1094342015593158

12. Geras, K.J., Wolfson, S., Shen, Y., et al.: High-resolution breast cancer screening with multi-view deep convolutional neural networks. CoRR abs/1703.07047 (2017), https://arxiv.org/abs/1703.07047

13. He, K., Zhang, X., Ren, S., Sun, J.: Spatial pyramid pooling in deep convolutional networks for visual recognition. IEEE transactions on pattern analysis and machine intelligence 37(9), 1904–1916 (2015), DOI: 10.1007/978-3-319-10578-9_23

14. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, 27-30 June 2016, Las Vegas, NV, USA. pp. 770–778. IEEE (2016), DOI: 10.1109/cvpr.2016.90

15. Jiang, Z., Gao, W., Wang, L., et al.: HPC AI500: a benchmark suite for HPC AI systems. In: International Symposium on Benchmarking, Measuring and Optimization, 10-13 Dec. 2018, Seattle, WA, USA. pp. 10–22. Springer (2018), DOI: 10.1007/978-3-030-32813-9_2

16. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. CoRR abs/1412.6980 (2014), https://arxiv.org/abs/1412.6980

17. Kudo, S., Nitadori, K., Ina, T., Imamura, T.: Implementation and Numerical Techniques for One EFlop/s HPL-AI Benchmark on Fugaku. In: Proceedings of the 11th IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale, 13 Nov. 2020, GA, USA. pp. 69–76. IEEE (2020), DOI: 10.1109/ScalA51936.2020.00014

18. Kudo, S., Nitadori, K., Ina, T., Imamura, T.: Prompt report on Exa-scale HPL-AI benchmark. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER), 14-17 Sept. 2020, Kobe, Japan. pp. 418–419. IEEE (2020), DOI: 10.1109/cluster49012.2020.00058

19. Kurth, T., Treichler, S., Romero, J., et al.: Exascale deep learning for climate analytics. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, 11-16 Nov. 2018, Dallas, TX, USA. pp. 649–660. IEEE (2018), DOI: 10.1109/sc.2018.00054

20. Liu, Y., Zhang, H., Zeng, L., Wu, W., Zhang, C.: MLbench: benchmarking machine learning services against human experts. Proceedings of the VLDB Endowment 11(10), 1220–1232 (2018), DOI: 10.14778/3231751.3231770

21. Lotter, W., Sorensen, G., Cox, D.: A multi-scale CNN and curriculum learning strategy for mammogram classification. In: Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support, 17 Sept. 2017, Québec City, QC, Canada, pp. 169–177. Springer (2017), DOI: 10.1007/978-3-319-67558-9_20

22. Mantovani, F., Garcia-Gasulla, M., Gracia, J., et al.: Performance and energy consumption of HPC workloads on a cluster based on Arm ThunderX2 CPU. Future Generation Computer Systems 112, 800–818 (2020), DOI: 10.1016/j.future.2020.06.033

23. Mathuriya, A., Bard, D., Mendygral, P., et al.: CosmoFlow: Using deep learning to learn the universe at scale. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, 11-16 Nov. 2018, Dallas, TX, USA. pp. 819–829. IEEE (2018), DOI: 10.1109/sc.2018.00068

24. Monnier, N., Lofstead, J., Lawson, M., Curry, M.: Profiling platform storage using IO500 and mistral. In: 2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW), 18 Nov. 2019, Denver, CO, USA. pp. 60–73. IEEE (2019), DOI: 10.1109/pdsw49588.2019.00011

25. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the Graph 500. Cray Users Group (CUG) 19, 45–74 (2010)

26. Parés, F., Arias-Duart, A., Garcia-Gasulla, D., et al.: A Closer Look at Art Mediums: The MAMe Image Classification Dataset. CoRR abs/2007.13693 (2020), `https://arxiv.org/abs/2007.13693`

27. Ramirez-Gargallo, G., Garcia-Gasulla, M., Mantovani, F.: TensorFlow on state-of-the-art HPC clusters: a machine learning use case. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 14-17 May 2019, Larnaca, Cyprus. pp. 1–8. IEEE (2019), DOI: 10.1109/ccgrid.2019.00067

28. Reichstein, M., Camps-Valls, G., Stevens, B., et al.: Deep learning and process understanding for data-driven earth system science. Nature 566(7743), 195–204 (2019), DOI: 10.1038/s41586-019-0912-1

29. Sotiropoulos, I.N.: Handling variable shaped & high resolution images for multi-class classification problem. Master's thesis, Universitat Politècnica de Catalunya (2020)

30. Treml, M., Arjona-Medina, J., Unterthiner, T., et al.: Speeding up semantic segmentation for autonomous driving. In: MLITS, NIPS Workshop. vol. 2, p. 7 (2016)

31. Wu, N., Phang, J., Park, J., et al.: Deep neural networks improve radiologists' performance in breast cancer screening. IEEE transactions on medical imaging 39(4), 1184–1194 (2019), DOI: 10.1109/TMI.2019.2945514

DOI: 10.14529/jsfi210104

Computational Resource Consumption in Convolutional Neural Network Training – A Focus on Memory

Luis A. Torres[1,2,3], *Carlos J. Barrios*[1,2,3], *Yves Denneulin*[4,5,6,7,8]

Deep neural networks (DNNs) have grown in popularity in recent years thanks to the increase in computing power and the size and relevance of data sets. This has made it possible to build more complex models and include more areas of research and application. At the same time, the amount of data generated during the training process of these models puts great pressure on the capacity and bandwidth of the memory subsystem and, as a direct consequence, has become one of the biggest bottlenecks for the scalability of neural networks. Therefore, the optimizing of the workloads produced by DNNs in the memory subsystem requires a detailed understanding of access to the memory and the interactions between the processor, accelerator devices, and the system memory hierarchy. However, contrary to what would be expected, most DNN profilers work at a high level, so they only perform an analysis of the model and individual layers of the network leaving aside the complex interactions between all the hardware components involved in the training. This article shows the characterization performed using a convolutional neural network implemented in the two most popular frameworks: TensorFlow and Pytorch. Likewise, the behavior of the component interactions is discussed by varying the batch size for two sets of synthetic data and showing the results obtained by the profiler created for the study. Moreover, the results obtained when evaluating the AlexNet version on TensorFlow and its similarity in behavior when using a basic CNN are included.

Keywords: High-Performance Computing, deep learning, profiling, performance characterization, memory consumption.

Introduction

In recent years, Deep Learning (DL) algorithms have become popular due to their ability to extract features from input data. They are based on artificial neural networks and contain a lot of hidden layers, which is why they are known as Deep Neural Networks [18]. This type of network can contain millions of internal parameters resulting from multiple nonlinear and iterative transformations that occur in matrix or tensory form.

Deep neural networks have been divided into several groups including Convolutional Neural Networks, Recurrent Neural Networks (RNNs), and Long Short Term Memory (LSTM) [25]. These types of structures are used to create specific models according to a goal and require a great computing capacity and significant time (depending on the complexity of the model and computational resources) in their training to obtain a suitable model. It should be noted that the vast majority of this training time is used in Matrix-by-Vector Multiplication (MVM) tasks produced by convolutional or classification layers (dense layers). As mentioned earlier, such structures contain a large number of parameters and that, together with MVM operations performed on hidden layers, lead to a high transfer rate between memory, processor, and accelerator devices, requiring higher processing and memory capabilities. In this way, memory has

[1]Universidad Industrial de Santander, Bucaramanga, Colombia
[2]Supercomputación y Cálculo Científico UIS (SC3UIS), Bucaramanga, Colombia
[3]Cómputo Avanzado y a Gran Escala (CAGE), Bucaramanga, Colombia
[4]Université Grenoble Alpes, Grenoble, France
[5]French National Centre for Scientific Research (CNRS), Grenoble, France
[6]Inria, Grenoble, France
[7]Grenoble INP Institute of Engineering Univ. Grenoble Alpes, Grenoble, France
[8]Laboratoire d'Informatique de Grenoble (LIG), Grenoble, France

become a limit to the model that can be studied and the amount of data used for your training. Finally, it should be noted that MVM operations are highly parallelisable and therefore so are the Deep Learning algorithms.

Most studies using memory profilers are based on high-level understanding of the individual DNN layers or analytical models such as the Roofline [30] (roofline analysis helps to visualize the limits imposed by the hardware, as well as to determine the main limiting factor - memory bandwidth or computational capacity - thus leading to an ideal roadmap of possible optimization steps [29]). These approaches do not capture the complex interaction between the CPU, memory, and accelerator devices. As such, it only provides high-level memory performance sources in a static CPU / Memory / Device configuration.

The purpose of this work is to present the analysis of the characterization of the resources consumed in the training of a convolutional neural network implemented in both Tensorflow and Pytorch in order to determine the behavior of the workloads and identify possible causes of the bottlenecks in the training phase. The characterization has been performed using two synthetic datasets of sixty thousand and six hundred thousand 32x32x3 tensioners and using values of 32, 64, 128, 256 and 512 for the batch size.

In Section 1, we present a summary of related works that address solutions for memory problems in neural network training and where we start from as motivation for the realization of this work. Section 2 describes the methodology used to carry out the study, detailing the resources implemented in it. In Section 3, the reason for the memory problem presented during training is described. In Section 4 the results are shown and an evaluation of them is made according to the behavior obtained by the profiler. The last section concludes this paper.

1. Related Works

The requirements of computing capabilities for performing deep neural network training have increased significantly in recent years. Similarly, each new model that emerges for a specific field requires greater precision; much more complex and sophisticated models, with a larger number of layers and neurons, increase the number of trainable parameters of the network [9, 12]. These models dynamically generate tens or hundreds of MegaBytes of intermediate data (activations or feature maps for convolutional layers) for each layer of the network, data that often exceeds the capacity of the first levels of the memory hierarchy and puts enormous pressure on the bandwidth of the main memory [5].

All of the above have forced accelerator designers for deep neural networks to use high-cost memory solutions such as HBM (High Bandwidth Memory) used in Google TPUs [11]. Other solutions have been proposed to overcome these limitations such as the development of new techniques to improve training by working directly on the neural network graph [3, 15] or working with sparse matrices [20]. Designing specialized dense nodes for the effective use of accelerators has also been done. These nodes include: NVIDIA Tesla A100, Google TPU, or Intel GAUDI. On such nodes, training efficiency depends on model parallelization (HoroVod [24] and KARMA [28]) and effective communications between accelerators performed by a specialized network such as NVIDIA NVLink [16].

Now, despite these innovative designs, the use of increasingly deep and dense network topologies has made the resources available for training still a problem, particularly memory capacity. In light of this, techniques such as using throttle memory as an application-level cache relative to host memory have emerged during the training process [21]. This technique is very sensitive

to communication bandwidth and can incur latency due to communication and data synchronization. The use of disaggregated memory has also been considered [17], but it presents the same bottleneck as the previous one: the PCIe bus.

Other solutions include ASICs (Application-specific integrated circuit) which can be up to three times faster than general-purpose devices [8] and specialized accelerators such as DaDianNao [4] where a nearby data processing approach is adopted and has a neural functional unit that performs arithmetic operations and receives the values of the network weights from adjacent eDRAM (Embedded DRAM) memory. Finally, other types of architectures have been proposed which are memory-centric and where memory modules are added within the accelerators. This type of architecture discusses memory modules decoupled from the PCIe and parked locally within the interconnection of devices, using NVlink for example. This maximizes the communication bandwidth between them while expanding the total memory capacity of the system [14].

It should be noted that all these improvements in the hardware and the training methods of deep neural networks aim both at increasing the performance of the model and reducing the training times. Most of the research on the hardware used for training has been focused on scaling the computational capabilities and their performances [7, 26, 32]. However, few have addressed studies on the interactions of the hardware components involved in the process and especially in the memory subsystem. The latter has become the most important bottleneck for the scalability and performance of the models.

Finally, studies such as Chishti's [5] from Intel Laboratories showed in a simulation environment called DLSim the problem previously raised along with memory barriers. None of the profilers found, do a study of the problem from the point of view of communication and interactions of the components involved in model training. This is the motivation for the study presented in this paper.

2. Methodology

2.1. Model Selection and Dataset

There is now a large number of models and datasets covering fields such as image classification, object detection, speech recognition, generative adversarial nets, and deep reinforcement learning. These range from models with few hidden layers such as AlexNet [12] as well as other highly complex ones that can require huge computational capabilities for their training (Inception-v3 [27] as such as Resnet [9]). It is also important to note that many of the current datasets are large because of the increase in data available to create them. As a result, it is decided to first work with two synthetic datasets to control the total size of the dataset as well as its dimensions. On the other hand, it has been decided to create a small convolutional network model to analyze the interactions made during training with the main computational resources. In general, convolutional neural networks repeat the same process layer by layer to abstract more detailed features and finally end up with a fully connected layer to determine the class to which the image entered at the beginning of the network belongs. Therefore, the use of a denser model was ruled out, estimating that the process in each convolutional layer would have similar behavior and that memory use would increase as a function of the number of filters used among other hyperparameters and, would not present variability in the interaction of the hardware components involved. The model used for testing has been implemented in both Tensorflow and

Pytorch, its architecture is shown in Tab. 1. Finally, the AlexNet implementation in Tensorflow is shown in order to observe the variation presented in the interaction of the components when varying the model.

The two synthetic datasets were generated by simulating images of three channels each with dimensions of 32x32. The sizes selected for the datasets were 60,000 (DT1) and 600,000 (DT2), each representing an in-memory occupancy of 703 Mbytes and 6,867 Mbytes respectively.

Table 1. Convolutional Neural Network Model Architecture

	Layer (type)	**Parameters**
1	Convolutional	filters = 32, kernel_size = 3x3
2	Max Pooling	pool_size = 2
3	Convolutional	filters = 64, kernel_size = 3x3
4	Max Pooling	pool_size = 2
5	Convolutional	filters = 64, kernel_size = 3x3
6	Flatten	null
7	Dense	units = 32
8	Dense	units = 10

2.2. Frameworks Selection

Currently, there has been a constant development of both hardware and software tools that are available for the implementation of the different machine learning algorithms. In terms of software, the open-source frameworks available are Tensorflow [2], PyTorch [19], Keras [1], Torch [6], CNTK [31], Caffe [10], among others.

Among the frameworks mentioned above, none has emerged as dominant in the field, and therefore the choice of the TensorFlow and PyTorch frameworks has been made for their popularity [13], their ability to work on accelerators such as GPUs, the simplicity of implementation, and their programming similarity. Another essential reason for this choice is that the monitor created to capture the interactions between training and computational resources is designed to capture Python processes and threads in the operating system.

2.3. Tool for Characterizing the Resources Consumed

This section describes the tool developed for analysis. It is designed to capture the exchanges of the different computational resources that interact during the training process of a convolutional neural network. Profilers exist to analyze the behavior of the models, such as Tensorboard for TensorFlow and PyTorch Profiler for PyTorch, but they do not measure the actual interactions between the different components involved during the training process.

The tool relies on two libraries, psutil and pynvml. The first one gives the information of the running processes as well as the resources consumed by them in the system (CPU, Memory, Disk, Network, Sensors). The second one is a wrapper for the NVML library that monitors and manages several of the states of NVIDIA GPU devices: GPU Usage, Device Memory, PCIe Bus Interaction, among others. The monitor[9] captures and records the main computational

[9]https://github.com/alejandrotorresn/PhD/blob/master/monitor.py

components that interact during model execution and training. It is designed to capture the percentage of CPU usage, the amount of memory consumed, the percentage of GPU usage, the total GPU memory, and the traffic on the PCIe bus generated by the GPU. The latter returns two data: the first measures the data transferred and the second the data received by the device through this channel. These values are recorded for the entire duration of the process execution with an interval of approximately 0.5 seconds between samples. This interval is the maximum reduction allowed by the library used for the monitor development.

2.4. Experimental Environment

To conduct these experiments we use two servers. One with the Centos 7.9 operating system and the other with Debian 10. The first compute node is equipped with 4 Intel(R) Xeon(R) CPU X7560 at 2.27 GHz processors with 64 cores in total, 125 GB of RAM, and 2 x Nvidia GeForce GTX TITAN X cards each with 12 GB of memory. This is one of the nodes of the FELIX-denomid GUANE cluster. On the other hand, to verify the results obtained, tests are carried out on a node of the chifflot cluster in Lille belonging to Grid5000[10]. This node is equipped with two Intel(R) Xeon(R) Gold 6126 at 2.60 GHz processors with 48 cores in total, 192 GB of RAM, and one 2 x Nvidia Tesla P100 cards each with 16 GB of memory. For both architectures, we use only one card per server. The versions of the frameworks used are Tensorflow 2.2.0 and PyTorch 1.4.0. Essentially, it aims at verifying and compare resource consumption and its interaction by using two types of GPUs with different compute capabilities and to show the orchestration of resources of both frameworks in different architectures. The specifications of the GPUs used in the test can be seen in Tab. 2.

Table 2. NVIDIA GPUs specifications

	GTX TITAN X	**TESLA P100**
Engine Specs		
Architecture	Maxwell	Pascal
CUDA Cores	3072	3584
Base Clock (MHz)	1000	1189
Boost Clock (MHz)	1075	1328
Memory Specs		
Memory clock (MHz)	1752.5	715
Memory clock - effective (MHz)	7010	1430
Memory size (GB)	12	16
Memory type	GDDR5	HBM2
Memory Interface Width	384-bit	4096-bit
Memory Bandwidth (GB/sec)	336.5	732

These experimental environments will be used to measure the memory requirements necessary for the training of CNNs, its source will be detailed in the next section.

[10]https://www.grid5000.fr/w/Grid5000:Home

3. Memory Requirements and Use

As mentioned earlier, a variety of deep neural network types are specialized for a set of specific fields. The study uses the model defined in Tab. 1, a model of convolutional neural networks, which are designed for tasks such as computer vision, image recognition, or object detection. The basic design of any neural network consists of an input layer, an output layer, and multiple hidden layers. With regards to convolutional neural networks, each convolutional hidden layer is responsible for extracting certain characteristics from the input data and sending them to the next one. The first convolutional layers extract more general features while the latter gets more refined features. Finally, the convolution block output data is usually sent to fully connected layers to perform the classification task if that is the purpose of the model.

Convolutional neural networks are essentially composed of three types of layers that are responsible for extracting characteristics, reducing the output data sizes of each layer, and performing classification tasks as in the case of the simulated model in this work. The first is known as a convolutional layer and applies a group of filters to the inputs generating a feature map that is obtained from the convolution between the inputs and filters and the crossing of these values by an activation function. Normally in convolutional layers, the Rectified Linear Unit activation function (ReLU[11]) is used. This function has advantages over the preceding activation functions such as its ease of calculation and greatly accelerates the convergence of stochastic gradient descent compared to the sigmoid and Tanh functions due to its linear, non-saturating form [23].

The following layer is known as Pooling and is responsible for reducing feature map sizes using one of two widespread techniques: avg-pooling and max-pooling [22]. The process of extracting features by convolutional layers and reducing data in pooling layers is performed sequentially to the last layers that are known as Fully Connect layers. This type of neural network contains three different types of data: inputs, weights, and feature maps. It is important to note that the size of the feature map data is based on the batch parameter and the number of filters used in the layer.

Once the network models are designed, they must go through the training process to obtain the desired accuracy. This process is divided into a series of iterations where each involves a path of the model both forward and backward, known as forward and backward propagation. In forward propagation, the input data traverses the network from the first to the last layer in subgroups of the size specified by the batch parameter. At the end of this step, the outputs obtained are compared with the expected outputs (Supervised Learning), and with the back-propagation algorithm gradient maps are generated that will allow updating the weights of the network.

Within this order of ideas, memory and bandwidth needs arise due to the three types of data mentioned above: inputs, weights, and feature maps. Also, another important source is the gradient maps that are generated to update the weight. It is important to note that the feature maps obtained from forward propagation should be kept in memory until the gradient maps update the network values. This large amount of data generated by the network, along with the size of the data set, is responsible for the memory limits on some hardware architectures.

The next section will show the results obtained from the monitor. This measures the consumption of computational resources involved, as well as the times of the workloads. In partic-

[11] https://cs231n.github.io/neural-networks-1

ular, the limits reached in the memory generated by the data set, the data types generated by the model, and their training will be observed.

4. Evaluation and Results

4.1. Training Times

In this first part, the training times that both frameworks took in both architectures are shown using both datasets and various batch size. In Tab. 3 it can be observed that increasing the batch size significantly reduces training times but does not affect the behavior of resource use by frameworks, as will be seen in the next section. It is important to note that by keeping the same data set and increasing the batch size, the time differences between TensorFlow and PyTorch were significantly reduced.

Table 3. Training times (sec)

	FELIX				**Chifflot**			
	TensorFlow		**PyTorch**		**TensorFlow**		**PyTorch**	
Batch size	**DT1**	**DT2**	**DT1**	**DT2**	**DT1**	**DT2**	**DT1**	**DT2**
32	165	1591	313	3048	32	338	79	637
64	100	1026	165	1682	24	212	42	394
128	74	658	87	796	20	173	32	295
256	62	501	59	535	19	156	27	243
512	52	423	52	466	18	147	24	219

On the other hand, a significant difference can be observed in the training times of both frameworks in which small batches are used. The main reason for this, as shown in Fig. 1 and Fig. 3, is the degree of CPU and GPU usage by each of the frameworks. TensorFlow has maximized device memory usage and GPU usage, while PyTorch has prioritized host memory usage with less GPU usage, behavior that has been observed in both architectures and will be discussed in the next section.

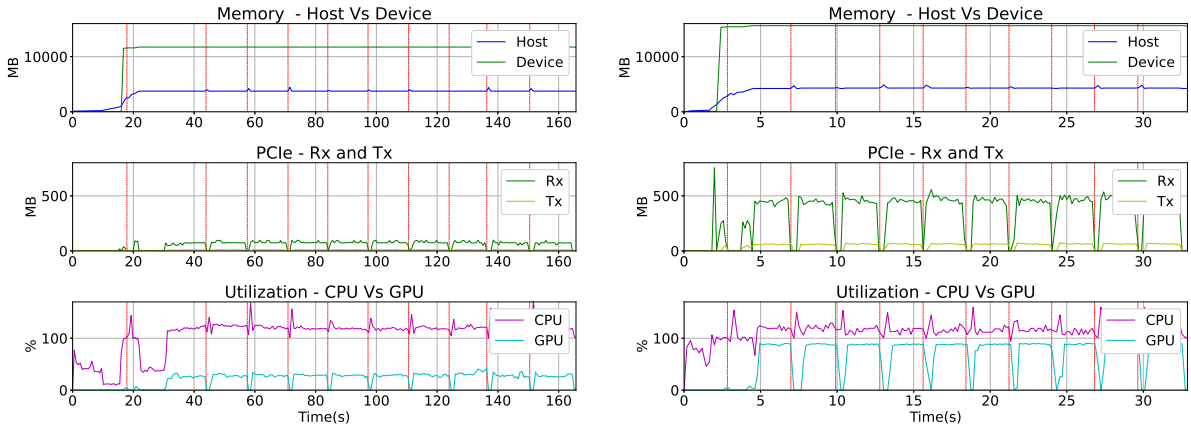
4.2. Interrelationship of Hardware Components

The experiments have been performed for different batch sizes, but only the particular results for batch size 32 are shown, since the results with the other batch sizes show similar behavior in terms of the orchestration of hardware resources measured by the monitor and allows us show the behavior studied more clearly. As mentioned in Section 2.4, two different platforms are used to carry out the experiments in order to be able to perform a contrast regarding the use of resources and their orchestration.

Figure 1 shows the results obtained during the model training process using the TensorFlow framework and a batch size of 32 for the 60,000 images dataset. The vertical red lines represent the beginning of each of the model training epochs that were set at a value of 10 for the study. The first part shows the memory consumption of the host and the device, the second part of the figure shows the communication of the device in its input channel (Rx) and its output channel (Tx) through the PCIe. Finally, the third part of the figure shows the percentage of memory consumption in both CPU and GPU. It should be noted that this third part of the graph

shows values that may exceed values of 100% due to the current configuration of the monitor developed. This monitor sums the percentage of each of the cores involved in the network training process without discriminating the amount involved, which means that the servers involved in the experiment could reach values of 4,800% or 6,400% for Chifflot and FELIX respectively.

When the datasets used in the experiment have been described, it has been commented that their sizes have been 703 MB and 6,867 MB for DT1 and DT2 respectively. These sizes are the space occupied by them in RAM memory when loaded in Python. The memory occupied by the model (inputs, weights, feature maps and gradient maps) reaches a maximum of 4,446 MB in the Host and 11,743 MB in the Device of the FELIX server and 4,880 MB in the Host and 15,621 MB on the Device of the Chifflot serve using DT1 and present little variability during the model training time as can be seen in the first part of Fig. 1. This difference between what the dataset occupies in memory and what it occupies together with the model shows that despite the small convolutional model used, there is a great impact on the memory of both the host and the device.



(a) FELIX - GUANE

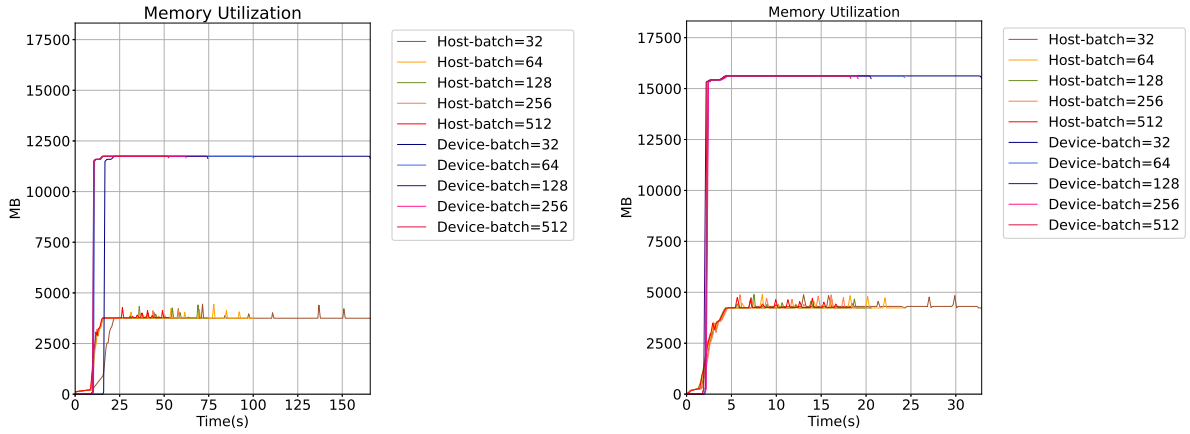
(b) Chifflot - Lille

Figure 1. Interaction of hardware components - DT1 - batch -32 - TensorFlow

The second part of Fig. 1 shows the interaction of the GPU through the PCIe bus. For both Felix and Chifflot, once the first training epoch has been initialized, a behavior with very little variability is presented with the difference that when the use of the GPU increases, the traffic in it increases in the same way. Finally, the third part of Fig. 1 shows the CPU and GPU usage during model training. The most important thing to highlight is that the change in the architecture allows the framework to make greater use of the GPU and reduce the training times of the model, but there is not a great variability in the use of resources, they are very similar during each epoch.

During the first training epoch fluctuating consumption of resources, as well as low values in the data transmitted and received by the device due to the start of the neural network by TensorFlow, are observed. Once the initial configurations are complete, the processes stabilize and resources consumption presents similar values for each epoch with very little variability during training time as mentioned before.

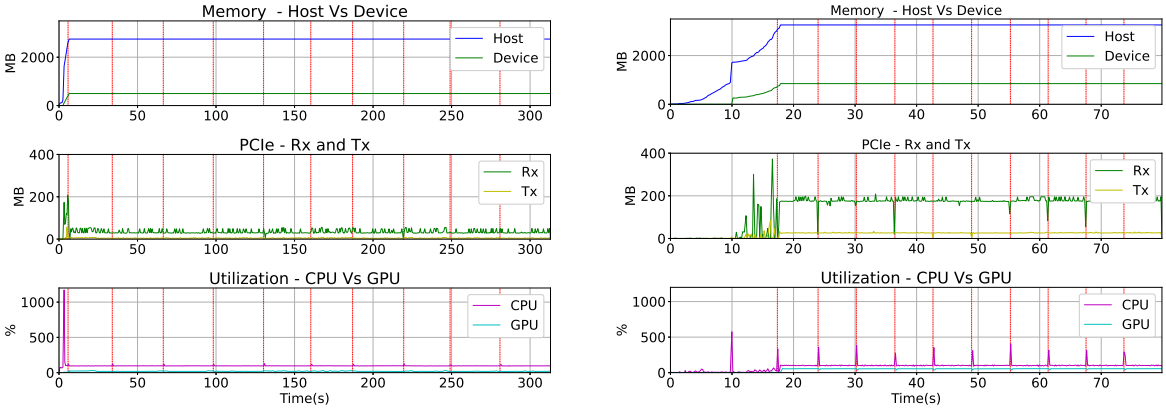
Figure 2 shows memory consumption for batches 32, 64, 128, 256 and 512 on both architectures. In addition, it is shown that for the same data set and the same neural network model, the memory consumption of the host does not present significant variations while the memory



(a) FELIX - GUANE (b) Chifflot - Lille

Figure 2. Host and device memory consumption for each of the batches set - DT1 - TensorFlow

consumption in the device presents small fluctuations that stabilize at a baseline of consumption. It should be noted that the upper part of the figure shows the memory consumption in the host while the lower part is the consumption in the device. For each batch, the duration of the training is different, and therefore, in the figure, it is observed how each corresponding line is cut.

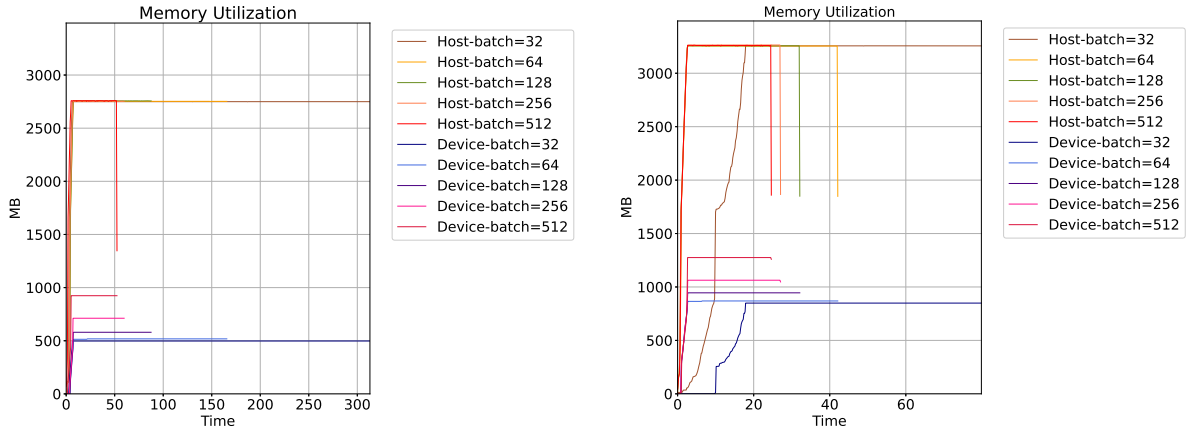


(a) FELIX - GUANE (b) Chifflot - Lille

Figure 3. Interaction of hardware components - DT1 - batch -32 - PyTorch

Figure 3 and Fig. 4 display the results for the same neural network, the same batch values as well as the same dataset using the PyTorch framework. As for PyTorch, the maximum value of memory consumption in the host has been 2,749 MB while in the device memory it has been 497 MB for FELIX and 3,256 MB in the host memory, and 849 MB in device memory in Chifflot using DT1. As with Tensorflow, there are no major variations in resource consumption after the first training period as can be seen in Fig. 1 and Fig. 3. In the same way, by increasing the use of the GPU in Chifflot, the traffic on the PCIe bus through the RX and Tx increases.

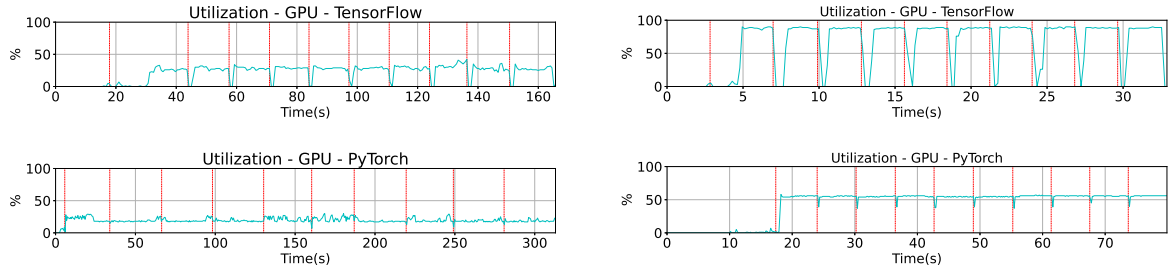
PyTorch has large peaks in CPU usage during model initialization and at the beginning of each training epoch, unlike Tensorflow, which tends to have more homogeneous behavior in the use of resources. But, the use of the device by PyTorch is less for this particular case of a



(a) FELIX - GUANE (b) Chifflot - Lille

Figure 4. Host and device memory consumption for each of the batches set - DT1 - PyTorch

dataset and a small model. This differentiation of the use of the GPU is observed in Fig. 5. The difference remains despite the change in server and therefore the architecture used in the tests.



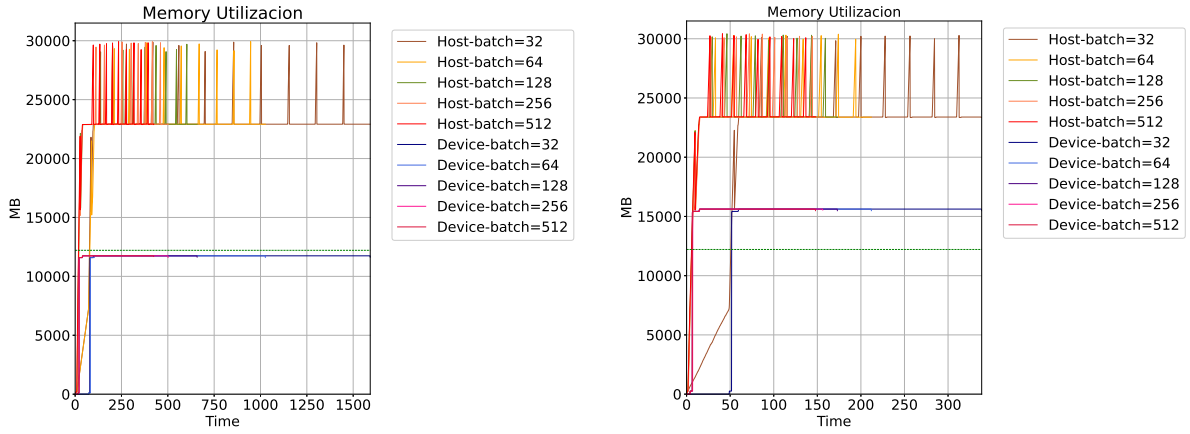
(a) FELIX - GUANE (b) Chifflot - Lille

Figure 5. GPU Utilization - DT1 - Tensorflow vs PyTorch

Figure 6 and Fig. 7 show the memory consumption during network training for the DT2 data set using the same CNN model implemented in both frameworks. The behavior in terms of memory use is similar despite the change in the dataset, increasing the memory requirement for both the host and the device as expected. Regarding the behavior of the two servers with different architectures, the training times decrease considerably with the use of more recent architecture, but the behavior of the use of the resources remains similar despite the change in the dataset and the architecture of the device for both frameworks.

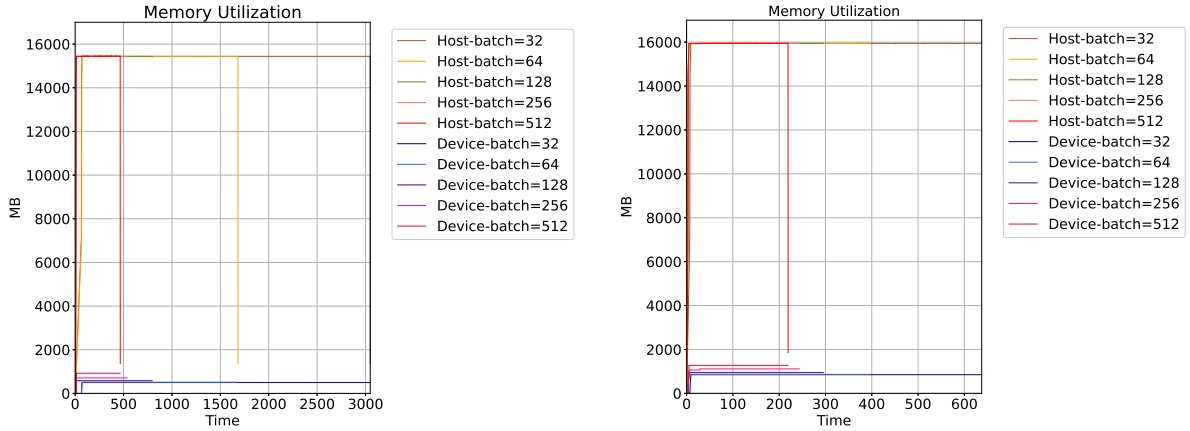
Figure 5 showed us that Tensorflow made better use of GPU for batch size 32 and that training times were improved by making an architecture change. Although the memory use behavior of both frameworks is similar, as the batch and dataset size is increased using the same model, improvements in the use of resources by PyTorch begin to show. These behaviors can be observed in Fig. 8 and Fig. 9 where the results obtained using DT2 with a batch of size 512 are shown. An increase in dataset size and batch size results in a significant increase in GPU usage by PyTorch without exceeding Tensorflow's resource handling. In the same way, the change in architecture has improved training times but hasn't shown differentiation in the orchestration of resources by the frameworks.

Finally, emphasizing memory consumption by both frameworks, it is observed that both TensorFlow and PyTorch present a maximum limit of both the host and device memory, presenting



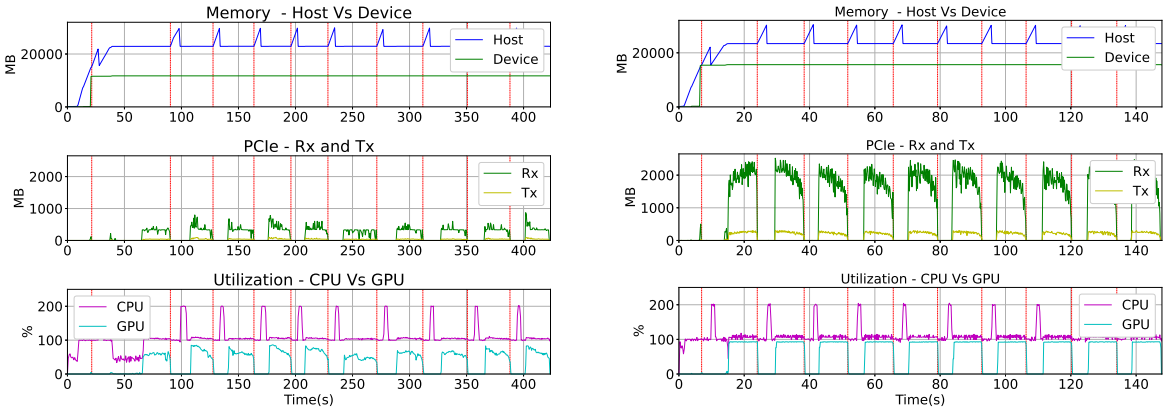
(a) FELIX - GUANE (b) Chifflot - Lille

Figure 6. Host and device memory consumption for each of the batches set - DT2 - TensorFlow



(a) FELIX - GUANE (b) Chifflot - Lille

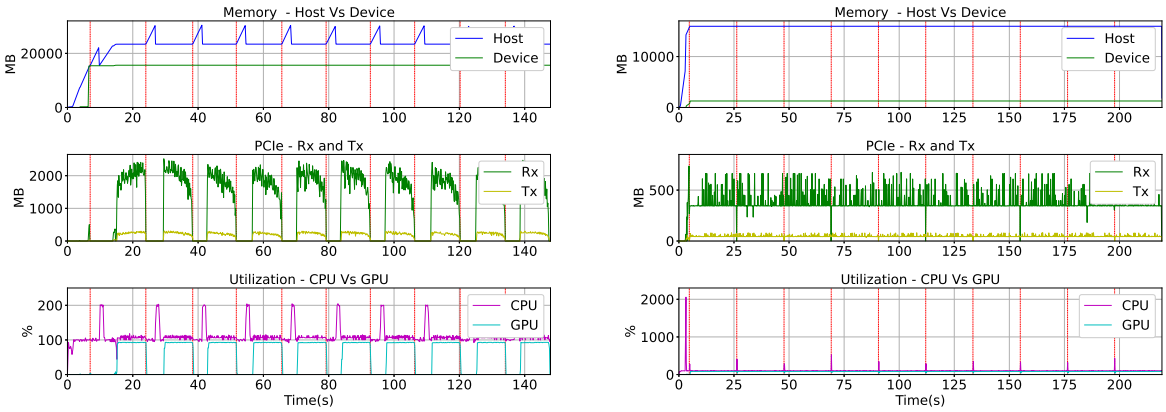
Figure 7. Host and device memory consumption for each of the batches set - DT2 - PyTorch



(a) FELIX - GUANE (b) Chifflot - Lille

Figure 8. Interaction of hardware components - DT2 - batch 512 - Tensorflow

some fluctuations at the beginning of each training epoch. PyTorch presents an interesting feature in terms of memory usage, where batch variation incurs variations of device memory usage. Table 4 shows the maximum memory consumption peaks by the host and the device for the



(a) FELIX - GUANE

(b) Chifflot - Lille

Figure 9. Interaction of hardware components - DT2 - batch 512 - PyTorch

DT1 and DT2 datasets for both frameworks and architectures with a batch size of 32. It should be noted that PyTorch maintains a better use of the memory of both the host and the device at the cost of increasing the training time of the model.

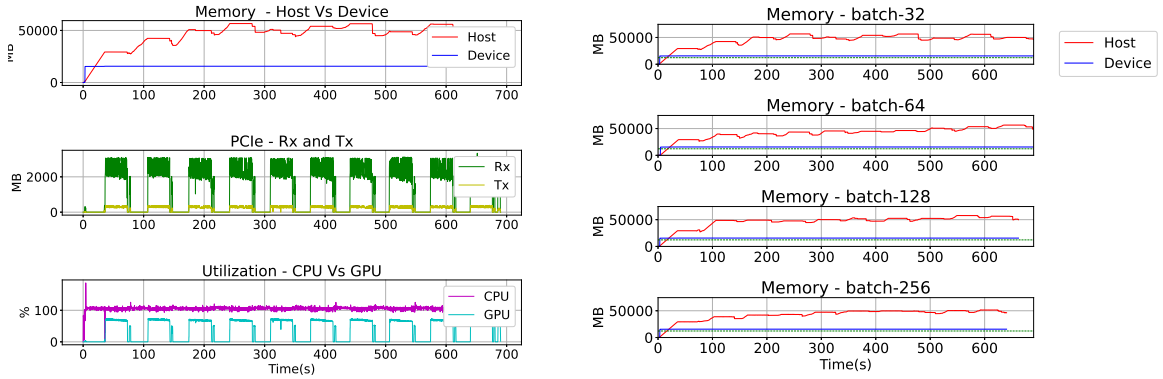
Table 4. Maximum Memory Consumption (MB) - batch-32

		FELIX		**Chifflot**	
		DT1	DT2	DT1	DT2
TensorFlow	Host	4446	29883	4880	30282
	Device	11743	11743	15621	15621
PyTorch	Host	2749	15439	3256	15945
	Device	497	497	849	849

4.3. AlexNet

This section shows the similarity of resource use and interaction by TensorFlow during AlexNet training. The training of this model is carried out exclusively on the Chifflot server using a batch size of 32 to measure the orchestration of the components and batches of size 32, 64, 128, and 256 are used for the study of both memory usage on the host as well as on the device, as shown in Fig. 10.

The results presented in this section show the behavior of memory and allowed observing the interaction of the hardware components that intervene from the beginning of the training workload. This allowes us to observe that the computational load is established during the first epoch of training and to see its little variability during the iterative process. In the following section, the results of other works will be discussed and related to those obtained during this study.



(a) Interaction of hardware components (b) Host and device memory consumption

Figure 10. Resource use by TensorFlow during AlexNet training

5. Discussion

The works presented by [5], Zhu [32] and Dai [7] show the bottleneck generated mainly by the memory subsystem when trying to train dense models with large datasets. These last two studies are mainly focused on resource consumption by model to high-level, but Chishti's work, done in a simulation environment, studies the behavior and interaction of the components involved during training. It is limited to the interaction between the processor and memory leaving aside accelerator devices such as the GPU. In order to solve or significantly reduce this problem, methods have emerged such as the optimization of the directed asynchronous graph created at the time of the execution of the model, as proposed by Le [15] and Boemer [3] or, methods of working with sparses matrices [20] to reduce memory consumption during training processes. They have also considered changing the way, memory levels are used during training as mentioned in Rhu [21] and Lim [17] works but they also present bottlenecks due to the high level of communication that occurs through the PCIe bus. In relation to the latter, new solutions are developed on specialized servers. They use a high-speed network for communication between throttled memory allowing to expand the memory capacity available in training by using multiple accelerators without using the system bus as Kwon proposes [14].

Following these ideas, most of the work done to address the problem of memory limit is using methods that reduce the memory size, avoiding the problem of the limits in communication between devices and between the same memory subsystem or the creation of new architectures specialized in the training of deep neural network models. The biggest advantage of these solutions is the implementation of sparse matrices in the training process because they can run on conventional architectures and be highly parallel.

Therefore, this work seeks to analyze the interaction between the main hardware components involved during training to look from another approach for bottlenecks that may occur in this process and then be able to address new optimization approaches. The results presented show above all the importance of finding new methods to optimize the use of the memory subsystem, host-device communications, highlighting the variability of training times when there is no effective communication with the accelerators. It is important to note that the traffic generated by the two communication lines of the device (Tx and Rx) is greater when there is greater use of the memory of the device. As a result, as the memory of the device becomes saturated, the traffic on the Rx line of the device increases, and also the traffic on the PCIe bus increases.

These latter measurements have not been studied in-depth, but they represent an interesting fact for the memory optimization used by CNN.

Conclusions

In this work, a characterization of the resources consumed and their interaction during the training of a convolutional neural network has been performed using two synthetic data sets. Batch size variability has not affected overall host or device memory consumption, and maximum rooflines have been set from the start of the training process based on dataset size and model complexity. On the other hand, training times, if they are affected by the batch size as observed in Fig. 2 and Fig. 4, the fact that they consume more host memory than the device are closely related to this variability. Finally, the monitor created to capture the resources and interaction between them has helped show the bottleneck that can become the memory subsystem. It is available for download.

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (`https://www.grid5000.fr`) in France and, the advanced computing platforms of the High Performance and Scientific Computing Center at Universidad Industrial de Santander (SC3UIS) (`http://www.sc3.uis.edu.co`) in Colombia.

References

1. Keras: Deep Learning for humans. `https://github.com/keras-team/keras` (2015), accessed: 2020-11-10

2. Abadi, M., Barham, P., Chen, J., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2-4 Nov. 2016, Savannah, GA, USA. pp. 265–283. USENIX Association, USA (2016), DOI: 10.5555/3026877.3026899

3. Boemer, F., Lao, Y., Cammarota, R., et al.: NGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data. In: Proceedings of the 16th ACM International Conference on Computing Frontiers, 30 April-2 May 2019, Alghero, Italy. pp. 3–13. Association for Computing Machinery, New York, NY, USA (2019), DOI: 10.1145/3310273.3323047

4. Chen, Y., Luo, T., Liu, S., et al.: DaDianNao: A Machine-Learning Supercomputer. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, 13-17 Dec. 2014, Cambridge, UK. pp. 609–622. IEEE (2014), DOI: 10.1109/MICRO.2014.58

5. Chishti, Z., Akin, B.: Memory System Characterization of Deep Learning Workloads. In: Proceedings of the International Symposium on Memory Systems, 30 Sept.-3 Oct. 2019,

Washington, District of Columbia, USA. pp. 497–505. Association for Computing Machinery, New York, NY, USA (2019), DOI: 10.1145/3357526.3357569

6. Collobert, R., Kavukcuoglu, K.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)

7. Dai, W., Berleant, D.: Benchmarking Contemporary Deep Learning Hardware and Frameworks: A Survey of Qualitative Metrics. In: 2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI), 12-14 Dec. 2019, Los Angeles, CA, USA. IEEE (2019), DOI: 10.1109/cogmi48466.2019.00029

8. Hameed, R., Qadeer, W., Wachs, M., et al.: Understanding Sources of Inefficiency in General-Purpose Chips. SIGARCH Comput. Archit. News 38(3), 37–47 (2010), DOI: 10.1145/1816038.1815968

9. He, K., Zhang, X., Ren, S., et al.: Deep Residual Learning for Image Recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 27-30 June 2016, Las Vegas, NV, USA. pp. 770–778. IEEE (2016), DOI: 10.1109/CVPR.2016.90

10. Jia, Y., Shelhamer, E., Donahue, J., et al.: Caffe: Convolutional Architecture for Fast Feature Embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, 3-7 Nov. 2014, Orlando, Florida, USA. pp. 675–678. Association for Computing Machinery, New York, NY, USA (2014), DOI: 10.1145/2647868.2654889

11. Jouppi, N.P., Young, C., Patil, N., et al.: In-Datacenter Performance Analysis of a Tensor Processing Unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, 24-28 June 2017, Toronto, ON, Canada. pp. 1–12. Association for Computing Machinery, New York, NY, USA (2017), DOI: 10.1145/3079856.3080246

12. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. Commun. ACM 60(6), 84–90 (2017), DOI: 10.1145/3065386

13. Kumar, V.: 5 Deep Learning Frameworks to Consider for 2020. `https://opendatascience.com/5-deep-learning-frameworks-to-consider-for-2020` (2020), accessed: 2020-11-10

14. Kwon, Y., Rhu, M.: Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 20-24 Oct. 2018, Fukuoka, Japan. pp. 148–161. IEEE (2018), DOI: 10.1109/MICRO.2018.00021

15. Le, T.D., Imai, H., Negishi, Y., et al.: Automatic GPU Memory Management for Large Neural Models in TensorFlow. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management, 23 June 2019, Phoenix, AZ, USA. pp. 1–13. Association for Computing Machinery (2019), DOI: 10.1145/3315573.3329984

16. Li, A., Song, S.L., Chen, J., et al.: Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE Transactions on Parallel and Distributed Systems 31(1), 94–110 (2020), DOI: 10.1109/TPDS.2019.2928289

17. Lim, K., Turner, Y., Santos, J.R., et al.: System-level implications of disaggregated memory. In: IEEE International Symposium on High-Performance Comp Architecture, 25-29 Feb. 2012, New Orleans, LA, USA. pp. 1–12. IEEE (2012), DOI: 10.1109/HPCA.2012.6168955

18. Mayer, R., Jacobsen, H.A.: Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools. ACM Comput. Surv. 53(1) (2020), DOI: 10.1145/3363554

19. Paszke, A., Gross, S., Chintala, S., Chanan, G.: PyTorch. `https://github.com/pytorch/pytorch` (2016), accessed: 2020-11-10

20. Qin, E., Samajdar, A., Kwon, H., et al.: SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 22-26 Feb. 2020, San Diego, CA, USA. pp. 58–70. IEEE (2020), DOI: 10.1109/HPCA47549.2020.00015

21. Rhu, M., Gimelshein, N., Clemons, J., et al.: vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 15-19 Oct. 2016, Taipei, Taiwan. pp. 1–13. IEEE (2016), DOI: 10.1109/MICRO.2016.7783721

22. Saeedan, F., Weber, N., Goesele, M., Roth, S.: Detail-Preserving Pooling in Deep Networks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 18-23 June 2018, Salt Lake City, UT, USA. pp. 9108–9116. IEEE (2018), DOI: 10.1109/CVPR.2018.00949

23. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Networks 61, 85–117 (2015), DOI: 10.1016/j.neunet.2014.09.003

24. Sergeev, l.: Horovod. `https://github.com/horovod/horovod` (2017), accessed: 2020-11-10

25. Shatnawi, A., Al-Bdour, G., Al-Qurran, R., et al.: A comparative study of open source deep learning frameworks. In: 2018 9th International Conference on Information and Communication Systems (ICICS), 3-5 April 2018, Irbid, Jordan. pp. 72–77. IEEE (2018), DOI: 10.1109/IACS.2018.8355444

26. Simmons, C., Holliday, M.A.: A Comparison of Two Popular Machine Learning Frameworks. J. Comput. Sci. Coll. 35(4), 20–25 (2019), DOI: 10.5555/3381631.3381635

27. Szegedy, C., Vanhoucke, V., Ioffe, S., et al.: Rethinking the Inception Architecture for Computer Vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 27-30 June 2016, Las Vegas, NV, USA. pp. 2818–2826. IEEE (2016), DOI: 10.1109/CVPR.2016.308

28. Wahib, M., Zhang, H., Nguyen, T.T., et al.: Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 9-19 Nov. 2020, Atlanta, Georgia. IEEE Press (2020), DOI: 10.5555/3433701.3433726

29. Wang, Y., Yang, C., Farrell, S., et al.: Time-Based Roofline for Deep Learning Performance Analysis. In: 2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS), 11 Nov. 2020, Atlanta, GA, USA. pp. 10–19. IEEE (2020), DOI: 10.1109/DLS51937.2020.00007

30. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM 52(4), 65–76 (2009), DOI: 10.1145/1498765.1498785

31. Yu, D., Eversole, A., Seltzer, M., et al.: An Introduction to Computational Networks and the Computational Network Toolkit (2014), https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/, accessed: 2020-11-10

32. Zhu, H., Akrout, M., Zheng, B., et al.: Benchmarking and Analyzing Deep Neural Network Training. In: 2018 IEEE International Symposium on Workload Characterization (IISWC), 30 Sept.-2 Oct. 2018, Raleigh, NC, USA. pp. 88–100. IEEE (2018), DOI: 10.1109/IISWC.2018.8573476

DOI: 10.14529/jsfi210105

The MareNostrum Experimental Exascale Platform (MEEP)

Alexander Fell[1]***, Daniel J. Mazure***[1]***, Teresa C. Garcia***[1]***, Borja Perez***[1]***, Xavier Teruel***[1]***, Pete Wilson***[1]***, John D. Davis***[1]

Nascent Open Source Instruction Set Architectures such as OpenPOWER or RISC-V, allow software/hardware co-designers to fully utilize the underlying hardware, modify it or extend it based on their needs. In this paper, we introduce the vision of the MareNostrum Experimental Exascale Platform (MEEP), an Open Source platform enabling software and hardware stack experimentation targeting the High-Performance Computing (HPC) ecosystem. MEEP is built with state-of-the-art FPGAs that support PCIe and High Bandwidth Memory (HBM), making it ideal to emulate chiplet-based HPC accelerators such as ACME, at the chip, package, and/or system level. MEEP provides an FPGA Shell containing standardized interfaces (I/O and memory), enabling an emulated accelerator to communicate with the hardware of the FPGA and ensures quick integration. The first demonstration of MEEP is mapping a new accelerator, the Accelerated Compute and Memory Engine (ACME), on to this digital laboratory. This enables exploration of this novel disaggregated architecture, which separates the computation from the memory operations, optimizing the accelerator for both dense (compute-bound) and sparse (memory-bandwidth bound) workloads. Dense workloads focus on the computational capabilities of the engine, while dedicated processors for memory accesses optimize non-unit stride and/or random memory accesses required by sparse workloads. MEEP is an open source digital laboratory that can provide a future environment for full-stack co-design and pre-silicon exploration. MEEP invites software developers and hardware engineers to build the application, compiler, libraries and the hardware to solve future challenges in the HPC, AI, ML, and DL domains.

Keywords: high performance computing (HPC), accelerator, software stack, open source hardware.

Introduction

Today, Linux is the omnipresent Operating System (OS) in the Internet-of-Things (IoT) space running on small embedded systems, in the mobile space in the form of Android and it is ubiquitous in High-Performance Computing (HPC) and cloud-based systems with various Open Source Software (OSS) components built on top. OSS provides the foundational building blocks for the OS, toolchain, runtimes, frameworks, libraries, and all the way up to the application layer enabling rapid development and extension of any layer in the software stack.

However, when examining hardware, current commercial off the shelf solutions (COTS) are closed hardware ecosystems that only enable integration at the peripheral level through a defined interface such as PCIe and proprietary drivers. This stifles innovation by limiting optimizations to the software stack, preventing true software/hardware co-design. This problem has been intensified by the end of Dennard scaling and the dramatic slowdown in Moore's Law, requiring specialized hardware, in form of accelerators, to meet the system power and performance requirements. At the same time, the software stack is evolving, becoming more abstract. This enables higher programmer productivity, but sacrificing hardware efficiency. Thus, application owners will need to co-design the full stack, all layers of hardware and software, in order to meet their performance and power (FLOPs/W) targets. This level of tight integration is not possible in a closed or even partially open ecosystem.

[1]Barcelona Supercomputing Center (BSC), Barcelona, Spain

There have been Open Source Hardware (OSH) platforms in the past, but Moore's Law and many other reasons inhibited their adoption, e.g. continuous general purpose processor performance improvements, time to market, cost, software development, etc. Furthermore, unlike Linux, previous OSH was entangled in the companies that created them and/or generally poor quality. Mirroring the same model as Linux, RISC-V has followed a similar development path and has enjoyed significant industrial and academic adoption. The RISC-V ecosystem is in the nascent period where it can become the de facto open hardware platform of the future, having the same opportunity in hardware that Linux created as a foundation for OSS. This enables the co-design of the RISC-V hardware and the entire software stack, creating a better overall solution than the closed hardware approach that is done today.

In this paper, we introduce a new digital laboratory for software and hardware development, called the MareNostrum Experimental Exascale Platform (MEEP). MEEP is a high-level hardware emulation platform that also can be used as a Software Development Vehicle (SDV) and is described in Section 1. Its name (meep meep) pays homage to the RoadRunner supercomputer, the first PetaFLOPS system [1]. MEEP leverages OSS and extends various software layers in the stack to run on a RISC-V based accelerator and consists of a set of defined hardware interfaces, enabling a wide range of accelerators and processors to be emulated. This results in a platform that permits rapid development and testing of new HPC and High Performance Data Analytics (HPDA) hardware accelerators and the associated software ecosystem. The initial targeted software stack is presented in Section 2, comprising of existing HPC and emerging HPDA applications in fields such as machine learning, computer vision, and deep learning, co-designed with those accelerators to meet the desired power and performance expectations.

We demonstrate MEEP's SDV and hardware emulation capabilities by mapping an accelerator called the Accelerated Compute and Memory Engine (ACME) into MEEP. ACME separates computation and memory operations into compute and memory tiles similar to [24]. All tiles are interconnected by a Network-on-Chip (NoC) to ensure scalability. The compute tiles feature RISC-V scalar cores supporting a variety of coprocessor accelerators with a common set of interfaces. Those coprocessor accelerators support vector operations executed on a Vector Processing Unit (VPU) or alternatively, Systolic Arrays (SA) for image processing and neural networks, all connected to the scalar core through the Open Vector Interface (OVI) [22]. Furthermore, the memory tile is responsible for satisfying all memory operations.

Section 3 describes the ACME architecture in greater detail. Section 4 details the FPGA infrastructure and mapping ACME into the MEEP platform. We provide a synopsis of related work in Section 5 followed by a conclusion with a description of future work of this project.

1. MareNostrum Experimental Exascale Platform (MEEP)

MEEP is a flexible FPGA-based emulation platform based on European IP blocks, that explores hardware/software co-designs for Exascale Supercomputers and other hardware targets. As a platform, MEEP provides a foundation for building European-based chips and infrastructure to enable rapid prototyping, using a library of IPs and a standard set of interfaces to the Host CPU and other FPGAs in the system, using the FPGA Shell (refer to Section 4.1). In addition to RISC-V architecture and hardware ecosystem improvements, MEEP also advances the RISC-V software ecosystem with an enhanced and extended software toolchain and software stack, including a suite of HPC and HPDA applications.

MEEP ambitions to play two important roles within the Exascale computation paradigm:

- SDV platform: Enables software development and experimentation, accelerating software maturity compared to the software simulation limitations by enabling software readiness for new hardware. In order to do this, MEEP executes the whole software stack, enabling a set of tools and mechanisms to integrate new functionalities for future challenges in the HPC and HPDA domains. MEEP is designed to run traditional HPC workloads and ecosystems. Furthermore, we see a broader set of applications with high compute requirements in new HPDA AI/ML/DL frameworks that will also be supported, as well as more complex workflows based on experimental programming models (e.g., COMPSs). More details are provided in Section 2.
- Pre-silicon validation platform: Allows testing and validating of new IPs blocks and/or systems before being committed to silicon. Thus, MEEP can leverage the FPGA built-in components and hardware macros, and efficiently map other components (IPs or custom designs) to the available structures of the FPGA. In addition, MEEP provides a foundation for building chips and infrastructure to enable rapid prototyping using a library of IPs and a standard set of interfaces to the Host CPU and other FPGAs in the system using the MEEP FPGA Shell (PCIe, HBM, DDR, Ethernet, and other interfaces), detailed in Section 4.

In Fig. 1, the whole MEEP software and hardware stack is shown. The underlying hardware, emulated by an FPGA, is able to run the software stack including the Linux OS as described in Section 2. The communication between the emulated hardware and software is enabled by the MEEP FPGA Shell, which provides a set of standard interfaces to the I/O and memory components available in the FPGA, such as memory controllers to the memory(s), Ethernet, or PCI Express connectivity. Within the MEEP FPGA Shell, there is an emulation payload that is user defined ranging from High-Level Synthesis (HLS) accelerators to RTL accelerators and/or processors. In this paper, we present an emulated accelerator based on ASIC RTL containing a self-hosted accelerator called ACME (refer to Section 3). Finally, MEEP provides a software and hardware toolchain for debugging, profiling and performance monitoring.

2. Software Stack

The MEEP software stack includes all the levels at which the software can operate: from the application level to the low-level operating system services (e.g., communication). Fig. 1 provides a detailed view of this software stack running on top of the *FPGA Emulated Hardware Accelerator* layer. We use a top-down approach to describe the software layers in the stack.

We have identified a set of target HPC and HPDA applications to demonstrate the SDV capabilities of MEEP. These workloads include traditional HPC applications (e.g. Alya, Quantum Expresso, OpenIFS, and NEMO), HPDA applications based on TensorFlow [5] and Apache Spark [34] (e.g. MLperf, AIS-Predict and SMUFIN), and work-flows based on the COMPSs [16] programming model (e.g. Guidance, MLMC, BioBB, NMMB-Monarch, and Dislib). This wide set of applications provides a representative suite of highly-relevant HPC and HPDA workloads that can be enabled with MEEP as an SDV, executing actual code on the emulated ACME accelerator.

We take a bottom-up approach to the software-hardware co-design methodology, in terms of the SDV support. From the applications shown in Fig. 1, we extract simplified computation kernels that represent a majority of the execution time of the application. These simplified, yet high-relevance, software microkernels are considered as the starting point of the analysis and verification process, and include kernels such as GEMM, SpMV, FFT, Somier, and AXPY. The latter is used as a guiding example in Section 3.4. In addition to the mentioned workloads, we

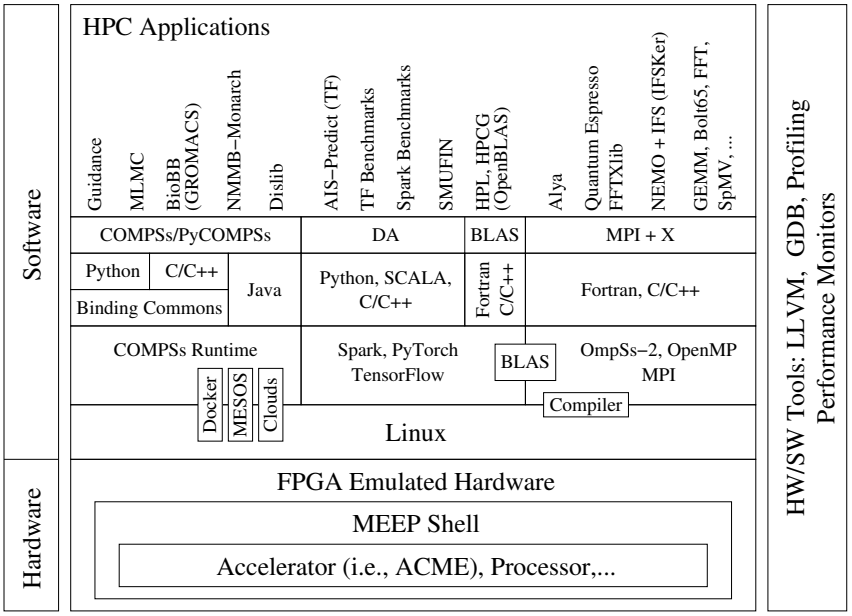


Figure 1. Full Stack of the MEEP design

also include a set of benchmarks and kernels. Benchmarks such as the High-Performance Linpack, Conjugate Gradient, or other mock-up applications like the FFTXlib code reproduce the behavior of the Quantum Expresso, enabling the testing of the expected system performance using a simpler code rather than the entire application. In the HPDA context, the TensorFlow and Apache Spark benchmarks are used to test data analytic service behavior. Finally, once all of these smaller codes run on the SDV, we can execute the complete applications with confidence. Note, this set of applications and benchmarks also includes applications requiring specific hardware characteristics such as the processing of images or natural languages. The main objective of this set of programs is to evaluate the performance of special-purpose hardware accelerators, i.e., SAs and data streaming.

Below the application level we can find the compiler support and the use of different programming languages. The MEEP project bases its optimization support on LLVM and Clang [14]. In this context, we are developing compiler extensions that exploit the performance of the ACME architecture features. The set of selected applications imposes a great variety of programming languages (e.g. C/C++, Fortran, Java, or Python), but the lack of sufficient software ecosystem maturity currently limits the direct use of all these application and certain optimizations. When this is the case, we rely on the generic compilation mechanism supported by the system and the use of third-party optimized libraries to exploit features or limited capabilities of the RISC-V software ecosystem and toolchain.

In addition to the compilation support, the applications have additional libraries and services for commonly optimized algorithms for the platform (e.g. BLAS-like library services). Additional examples include support for run-time libraries for the traditional HPC applications and new work flows (i.e., MPI [17], OpenMP [18], and COMPSs [16]). Porting and optimizing these libraries and services for the RISC-V based ACME architecture is a huge step in the direction of abstraction between the application level and the platform's OS, as well as generally improving the RISC-V software ecosystem maturity.

Further, the ACME architecture also supports the use of containers as the fundamental mechanism to pack, deploy and offload the execution of kernels and applications. The implementation process for containerization is initially based on the COMPSs programming model, as it is consid-

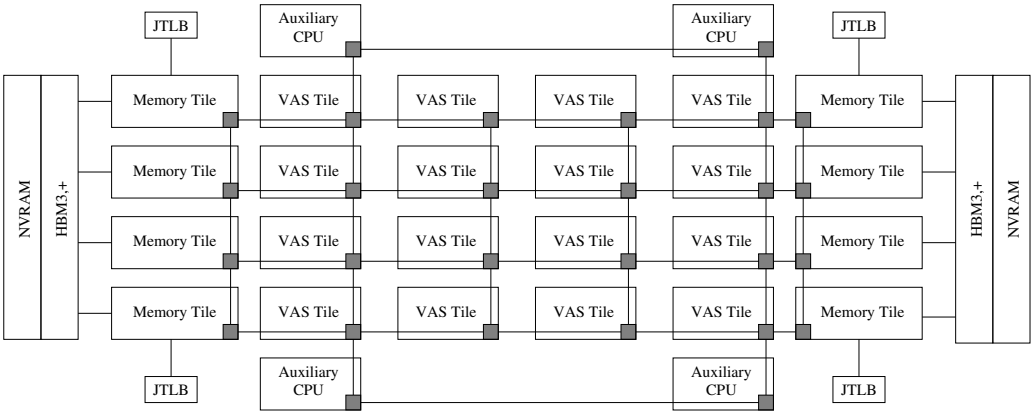


Figure 2. An overview of the Accelerated Compute and Memory Engine (ACME)

ered the most complex use case. There is the more common approach that packages application tasks, phases or stages in containers. However, if this cannot be supported by the application, we can use a different approach where the container is considered as a single-phase work-flow execution, i.e. a COMPSs application is defined a single task or container.

The OS is the lowest software layer in the software stack and MEEP relies on the Linux OS as the engine for this hardware abstraction. A modified, lightweight version of Linux takes care of the low-level duties such as discovering, enumerating, and initializing all physical components of the system. In order to reduce data movement in the system, the ACME architecture executes most of the application code in the accelerator (a self-hosted accelerator), requiring a minimal set of OS features for this execution mode. The traditional Host CPU provides services, like global resource management and name services. However, for pragmatic implementation reasons, ACME also supports a traditional offload execution mode, similar to what is used for General Purpose GPU computation today. In both execution modes, a minimum OS support is required in ACME to configure the hardware and provide the fundamental services: application execution, access to memory, or offload and execute a kernel.

3. Accelerated Compute and Memory Engine (ACME)

The ACME accelerator, as a demonstrator for MEEP, represents the desire to improve the performance for dense as well as sparse workloads. Dense workloads are compute bound, since the performance depends on the number of fused multiply-add (FMA) or fused multiply-accumulate (FMAC) units available in the system, coupled with a reasonable memory hierarchy that can efficiently exploit caching and data reuse. Unfortunately, sparse workloads are limited by the memory bandwidth and hence the focus shifts from computational units to maximizing memory utilization. The purpose of ACME is to find a reasonable balance between the memory hierarchy design for sparse workloads and the FMA architecture for dense workloads.

ACME relies on multiple specialized cores to manage the internal and external resources to improve energy efficiency and performance. It decouples the arithmetic and memory operations by disaggregating the memory access from the execution [24].

3.1. Overview

Moving from a high level view of ACME (Fig. 2) down to a more detailed view (Fig. 3), the core of the accelerator is the Vector And Systolic (VAS) Accelerator Tile. It consists of a cluster of

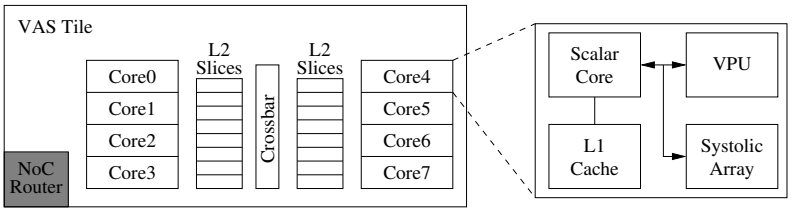


Figure 3. The VAS Tile of ACME

eight scalar RISC-V cores with each core having the capability of supporting several coprocessors: a Vector Processing Unit (VPU) with 16 vector lanes, and based on the targeted application, different Systolic Arrays (SA) designs.

Each VAS Tile includes a distributed L2 data cache of 4MB in size with 16-ways and 16 banks. This L2 can be reconfigured partially to function as a scratchpad depending on the type of workload. Through a NoC, the VAS Tiles are able to communicate with the Memory Tiles to facilitate memory operations decoupling access and execute operations.

Each Memory Tile consists of a Memory CPU (MCPU), which organizes and dispatches memory requests originating in the VAS Tiles to the associated Memory Controller (MC) and its attached slice of the overall High Bandwidth Memory (HBM). The memory address space is mapped across 16 MCs. Therefore, requests that are out of the address range of the current MC, need to be forwarded to the Memory Tiles and MCs that own that address range. The current implementation of ACME integrates HBM due to the limitations of the MEEP FPGA platform. However, non-volatile memories (NVRAM) technologies will be considered in the future [33], when available.

ACME is equipped with four auxiliary CPUs, which are similar to the MCPUs. Those CPUs establish the connectivity to the external components and also run the OS with its daemons and services which are usually provided by the host CPU. Therefore, ACME becomes self-sufficient, orchestrating its own internal resources and hence it is a self-hosted accelerator. By enabling this capability in ACME, the application data can be stored in a single location and does not have to be moved between the host CPU and accelerator, saving significant cost and energy. In this scenario, we envision the host CPU only having reduced capabilities to provide storage and naming services to supply data and organize communication.

3.2. Memory Tile

Each Memory Tile is equipped with a Memory Controller (MC), a slice of HBM, a Jumbo Translation Lookaside Buffer (JTLB) for address translation and the MCPU as the central element of the Tile. In order to minimize hardware overhead and benefit from shared structures and resources, the MCPU is a fine-grain multithreaded core, a standard RISC-V RV64IMAC core without support for floating-point operations, that is biased to manage memory requests and related operations. There is a one-to-one mapping of hardware threads in the MCPU and hardware threads in the VAS tile. The bonded threads form the decoupled access/execute architecture in ACME. Additional MCPU architecture details will be determined by simulation [19].

3.2.1. MCPU memory access orchestration

The MCPU maintains a queue of outstanding memory requests for its associated memory controller. It is able to reorder those outstanding requests based on the DRAM page to be opened

and the age of the request, avoiding starvation which may produce exceptionally long latencies. These requests are enriched with meta-data indicating the destination of the data in the VAS Tile such as the cache, scratchpad or directly into the register file. Based on the access patterns from other threads and cores, there may be multiple recipients for a single memory request that the MCPU must track. In some cases, based on the spatial and temporal nature and number of recipients for a particular set of addresses, it may make sense to store the entire DRAM page in an L3 Row Buffer (RB in Fig. 4). This effectively creates a virtual DRAM open page, increasing the DRAM interleaving and memory bandwidth, while reducing the latency.

Non-unit stride memory operations to scatter or gather single vector elements from specified indices result in non-uniform access patterns which are typically performance limiting. In order to optimize this case, the MCPU implements vector index registers to make the scatter and gather operations more efficient and high performance. The goal is to provide unit stride data structures to the VAS Tiles even for sparse data structures, dramatically improving the bit efficiency by removing the parasitic bits that exist in traditional cache structures. Every byte in the vector will be consumed by the VAS Tile, whereas in a traditional cache, the worst and sometimes very common case is that only one word per cache line is consumed by the computation, although other words are moved and stored throughout the on-chip memory hierarchy vs being pruned at the memory controller. The MCPU supports this capability to only move around useful bits, saving energy and improving performance.

The MCPU enables more advanced memory request reordering, caching DRAM pages to improve access timings for multiple accesses, and coalescing non-unit stride data accesses. Moreover, further optimizations are possible based on the ability to specify and retain application data structure information like the overall vector length. Thus, the MCPU can more deeply understand how to manage the memory requests because a variety of program information is known without requiring prediction. For example, the ACME architecture can take advantage of large memory windows by providing vector memory access patterns before the application requires the data. With this knowledge, ACME can provide better memory orchestration, increasing effective memory bandwidth, and reducing unnecessary data movement on chip, especially for non-unit stride access patterns. Thus, ACME shifts the energy usage in the system based on the type of application. For dense applications that are compute bound, the MCPU utilization is very low, enabling most of the system energy to be directed to the VAS Tiles and associated accelerators. However, for sparse workloads, the MCPU utilization is high as it optimizes the memory hierarchy, shifting energy to the MCPUs and away from the VAS Tiles.

3.3. Vector and Systolic (VAS) Accelerator Tile

ACME is targeting both dense and sparse workloads which stress different aspects of the architecture. As a result, ACME adapts the number of active vector lanes per core depending on the arithmetic intensity. The VAS tile is composed of several scalar cores. Each scalar core decodes and dispatches instructions to the tightly coupled functional units and loosely-coupled coprocessor accelerators. Thus, the scalar core issues and commits all instructions as a single instruction stream, but the actual instruction execution depends on its type. Scalar instructions are executed on the scalar core, vector instructions are executed on the vector coprocessor or VPU, systolic array instructions are executed in the SA and memory operations are handled by the Memory Tile.

The VPU is connected through an Open Vector Interface (OVI) [22] to the scalar core. Instructions are forwarded, along with cache lines, over this interface to the VPU. This interface is sufficient for applications with unit stride access patterns. For applications with data that is not laid out in memory in unit stride, this interface suffers from significant performance and energy degradation, transporting useless data from the DRAM, through the cache hierarchy to the VPU and is then discarded. We call these bytes, *parasitic bytes*, consuming both energy and performance. Thus, we extend the VPU to also have its own memory interface to a shared L2 scratchpad, where data is stored in a dense representation as unit stride vectors. The hardware-managed scratchpad acts as a second level register file, enabling dynamic or virtual vector lengths with streaming support to the accelerators. By using vector length agnostic programming techniques, the application can convey application data structure details to the architecture. Not only is all data in the scratchpad useful, ACME also exterminates the *parasitic bytes*.

In addition, the SAs use the same OVI interface to the scalar core and memory interface to the scratchpad, enabling a wide variety of SA implementations, from image and video processing to neural networks. Historically, SAs are set up as memory mapped accelerators with on-chip memory regions that stream the data through the arrays with mailboxes to coordinate execution. OSH enables new instructions to be defined, leveraging the software toolchain to manage resources more effectively. Thus, we can use traditional memory and compute operations defined as custom RISC-V instructions to control and orchestrate these accelerators. Finally, the SAs share a common infrastructure even though they are computationally different. We implement a common SA shell that provides the coprocessor interface, as well as a memory interface to the L2 scratchpad to support multiple SA variants.

Each core in the VAS Tile supports up to 8 threads to hide memory latency for sparse workloads. A single-threaded mode is also supported to target dense computations using all the computational resources. This applies to kernels like GEMM or the computations performed by the SAs. The multithreaded mode targets sparse workloads that are memory-bandwidth bound to balance the memory bandwidth with the computational intensity (number of vector lanes). More threads each bound to a fusion of vector lanes, are supported to hide the memory latency. The smallest unit of vector compute is two vector lanes or a vector-lane pair. Within the VPU, multiple vector-lane pairs can be fused together, with up to 16 lanes (most useful for single-threaded dense applications). We believe this combination of multithreading and long virtual vectors enables a very large memory window to maximize HBM utilization and efficiency. In this case, we have selected a coarse-grain multithreading approach that enables the scalar cores to coordinate all the work required to support the various coprocessors. This configuration provides a lot more flexibility with regard to memory scheduling and opportunities for spatial and temporal locality versus a very long Virtual Vector Length (VVL) alone. Ideally, this also leads to a better dynamic load balance in the system.

3.4. Example SAXPY

In this section, we describe how the various components of ACME interact. For clarity, we focus on only one scalar RISC-V core in the VAS Tile and one Memory Tile as shown in Fig. 4.

For this example, we use a dense vector length agnostic SAXPY kernel example to explain the execution model. Vector instructions are forwarded to the VPU from the scalar core using the OVI, while vector memory operations are directed to the Memory Tile by the scalar core. A VAS Tile always forwards the memory operation to the same Memory Tile, to the bonded MCPU

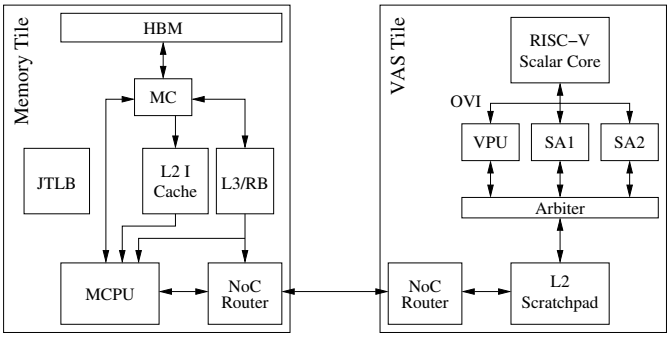


Figure 4. A simplified ACME used in the example

hardware thread. Hence, the MCPU in that tile is aware of the state of the program executed on the scalar core.

The application to be executed is $Z = aX + Y$, a function from the standard Basic Linear Algebra Subprograms (BLAS) library [3]. There are two input vectors in listing 5, X and Y, with n single precision (32 bit) floating point values each and a scalar value a. In the listing, the application vector length (AVL), n, is stored in register `a0`, the value of a is stored in `fa0`, while the elements for X and Y are located at the addresses stored in registers `a1` and `a2`, respectively.

First, a `vsetvli` instruction is executed by the scalar core and forwarded to the MCPU as a transaction. It contains the AVL from `a0`, the type of the elements (`e32`), the vector register grouping (`m8`) and the desired Virtual Vector Length (VVL). The MCPU processes the request and sets the VVL for all subsequent memory operations (until the next `vsetvli`) and acknowledges the transaction. Note, the novelty of ACME is that the VVL can be larger than the Physical Vector Length (PVL) that the VPU's 1st level Vector Register File (VRF) supports. ACME implements a dual purpose L2 memory structure that can operate as a cache and/or scratchpad. To accommodate the entire VVL of the vector load instruction (`vle`) in line 3, dedicated scratchpad memory space needs to be reserved and associated with `v0`.

As an example, consider the AVL to have 10^6 elements, while for single precision values PVL = 4. In this scenario let VVL = 16. Therefore, the vector load returns a vector containing 16 elements to be stored in the scratchpad and the opportunity to preload the first 4 elements into the VRF as PVL.

The next instruction is a vector load instruction (line 3). It instructs the scalar core, to start filling vector register `v0` with elements starting from address in `a1`. A transaction containing this instruction is forwarded to the MCPU, which starts collecting the elements from the indicated address. In case of an ACME having multiple Memory Tiles, and if the address is not in the range of the attached physical memory, transactions are created with the addresses of the missing elements and forwarded to the Memory Tiles that serve the requested addresses.

Based on the vector lengths computed earlier, the MCPU starts returning VVL elements. A scoreboard mechanism inside the VAS Tile keeps track of the data arrival and stores it into the vector registers and allocated space in the scratchpad.

Since this program is a dense workload, this load instruction represents a loading pattern with unit strides. Sparse workloads are indicated by non-unit stride or gather/scatter memory operations. In case of a load instruction, the MCPU collects the required elements and coalesces them into a dense vector. The opposite is done with stores, sending data from the scratchpad to memory. Therefore, NoC bandwidth and memory storage is preserved by avoiding handling elements, which are discarded eventually.

```
1  saxpy:
2      vsetvli a4, a0, e32, m8      ; determine VVL from AVL
3      vle32.v v0, (a1)             ; load vector
4      sub a0, a0, a4
5      slli a4, a4, 2
6      add a1, a1, a4
7      vle32.v v8, (a2)
8      vfmacc.vf v8, fa0, v0        ; Z = aX + Y
9      vse32.v v8, (a2)             ; store vector
10     add a2, a2, a4
11     bnez a0, saxpy               ; jump to next loop iteration
12     ret
```

Figure 5. The assembly source code for a RISC-V with vector instructions from the vector extension in bold

The next three instructions in Fig. 5 in lines 4–6 are scalar instructions executed on the scalar core itself, while the following vector load in line 7 is similar to the one in line 3.

The FMAC vector instruction (line 8) is forwarded to the VPU together with the scalar value in register `fa0`. The result of that operation is stored back into `v8`. This requires a coordinated streaming of input and output data for the operands in the scratchpad, and in this case a shared source and destination register.

Once the entire VVL has been computed, the `vfmacc` instruction is retired. Since a store instruction follows, a transaction with the data in the scratchpad region associated with `v8` is issued to the Memory Tile, which then moves the data back into the HBM.

This concludes one iteration of the loop, in which a part of the entire application vector has been computed. In case of AVL > VVL multiple iterations of the loop are executed. Since the MCPU is aware of AVL as well as VVL, due to the `vsetvli` instruction initially sent to the MCPU, multiple repeating memory operations with different offsets are expected. Therefore, while the VPU computes the results for the current iteration, the MCPU is able to preload the next vector with VVL elements and temporarily store it in its L3 RB to have it ready as soon as the first load instruction of the next iteration arrives at the Memory Tile. Hence computation and memory access are parallelized.

ACME offers many opportunities to further optimize the architecture. For instance, a scaled down version of the program executing on the scalar core may be executed on the MCPU similar to [11] or [25]. However, this is part of the future work and exploration.

3.5. Simulating ACME

The development of new architectures involves high level decisions that determine the general flavor of the design. These decisions, which include details such as core counts or the overall cache hierarchy, should be made early, via simulation, so later FPGA-based analysis can focus on what it should: lower level details. In the case of ACME, more aggressive design points require early investigation before implementation, like the proposed novel data handling schemes of the MCPU and its automatically managed scratchpad. However, simulation of HPC-capable architectures is still time-consuming, due to the size of the workloads and number of cores that need to be simulated in order to fully exercise the modelled system. This is at odds with the very purpose of

early simulation, which is performing large amounts of executions to identify certain parameters of the system and evaluate new ideas.

For the reasons above, MEEP proposes Coyote [19], a new execution-driven simulator for the RISC-V ISA. Coyote strikes a balance between simulation throughput and fidelity by focusing on the modelling of the data movement throughout the memory hierarchy. This captures the right amount of detail to enable the comparison of different designs while preventing simulation times from growing beyond the limits of reasonable design space exploration. To leverage previous community efforts, Coyote is based on two pre-existing simulators: Spike [21] and Sparta [23]. Spike, as the open-source golden-standard for RISC-V, provides the capabilities for functional simulation, including the vector extension and also the modelling of L1 caches. Additionally, it has been extended to support other features such as RAW dependency tracking and different instruction latencies. Sparta, a framework to build event-driven simulators developed by SiFive, sits on top of Spike to provide the modelling capabilities for the memory hierarchy of the system, which is exercised by events generated by the execution of simulated applications in Spike.

In order to reduce the overhead of the interaction between the two pieces that build Coyote, they have been integrated as slaves into a *simulation orchestrator*. It is in charge of managing the communication of events between the functional and the event-driven side of Coyote and keeping the clocks synchronized. In every cycle, an instruction is simulated in each of the simulated active cores. The simulation of an instruction might have different effects:

- A RAW dependency might be detected in the registers that are read. In this case, the core is marked as inactive. It will not be able to execute again until after the dependency is satisfied.
- An event that needs to be communicated to the Sparta event-driven engine might be generated. This includes requests to the L2, interactions with the MCPU and more.

After the simulation of instructions, events are submitted to Sparta. Then, the Sparta clock is advanced and events are handled. Submitting an event to Sparta will internally generate a chain of events that will correctly exercise the modelled memory hierarchy and interact with the MCPUs. The end result of this chain of events is usually an interaction back to the simulation orchestrator, that will update certain values in Spike. An example of this is marking a RAW dependency as satisfied when a memory access submitted to the L2 is satisfied. The handling of the different types of events has been implemented following the visitor design pattern, for easy maintainability and extensibility.

Coyote models tiled architectures similar to ACME. This includes core-private L1s and a slice of banked L2 per tile. The L2 can be configured as private to the cores that belong to a tile or fully shared across the system, forming a NUCA. A basic memory controller, which implements a subset of memory commands and different address mapping policies. The NoC connecting the tiles and memory controllers is modelled implementing three different levels of accuracy, ranging from a very simple model based on the average number of cycles to travel through the network to a detailed one that is based on the well-known Booksim simulator [13] (integrated also as a slave to the simulation orchestrator). Many of the parameters regarding the sizing and timing of each of the components can be configured through parameters in order to evaluate the behavior of different configurations.

With respect to applications, Coyote executes baremetal applications compiled using either the standard GNU compiler for RISC-V or the EPI compiler, if vector intrinsics are used. As a result, it produces general statistics related the use of the memory hierarchy, such as the miss rate

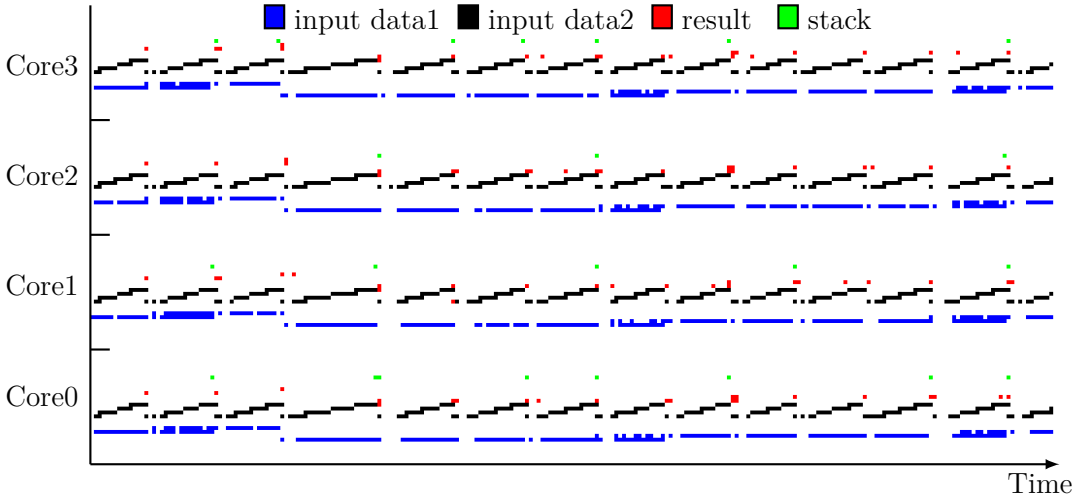


Figure 6. Paraver L2 miss trace for a 4 core matrix multiplication. Different accessed data structures have different colors

M3	V3	V7	V11	V15	M7
M2	V2	V6	V10	V14	M6
M1	V1	V5	V9	V13	M5
M0	V0	V4	V8	V12	M4

(a) Fully shared cache forming a NUCA

M3	V3	V7	V11	V15	M7
M2	V2	V6	V10	V14	M6
M1	V1	V5	V9	V13	M5
M0	V0	V4	V8	V12	M4

(b) Tile with private L2 caches

Figure 7. Heatmap (the lighter the shade, the sparser the traffic) for the destination of packets (M – Memory Tile, V – VAS Tile) in a matrix multiplication

per L2 slice, the number of stalls in the L2 due to MSHR exhaustion or the amount of requests serviced by each memory bank. Coyote can also produce a trace that can be visualized using paraver [20] to identify patterns and analyze the behavior of applications in detail. As an example, Fig. 6 shows a portion of an L2 miss trace for a vector matrix multiplication using 4 simulated cores. Misses for different data structures have different colors, which eases the behavioral analysis and and helps to identify patterns. Regarding the NoC, Coyote also produces a heat map to easily identify hotspots, which are potentially related to inefficient data management. Figure 7 compares two heatmaps for a vector matrix multiplication simulated on a 16 VAS Tile and 8 Memory Tile configuration. Memory Tiles are placed in the first and last columns, for a scaled down version of the proposed ACME layout shown in Fig. 2. Both heatmaps are for the execution of the exact same application and only differ regarding how the L2 is shared, either forming a NUCA across all the tiles (Fig. 7a) or private to each tile (Fig. 7b). This figure shows that the NUCA generates a lot more traffic indicated by the darker shades, among the VAS tiles, while relieving the memory tiles (lighter shades).

As a result of its modelling capabilities and its overall design, which strikes a balance between fidelity and simulation throughput, Coyote can provide insight into how high level design decisions

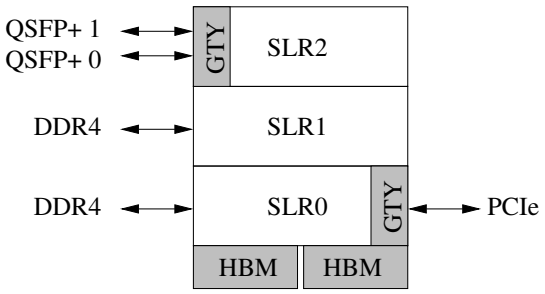


Figure 8. The Xilinx Virtex Ultrascale+ VU37P with HBM combines three Super Logic Regions (SLR) and HBM to a System-in-Package (SiP). The shaded blocks are part of the MEEP Shell

impact the performance of HPC architectures. This enables the evaluation of different designs prior to more costly FPGA implementation and emulation.

4. FPGA

FPGAs enable MEEP to provide silicon validation capabilities. MEEP consists of 12 nodes comprising 8 Xilinx Virtex Ultrascale+ FPGAs [31], each able to establish a dense interconnect among the FPGAs. Those FPGAs exhibit up to 500 Mb of total on-chip integrated memory, in addition to up to 16 GB of HBM. With these characteristics, MEEP is a foundation for hardware emulation and SDV for other projects as eProcessor [8], and can be exploited in projects such as DRAC [15] or OpenPiton [7].

Xilinx Virtex Ultrascale+ FPGAs are built on top of a silicon interposer, connecting three Super Logic Regions (SLRs) together in the same package (System in Package, SiP). The connections among the SLRs are silicon based vias labeled as Super Logic Lines. This technology allows the FPGA part to exceed traditional sizes. They may be thought as three smaller FPGAs fused together in the same silicon forming a new larger device (refer to Fig. 8).

These FPGAs contain up to 1.3M LUTs, 260k Flip-Flops (FF), 2k BRAMs (36 Kbits) and 9k DSPs, among other common elements. In addition, the FPGAs contain integrated hard IPs such as PCIE4C (PCIe Gen4), CMAC (100 Gbit Ethernet), and memory controllers for the HBM and DDR RAM. A common collection of I/O and memory interfaces are implemented for these building blocks, which facilitates the use of MEEP as a silicon validation platform. These interfaces define the MEEP FPGA Shell, which is the foundation for an engine to enable the SDV and pre-silicon validation cycle.

4.1. FPGA Shell

The FPGA implementation is composed of two main components: the MEEP FPGA Shell and the emulation region for accelerators and/or CPUs. The MEEP FPGA Shell is a static, customizable perimeter architecture which guarantees that the emulation region remains portal/interchangeable across any other FPGA package or device that meets the defined I/O and memory interface specifications. The MEEP FPGA Shell goes beyond Xilinx' approach, which consists of multiple proprietary platforms, which may not meet the demands of the accelerator. Instead the MEEP FPGA Shell is fully customizable by the user to match a given FPGA architecture. On top of that, the MEEP Shell provides QDMA, FPGA-to-FPGA and Ethernet solutions, which Xilinx platforms lack. One example for an accelerator is ACME (refer to Section 3), but others can be considered as well as shown in Section 4.2.

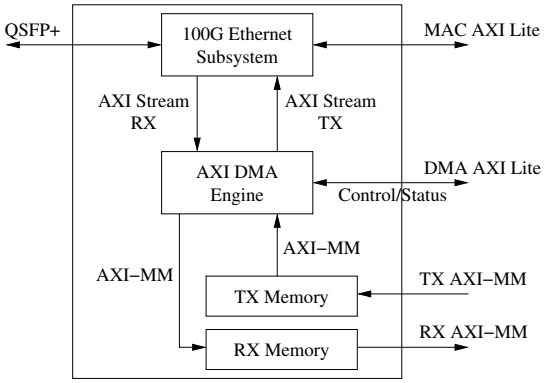


Figure 9. The MEEP FPGA Shell Ethernet IP. It can be used as long as the accelerator is compliant with the specified interfaces

The basic components of the MEEP FPGA Shell are PCIe, HBM, 100 Gb Ethernet, and Aurora for raw FPGA-to-FPGA communication. A DDR4 controller can also be included, which is used to simulate an envisioned Non-Volatile Memory (NVRAM), which is part of a future system beyond the scope of this project. The Shell is generated through a TCL-based script which lets the user create a custom MEEP FPGA Shell configuration, followed by the actual building process, parsing the emulated accelerator and checking that all the desired connections between the MEEP FPGA Shell and the accelerator engine are instantiated.

For the experiments, the Alveo U280 featuring the Xilinx Ultrascale+ VU37P FPGA [30] will be used to implement the MEEP FPGA Shell as well as emulate the ACME accelerator.

4.1.1. PCIe interface

As part of the MEEP FPGA Shell, PCIe implements the communication interface between the host and the emulated hardware. PCIe is already included as a hard IP in the VU37P FPGA, resulting in a reduction of resource requirements when compared to pure soft-logic PCIe implementations. Xilinx supports two different types of PCIe blocks, XDMA and QDMA. The Xilinx platforms or Amazon F1 utilize the XDMA block. However, MEEP FPGA Shell focuses its efforts on the QDMA implementation for flexibility and streaming capabilities.

The main difference between QDMA and other DMA offerings is the concept of queues, derived from the *queue set* concepts of Remote Direct Memory Access (RDMA) from high-performance computing (HPC) interconnects. These queues can be individually configured by the interface type. Based on how the DMA descriptors are loaded for a single queue, each queue provides a very low overhead option for continuous update functionality. The QDMA solution supports multiple Physical/Virtual Functions with scalable queues, and is ideal for applications that require small packet performance at low latency, as well as streaming data. QDMA offers more functionality at slightly higher resource costs with two advantages: First, it implements a high performance configurable scatter/gather DMA for the PCIe IP. Second, the IP provides an AXI4-Stream user interface in addition to AXI Memory Mapped and AXI-Lite. QDMA can be set to PCIe Gen3 or Gen4. In both cases, the peak bandwidth is 16 GB/s in each direction.

4.1.2. HBM interface

The HBM is the high performance DRAM interface that massively increases system memory bandwidth using SiP technology [27]. HBM enables maximum bandwidth, efficient routing and

Table 1. The clock frequencies and LUT utilization for various MEEP Shell IPs

Shell IP	Frequency [MHz]	User Clock	Resources [LUT]
PCIe (QDMA)	250	Fixed	70376
HBM	≤ 450	Maximum	1539
Ethernet	322.26	Fixed	7444
Aurora	≤ 402.23	Maximum	1500
Aurora (DMA Mode)			4849
DDR4	300	Fixed	18823

logic utilization, and optimizes power efficiency for workloads that process large datasets. The MEEP FPGA Shell provides a standard interface to HBM. An HBM stack of four DRAM dies has two 128 bit channels per die for a total of 8 channels and a width of 1024 bits [29]. In comparison, the bus width of GDDR memories is 32 bits, with 16 channels and a 512 bit wide memory interface. Memory bandwidth for Xilinx Virtex Ultrascale+ FPGAs with HBM is up to 460 GB/s delivered by two stacks of Samsung HBM2 with 8GB each. There is the flexibility to access pairs of pseudo channels or aggregate all the pseudo channels across two stacks of HBM together as a large memory access engine with much larger memory bandwidth and larger granularity. In the context of the accelerator, HBM related logic can be clocked at up to 450 MHz.

4.1.3. Aurora interface

The MEEP FPGA Shell can be configured to use of GTY transceivers [28] for point-to-point FPGA communication. Xilinx' Aurora IP wraps up the transceivers and encodes a 64b/66b protocol to provide enough state changes to allow a reasonable clock recovery and alignment of the data stream at the receiver. This allows easy adoption of custom protocols inside the accelerator which only needs to encode its own protocol in a 64 bit wide bus and assert a valid signal once the data is ready. The accelerator can also rely on a DMA based solution. In this scenario, the accelerator needs to configure the DMA to program transactions from the memory. This communication interface is full-duplex, so the same principles apply for the receiver (RX) and transmitter (TX). On the RX side, the implementation must be designed carefully though, as there is no back-pressure capability and it must be ready to consume any incoming data. In the context of MEEP, the user logic associated to the Aurora interface has been validated working at frequencies of up to 402.832 MHz.

Xilinx transceivers are composed of four lanes in a configuration named *Quad* sharing a common reference clock. With this configuration, each of the four lanes can be used to create different topologies such as torus, hypercube, etc.

4.1.4. 100 Gb Ethernet interface

The MEEP FPGA Shell also includes a 100 Gb Non-Return-to-Zero (NRZ) Ethernet solution shown in Fig. 9. It is built around the hard IP CMAC [32] and is present in the FPGA. This solution creates a ready-made interface to grant Ethernet functionality between the accelerator and any other device in the network, including an Ethernet switch. The accelerator only needs to

be compliant with several AXI interfaces in order to set-up, control and make use of the Ethernet IP. The associated Ethernet user logic is clocked at 322.266 MHz.

4.1.5. DDR4 interface

DDR4 is offered in the MEEP FPGA Shell either as main memory or it can be used to emulate NVRAM or other hybrid memory architectures with an order of magnitude higher effective capacity compared to HBM [12]. Current technology couples HBM with NVRAM as demonstrated by Intel [6, 33]. The accelerator can leverage this MEEP FPGA Shell feature to emulate such systems.

4.1.6. MEEP FPGA Shell details

In Tab. 1, a summary of the clocks associated with each of the MEEP FPGA Shell IPs is shown. Depending on the design, the maximum frequencies allowed for the HBM and Aurora may not be achieved by the other IPs, forcing the HBM and Aurora to be clocked down. The frequencies for the PCIe, Ethernet, and DDR4 are fixed by the specifications and cannot be configured by the user. As shown in the table, because the MEEP FPGA Shell leverages hard IPs, it only requires approximately 100k LUTs in total, which amounts to 8% of the overall resources available in the VU37P FPGA. The remaining resources can be used for the accelerator implementation.

4.2. Other Accelerator Examples Mapped to MEEP

The path to a fully functional ACME accelerator emulated in MEEP is in the near future. However, there are several intermediate milestones that demonstrate the utility and capabilities of MEEP. We present two different RISC-V IPs embedded in a user configurable MEEP FPGA Shell. Due to the standardized interfaces, the integration reduces time for implementation and validation, while the use of hard IP minimize the resource requirements of the MEEP FPGA Shell. Thus, the MEEP FPGA Shell can be used as a foundation to test multiple accelerators, processors or other IPs with varying configurations. The projects such as OpenPiton [7] and DRAC [15] are both aligned with the concept of the accelerator. OpenPiton is scalable in terms of cores. On the other hand, DRAC is a design which implements the scalar core of Lagarto and a VPU with two lanes. Both have been packaged as two different accelerators using the MEEP FPGA Shell as a demonstrator for the combination of the Shell and the accelerator. These projects serve two purposes: enabling a general MEEP FPGA Shell that is reusable across multiple projects and exploring a baseline IP to extend for ACME.

Embedding the accelerator into the MEEP FPGA Shell is straightforward, since only the interfaces the Shell offers have to match (refer to Fig. 9), avoiding any other configuration overhead.

In both cases, the MEEP FPGA Shell has been configured by the user to contain a PCIe QDMA and HBM without Aurora or Ethernet connectivity. The resource utilization in this configuration is 5% of the total FPGA resources (compared to Tab. 1). Likewise, OpenPiton has been mapped into the MEEP FPGA Shell as a many-core system consisting of four Ariane RISC-V cores (RV64GC). Fig. 10 reveals that this OpenPiton configuration requires 28% of the overall LUTs available on the FPGA. In comparison, DRAC with its Lagarto RISC-V core, including its 2-lane VPU requires 18% of the FPGA.

The use of the Shell significantly reduces implementation time during the accelerator development, since all Shell features are supported out of the box, with a proven validated set of

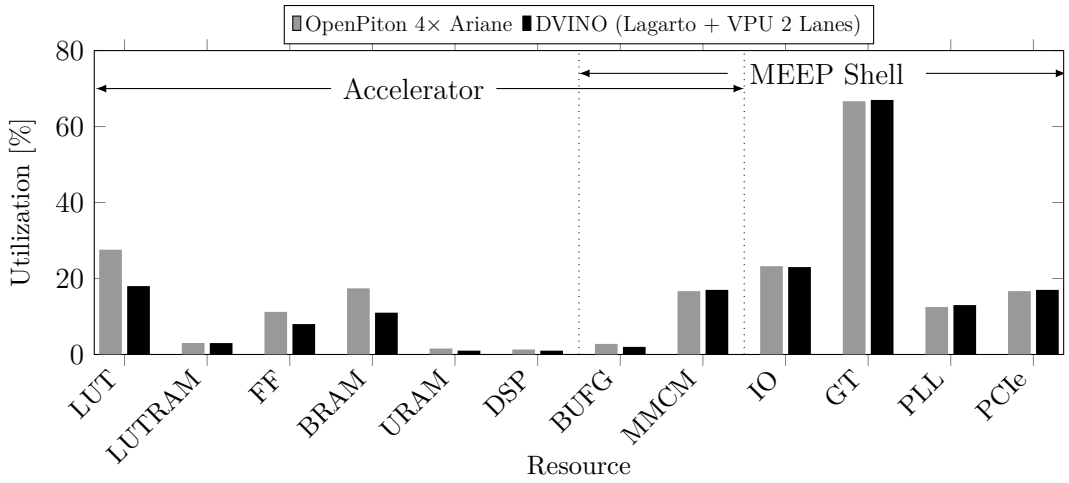


Figure 10. Utilization report for OpenPiton in a many-core (4 RV64GC Ariane cores) and DRAC (Lagarto and a 2-lanes VPU coprocessor)

interfaces. Connecting OpenPiton and DRAC to the Shell worked as expected the first time they have been connected, demonstrating one of the main goals of MEEP, which is to provide a solid hardware environment for software development and silicon validation.

As it is evident from Fig. 10, the selected accelerator does not have any impact on the I/O, GT, PLL, nor PCIe resource requirements, while BUFG and MMCM are affected marginally. The resources depicted on the right side in the figure belong to the MEEP FPGA Shell, which is the same for both accelerators. Therefore, the limiting factor in embedding a different accelerator is the availability of FPGA resources mentioned on the left side in the figure, mainly the LUTs and BRAM.

5. Related Work

Several platforms targeting simulation accelerators and small-scale SDVs precede MEEP. However, they are either closed source or do not support OSH and the related ecosystem. The PROTOFLEX [9] project is an FPGA-based architecture to accelerate full-system multiprocessor simulation and to facilitate high-performance instrumentation. It is not a specific simulator itself, but a template that offers a set of practical approaches for developing FPGA-accelerated simulators. It virtualizes the execution of many logical processors onto a consolidated number of multiple-context execution engines in the FPGA, which can be scaled, as needed, to deliver the necessary simulation performance at large savings in complexity.

Another example is DIABLO [26], a cost-efficient FPGA-based emulation methodology, which treats FPGAs as whole computers with tightly integrated hardware and software. DIABLO is not based on FPGA prototyping, but uses FPGAs to accelerate parameterized abstract performance models of the system instead. DIABLO is fully parameterizable and fully instrumented, and supports repeatable deterministic experiments. Most of the efforts on its model are focused on networking analysis and optimizations.

The MANGO [10] project is an FPGA-based many-core emulation platform. It facilitates the exploration of heterogeneous accelerators for being used in HPC systems running multiple applications with different Quality of Service (QoS) levels. To make this possible, MANGO has explored different but interrelated mechanisms across the architecture and system software. The

platform provides a large-scale cluster of multiple FPGA boards intended for experimenting with customized many-core systems, at the level of both processor and interconnect/system architecture, along with the supporting software stack.

Mont-Blanc [2] was a predecessor of the European Processing Initiative (EPI) [4] relying on the results of three consecutive projects (i.e., Mont-Blanc 1, 2, and 3), which explored the viability of using highly energy efficient ARM-based processors for HPC. The main objective of Mont-Blanc 2020 was to start developing building blocks (IPs) for an HPC processor.

Conclusion and Future Work

A large community is required to develop the OSS and OSH ecosystem. We must build tools and systems to enable independent development and exploration of software and hardware; one cannot exist without the other. MEEP is a digital laboratory that enables RISC-V ecosystem development for the HPC and HPDA domains based on a large-scale platform based on composable and reuseable IP blocks. In order to rapidly move forward in the RISC-V ecosystem, we need tools in the community to support software and hardware development. MEEP provides the former as a large-scale SDV and the latter as a pre-silicon hardware emulation platform. Combined, MEEP can realize a future vision HPC and HPDA accelerators in the form of the ACME architecture and demonstrate this full stack.

We have implemented the first phase of the MEEP platform using four Xilinx Alveo U280 [30] cards and have focused on the base infrastructure for the MEEP FPGA Shell, and a variety of initial OSH cores, and multiprocessor SoCs. We see a bright future for the RISC-V ecosystem in HPC and HPDA and plan to continue to develop and contribute using MEEP.

Acknowledgments

This work has been supported by the EU H2020 project MareNostrum Experimental Exascale Platform (MEEP), and funded by the European Commission under the grant agreement No. 946002.

References

1. Fact Sheet & Background: Roadrunner Smashes the Petaflop Barrier (2008), `http://www-03.ibm.com/press/us/en/pressrelease/24405.wss`

2. The Mont-Blanc Project (2020), `https://www.montblanc-project.eu/`

3. Basic Linear Algebra Subprograms (2021), `http://www.netlib.org/blas/`

4. The European Processor Initiative (2021), `https://www.european-processor-initiative.eu/`

5. Abadi, M., Agarwal, A., Barham, P., et al.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), `https://www.tensorflow.org/`

6. Altman, A., Arafa, M., Balasubramanian, K., et al.: Intel Optane Data Center Persistent Memory. In: 2019 IEEE Hot Chips 31 Symposium (HCS), 18-20 Aug. 2019, Cupertino, CA, USA. pp. i–xxv. IEEE (2019), DOI: 10.1109/HOTCHIPS.2019.8875668

7. Balkind, J., McKeown, M., Fu, Y., et al.: OpenPiton: an open source hardware platform for your research. Commun. ACM 62(12), 79–87 (2019), DOI: 10.1145/3366343

8. BSC: eProcessor: European, extendable, energy-efficient, energetic, embedded, extensible, Processor Ecosystem (2021), https://www.bsc.es/research-and-development/projects/eprocessor-european-extendable-energy-efficient-energetic-embedded

9. Chung, E.S., Nurvitadhi, E., Hoe, J.C., Falsafi, B., Mai, K.: PROToFLEX: FPGA-accelerated Hybrid Functional Simulator. In: 2007 IEEE International Parallel and Distributed Processing Symposium, 26-30 March 2007, Long Beach, CA, USA. pp. 1–6. IEEE (2007), DOI: 10.1109/IPDPS.2007.370516

10. Flich, J.: MANGO: Exploring Manycore Architectures for Next-GeneratiOn HPC Systems. In: 2017 Euromicro Conference on Digital System Design (DSD), 30 Aug.-1 Sept. 2017, Vienna, Austria. pp. 478–485. IEEE (2017), DOI: 10.1109/DSD.2017.51

11. Hashemi, M., Mutlu, O., Patt, Y.N.: Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 15-19 Oct. 2016, Taipei, Taiwan. pp. 61:1–61:12. IEEE Computer Society (2016), DOI: 10.1109/MICRO.2016.7783764

12. Izraelevitz, J., Yang, J., Zhang, L., et al.: Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. CoRR abs/1903.05714 (2019), http://arxiv.org/abs/1903.05714

13. Jiang, N., Becker, D.U., Michelogiannakis, G., et al.: A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 21-23 April 2013, Austin, TX, USA. pp. 86–96. IEEE (2013), DOI: 10.1109/ISPASS.2013.6557149

14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization, 20-24 March 2004, San Jose, CA, USA. pp. 75–86. IEEE (2004), DOI: 10.1109/CGO.2004.1281665

15. Leyva-Santes, N.I., Perez, I., Hernández-Calderón, C.A., et al.: Lagarto I RISC-V Multi-core: Research Challenges to Build and Integrate a Network-on-Chip. In: Supercomputing, 25-29 March 2019, Monterrey, Mexico,. pp. 237–248. Springer, Cham (2019), DOI: 10.1007/978-3-030-38043-4_20

16. Lordan, F., Tejedor, E., Ejarque, J., et al.: ServiceSs: An Interoperable Programming Framework for the Cloud. Journal of Grid Computing 12, 1–25 (2013), DOI: 10.1007/s10723-013-9272-5

17. Message Passing Interface Forum: MPI: A message-passing interface standard. Version 3.1 (2015), https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, accessed: 2021-04-23

18. OpenMP Architecture Review Board: OpenMP Application Programming Interface 5.1 (2020), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, accessed: 2021-04-23

19. Perez, B., Fell, A., Davis, J.D.: Coyote: An Open Source Simulation Tool to Enable RISC-V in HPC. In: Design, Automation, and Test in Europe, DATE (2021)

20. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A Tool to visualize and analyze parallel Code. WoTUG-18 44 (1995)

21. RISCV: The Spike RISC-V ISA Simulator (2021), https://github.com/riscv/riscv-isa-sim

22. Semidynamics: Open Vector Interface (2021), https://github.com/semidynamics/OpenVectorInterface

23. Si-Five: The Sparta Framework (2021), https://github.com/sparcians/map/tree/master/sparta

24. Smith, J.E.: Decoupled Access/Execute Computer Architectures. ACM Trans. of Computer Sys. 2(4), 289 (1984)

25. Srinivasan, V., Chowdhury, R.B.R., Rotenberg, E.: Slipstream Processors Revisited: Exploiting Branch Sets. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 30 May-3 June 2020, Valencia, Spain. pp. 105–117. IEEE (2020), DOI: 10.1109/ISCA45697.2020.00020

26. Tan, Z., Qian, Z., Chen, X., et al.: DIABLO: A Warehouse-Scale Computer Network Simulator Using FPGAs. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, 14-18 March 2015, Istanbul, Turkey. pp. 207–221. ACM, New York, NY, USA (2015), DOI: 10.1145/2694344.2694362

27. Wissolik, M., Zacher, D., Torza, A., Day, B.: Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance (2019)

28. Xilinx: UltraScale Architecture GTY Transceivers (2017)

29. Xilinx: AXI High Bandwidth Memory Controller v1.0 LogiCORE IP Product Guide (2019)

30. Xilinx: Alveo U280 Data Center Accelerator Card (2020), https://www.xilinx.com/products/boards-and-kits/alveo/u280.html

31. Xilinx: Virtex UltraScale+ (2020), https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html

32. Xilinx: UltraScale Devices Integrated 100G Ethernet Subsystem v2.6 (2021)

33. Yang, J., Kim, J., Hoseinzadeh, M., et al.: An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. CoRR abs/1908.03583 (2019), http://arxiv.org/abs/1908.03583

34. Zaharia, M., Xin, R.S., Wendell, P., et al.: Apache Spark: A unified engine for big data processing. Communications of the ACM 59(11), 56–65 (2016), DOI: 10.1145/2934664

DOI: 10.14529/jsfi210106

Micro-Workflows Data Stream Processing Model for Industrial Internet of Things

Ameer B. A. Alaasam[1], ***Gleb I. Radchenko***[1], ***Andrei N. Tchernykh***[1,2,3]

The fog computing paradigm has become prominent in stream processing for IoT systems where cloud computing struggles from high latency challenges. It enables the deployment of computational resources between the edge and cloud layers and helps to resolve constraints, primarily due to the need to react in real-time to state changes, improve the locality of data storage, and overcome external communication channels' limitations. There is an urgent need for tools and platforms to model, implement, manage, and monitor complex fog computing workflows. Traditional scientific workflow management systems (SWMSs) provide modularity and flexibility to design, execute, and monitor complex computational workflows used in smart industry applications. However, they are mainly focused on batch execution of jobs consisting of tightly coupled tasks. Integrating data streams into SWMSs of IoT systems is challenging. We proposed a micro-workflow model to redesign the monolith architecture of workflow systems into a set of smaller and independent workflows that support stream processing. Micro-workflow is an independent data stream processing service that can be deployed on different layers of the fog computing environment. To validate the feasibility and practicability of the micro-workflow refactoring, we provide intensive experimental analysis evaluating the interval between sensor messages, the time interval required to create a message, between sending sensor message and receiving the message in SWMS, including data serialization, network latency, etc. We show that the proposed decoupling support of the independence of implementation, execution, development, maintenance, and cross-platform deployment, where each micro-workflow becomes a standalone computational unit, is a suitable mechanism for IoT stream processing.

Keywords: stream processing, fog computing, cloud computing, scientific workflow, micro-workflow, IoT.

Introduction

The Industrial Internet of Things (IIoT) comprises networked objects, cyber-physical assets, associated information technologies, cloud and edge computing platforms. It enables real-time data processing and exchange of processes, products, and service information within the industrial environment to optimize overall production value [7]. An essential feature of computing services in IIoT are on-site processing, ensuring security requirements, and data pre-processing before sending it to the cloud. Thus, there is a need for an intermediate processing power between industrial IoT and cloud [1]. Fog computing provides such resources by moving some of the data processing tasks from the cloud to fog nodes located closer to the network's edge.

IIoT applications require real-time processing of data streams and signals from multiple sensors (data sources). Event-Driven Architecture (EDA) is the most adapted to this type of applications [3]. EDA is a system architecture made up of highly decoupled, single-purpose event processing components which asynchronously receive and process events [28]. EDA, by its nature, is extremely loosely coupled and highly distributed [12]. A monolithic architecture does not provide the efficiency and flexibility required to support large-scale IIoT systems that require native EDA support and a multilayered fog computing infrastructure.

[1]South Ural State University, Chelyabinsk, Russia
[2]CICESE Research Center Ensenada, Mexico
[3]Ivannikov Institute for System Programming of the RAS, Russia

Due to the complexity in controlling a distributed application workflows, Workflow Management Systems (WFMSs) are often used to assist in the data and task partitioning, providing robust means of describing applications, the control, data dependencies, and the logical reasoning necessary for distributed execution [31]. But a number of challenges appear with WFMSs, for example, the tasks of the workflow are tightly coupled by means of intricate dependencies between them [35]. Also, the features of data visualization and data stream input/output are limited support in current WFMSs [5]. Thus in [2, 27], the concept of Micro-Workflows has been presented. The Micro-Workflow approach supports the redesign of monolith workflows into independent smaller workflows, maintaining stream processing and independent lifecycle management.

This paper presents the micro-workflows model supporting the decoupling process of tightly coupled dependencies in monolith workflow to be refactored in the form of independent smaller micro-workflows, connected via event streaming platform. The Micro-Workflow model can solve a number of problems when using traditional workflow management systems in highly distributed environments, such as fog computing systems to support IIoT. Also, we provide an overview of current IIoT and fog computing challenges in areas such as monolithic application architecture, virtualization, containerization, computational workflows, and data flow management.

The rest of this paper is organized as follows. Section 1 discusses the road from a monolithic into a loosely coupled system architecture, the challenges of fog and cloud computing, the required virtualization infrastructure, and discusses the importance and challenges of computational workflows and data flow management in fog environments. Section 2 provides the proposed model of the micro-workflow architecture. Section 3 provides the experiments and results, while conclusions are provided in Section 3.3.

1. The State of the Art

1.1. From Monolith to Loosely-Coupled Architecture

An event-driven architecture includes sensors and other sources of data; processors that fuse data from multiple sensors and detect patterns over time and deduce events that occurred or predict events; responders for initiating actions in response to events; communication links for transferring information between components; and administrative software for monitoring, tailoring and managing the application [10]. EDA is an extremely loosely coupled system architecture that is made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events [12, 28]. Microservice architecture is considered the most promising in designing loosely coupled systems targeted at event processing today. The microservice approach is based on dividing a computing system into small independent computing services, each implementing its particular aspect of the application's business logic. This approach can overcome many of the significant disadvantages of the so-called monolithic architecture, such as the challenges of distributing the computational load, the presence of a single point of failure, and the challenges of ensuring continuous system operation during maintenance or upgrades while allowing independent development, deployment, scaling and migration of microservices from one computational resource to another [29]. Despite the benefits of this approach, such flexibility does not come for free. For example, communication costs increase due to the need to organize data exchange between microservices, which, in turn, increases the complexity of data flow management and integration of highly distributed components of the

system. Thus, moving to a more loosely coupled design is a multi-objective problem that requires in-depth research to find the best solution to the various issues that arise from decoupling.

1.2. From Cloud to Fog Computing

In highly distributed industrial IoT systems, cloud computing-based solutions become unacceptable due to high latency and network congestion caused by an inability to process the data-flow from the enormous amount of devices in an acceptable time [21]. Thus, Fog computing manages this problem by moving some computational tasks closer to the data sources. Fog Computing is a virtualized platform that provides storage, compute, and networking between end devices and Cloud Computing Data Centers, typically, but not exclusively, located at the edge of the network [6]. Fog nodes can collect, cache, and pre-process data from IIoT sensors before or instead of sending it to the cloud. Another scenario may require certain parts of the system to process data in near-real time. In this case, computing services can be deployed on the nearest fog node to provide a faster response time.

On the other hand, Cloud servers provide significant replication, load balancing, and resilience capabilities. Fog nodes, which are logically decentralized and geographically distributed at network edges, cannot provide this resilience level. For example, finding and resolving a point of failure in a fog computing environment is more complicated than doing that in the cloud [14]. Also, moving toward fog means increasing the computing decoupling, where parts of computing are moved physically to the fog nodes at the edge of the network. Increased component decoupling can lead to difficulties in supporting such systems. For example, the emergence of a large number of independent geographically decoupled components entails significant overhead due to the appearance of a large number of points of failure [18]. The problems of efficient workload allocation, the distribution of computational tasks between edge and cloud resources, and the heterogeneous infrastructure of edge devices need to be solved [8].

1.3. Virtualization Infrastructure

In cloud or fog computing, ensuring resource sharing and dynamic resource provisioning is a fundamental challenge. Virtualization technologies used at various levels (including hardware and application platforms) are used for this purpose. However, large overheads associated with the use of Virtual machines (VMs) can limit the efficiency of computational resources [34]. The challenges associated with using fog-level VMs are even more significant due to the limited resources, processing power, and network traffic at the edge of the network where fog nodes are deployed. This problem can be addressed by containerization technology which allows running containers as separate processes directly on the kernel of hosting OS, providing lightweight isolation for the processes. However, using containerization also facing significant challenges related to the difficulty of containerizing the stateful computational units due to limited data portability support in containers compared to VM [4]. Therefore, if live migration between fog nodes is required, it is a challenge to use container technology without data loss. Thus, finding solutions that reduce the overhead of the virtualization infrastructure, and at the same time ensuring a level of data availability and state recovering, is a very active and vital research area in fog computing.

1.4. The Computational Workflow

The Digital Twin concept combines a control system, real-time simulation, a system for intelligent data analysis, and decision-making when developing industrial processes and systems [26, 32]. This is made possible by data flow and control signals connecting real-world objects and their digital models. Experience in implementing such digital twins shows that the simulation of complex technological processes requires the joint work of a large number of independent computing components, each of which is responsible for implementing its part of the computational process. Such an approach in the field of the smart industry is already implemented using scientific workflow systems. For example, in [20], a specific workflow suit has been developed for product performance degradation assessment and prediction based on Kepler scientific workflow management system (Kepler MS). The authors of [19] used Kepler in smart manufacturing to optimize temperatures across a commercial industrial scale furnace in the steam methane reforming process used to manufacture hydrogen gas. They used Kepler workflows to combine MATLAB and ANSYS packages to manage Computational Fluid Dynamics (CFD) calculations. For distributed computing, additional instances for CFD calculations may be deployed when running in parallel using the MPI message-passing scheme.

Scientific workflows (SWF) are a cornerstone of modern scientific computing. They are used for complex computational applications that require robust management for big data that are typically stored and processed at heterogeneous, distributed resources [30]. SWF systems make scientists focus on their research rather than on details of computation management. For example, the Pegasus framework allows users to represent workflows at an abstract level while it takes care of the execution systems particulars [11]. Similarly, the main goal of the Kepler MS is to support different execution scenarios where a user can develop and use its modules to manage different execution behaviors in different environments, including private computational resources [25]. The unique features of Kepler are that the underlying workflow engine handles the provenance, reproducibility aspects of the code, performs orchestration of data flow, and automates execution on heterogeneous computing resources [33].

However, several issues arise regarding the use of the WFMs. Workflows are executed as a batch processing model, where a set of data is collected and fed into a workflow as a batch, which is processed sequentially within the corresponding workflow [15]. The workflow tasks, in this case, are closely related to each other due to the complex dependencies between them. Also, workflow jobs may generate a large amount of intermediate data during the workflow lifecycle [35]. In such tightly coupled behavior, a heavy data transfer among workflow tasks can cause a significant slow down in execution [22]. Thus, to manage and efficiently execute workflows, it is necessary to consider the features of this type of computing process, including the limitation of resources over time and planning features, taking into account the location of resulting and intermediate data [16]. One solution for such a monolith behavior problem is to divide it into smaller elements. For example, the authors in [17] proposed to divide the problem of workflow into two smaller subproblems; the first to allocating multiple workflows into multiple data centers, and the second for allocating the tasks of each workflow into the computing resources inside each data center. However, still, each traditional workflow working in batch execution mode over tightly coupled tasks. Another challenge is that the features of data visualization and data stream input/output are limited support in current WFMSs [5]. Such complexity increased in case of fog computing systems, where the execution environment itself decoupled over multiple separated geographical locations. Thus, to fit fog computing EDA

requirements, the computing system itself should be decoupled into fine-grained services that support independent deployment and lifetime management.

1.5. The Data Flow

When organizing the computational process for systems such as the Digital Twin, the data management process organization is essential. In addition to fundamental factors such as volume and data type, several key factors influence the complexity of the data management process organization related to the properties of the computing environment that supports Digital Twin.

First, the digital twin requires combining stream data processing from sensors with batch data processing when performing intelligent analysis or big data processing tasks. Second, the multi-layered and geographically distributed nature of the underlying fog computing environment requires the organization of the data flow management system as a distributed system consisting of independent components, the communication between which is organized through message exchange. Moreover, data processing for industrial applications often requires the ability to predict the state of the system. Such a service needs to know information about state history to perform prediction. This behavior is called stateful, which means that the system must identify the input data source and determine what other input data came from the same source [24]. This requirement makes the lifetime management of services that are responsible for data flow processing considerably more challenging.

However, managing state data inside a computational service itself is considered a bottleneck, so it should be stored in a separate resource [23]. An example of such a concept is presented in [9], where the authors proposed a three-layered IoT data workload processing architecture consisting of two layers for data management (messaging layer for data input/output and volatile layer - to cache the updated results), while all the processing is concentrated in the third layer. In the other case [13], the authors built a digital twin system, where a central data layer based on a publish-subscribe architecture using MQTT messaging protocol broker linked together a physical object, a digital twin, and a web-based controlling dashboard integrated with the CAD system.

2. Micro-Workflow Model

In works [2, 27], the concept of Micro-Workflow was presented in an abstract form. The current work expands this concept to be an in-detail representation with the details of creating a Micro-Workflows oriented from the partitioned workflow. Also, it defines the inputs and outputs and their accurate representation, how this model helps solve a set of challenges in using workflow in fog computing.

The Micro-Workflow concept supports redesigning a monolithic workflow into a set of smaller, loosely-coupled Micro-Workflows (MWF). Each MWF acts as an independent service supporting stream data processing, independent deployment, and communication via the streaming middleware.

2.1. The Monolith Workflow

A workflow application can be represented as a Directed Acyclic Graph (DAG):

$$W = (V, E), \tag{1}$$

where:

W – the monolith workflow;

V – a set of vertices representing computational tasks;

E – a set of directed edges connecting vertices.

Each vertex v_i in V can have input and output edges. Let's define the in-degree and out-degree of a vertex v_i in W by $deg^+(v_i)$ and $deg^-(v_i)$ respectively. Let the vertex v_1 in V be the initial vertex of the W that has no predecessors. Let n be the total number of vertices in V, so v_n would be the final vertex that does not have any successor tasks. An edge $(v_i, v_j) \in E$ represents the data that is going as one of the outputs produced by v_i to be as one of the inputs of v_j. Figure 1 shows an example of workflow W where n assumed to be 5.

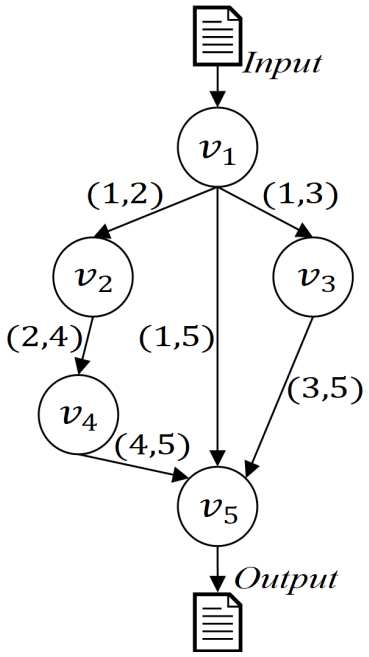


Figure 1. An example of workflow W where n assumed to be 5

2.2. The Sub-Workflow

We say that the workflow W is divided into a set of sub-workflows $S = (S_1, \dots S_k)$, $S_i = (V_i, E_i)$, when:

1. $\forall i = 1 \dots k \;\; (V_i \subset V, E_i \subset E)$;
2. $\forall v \in V, \exists S_i \;\; (v \in V_i)$;
3. $\forall i, j \;\; (i \neq j \implies S_i \cap S_j = \emptyset)$.

Figure 2 illustrates an example of workflow W partitioned into two sub-workflows S_1 and S_2.

2.3. Sub-Workflow Edges Classification

Let us define the following classes of edges and vertices, associated with the sub-workflow S_i:

- $EINT_i$: a set of internal edges, located inside of the sub-workflow S_i:

$$EINT_i = \{(v_k, v_l) \in E : v_k, v_l \in S_i\} \tag{2}$$

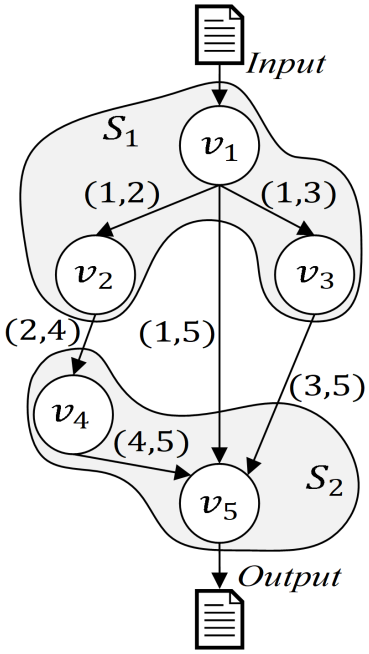


Figure 2. An example of workflow W partitioned into two sub-workflows S_1 and S_2

- $EINPUT_x$: a set of input edges with an initial vertex outside the sub-workflow S_i and an end vertex inside S_i:

$$EINPUT_i = \{(v_k, v_l) \in E : v_k \notin S_i,\ v_l \in S_i\} \tag{3}$$

- $EOUT_i$: a set of output edges with an initial vertex inside the sub-workflow S_i and an end vertex outside S_i:

$$EOUT_i = \{(v_k, v_l) \in E : v_k \in S_i,\ v_l \notin S_i\} \tag{4}$$

- $VINPUT_i$: a set of vertices in S_i that located on the head end of $EINPUT_i$ edges as well as vertices that have no input edges:

$$VINPUT_i = \{v_l \in S_i : (v_k, v_l) \in EINPUT_i\} \cup \left\{v \in S_i : deg^-(v) = 0\right\} \tag{5}$$

- $VOUT_i$: a set of vertices in S_i that located on the tail end of $EOUT_i$ edges as well as vertices that have no output edges:

$$VOUT_i = \{v_k \in S_i : (v_k, v_l) \in EOUT_i\} \cup \left\{v \in S_i : deg^+(v) = 0\right\} \tag{6}$$

2.4. Micro-Workflow Construction

To convert a sub-workflow S_i into a micro-workflow MWF_i, we need to extract all S_i vertices from the workflow W and provide communication mechanisms linking MWF_i with the event streaming platform via dedicated "Consumer vertex" (cv_i) and "Producer vertex" (pv_i) nodes. The cv_i vertex acts as a source of MWF_i, providing consumption of the input data stream from the event streaming platform and distributing it between the vertices in $VINPUT_i$. The pv_i vertex acts as a sink that collects the output from the $VOUT_i$ set and transmits it as a message to the event streaming platform. In this case, we can define the generation of $MWF_i = (MV_i,\ ME_i)$ from the sub-workflow $S_i = (V_i,\ E_i)$ as follows:

1. $MV_i = V_i \cup \{cv_i,\ pv_i\}$;

2. $ECV_i = \{(cv_i, v) : v \in VINPUT_i\}$;
3. $EPV_i = \{(v, cp_i) : v \in VOUT_i\}$;
4. $ME_i = EINT_i \cup ECV_i \cup EPV_i$.

Figure 3 shows the set of micro-workflows produced through the extraction of sub-workflows of the monolithic workflow W shown in Fig. 2.

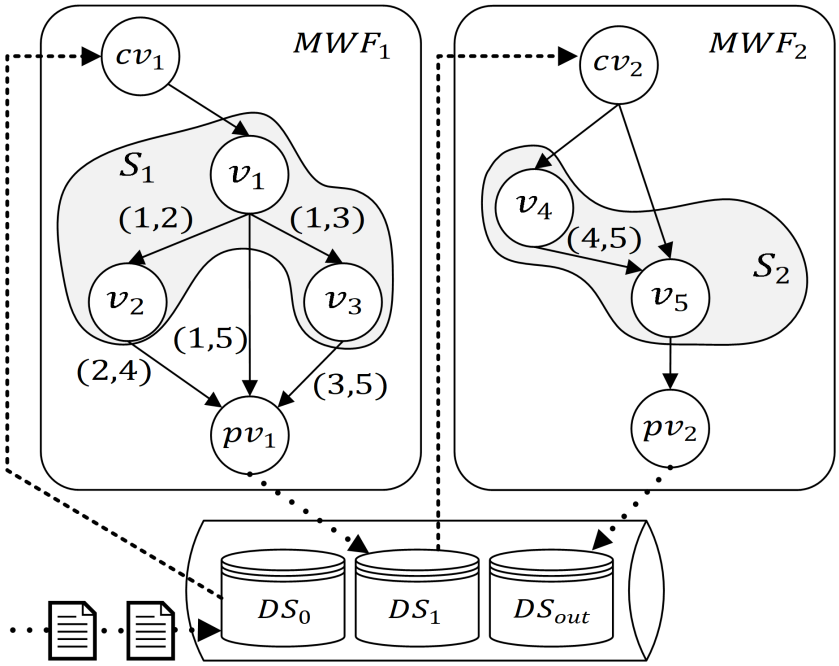


Figure 3. The result of applying the Micro-Workflow model

2.4.1. Implementation of data stream processing in micro-workflows

Micro-workflow concept integrates classical workflow and streaming data processing models. A set of dedicated channels (data stream stores) is formed in the event streaming platform structure to organize the interaction of micro-workflows via message exchange. Each message is a data set that includes a timestamp of message generation, information about the data source, and a structured collection of essential data itself. The following data stream stores are required:

- DS_0 is responsible for collecting, storing, and providing messages that contain the data sets necessary to initialize the computational process in the workflow.
- DS_i is responsible for receiving messages containing intermediate data from MWF_i and passing them to dependent micro workflows.
- DS_{out} is responsible for collecting, storing, and providing messages containing workflow result data.

Instead of the initial workflow node, the data sets needed to start the computational process are fed into the DS_0 data stream store of the event streaming platform in the form of messages. The processing of messages from the data stream store is organized as follows.

- CV_i of the corresponding MWF_i retrieves the subsequent message from DS_{i-1}.
- Based on the analysis of the received message, CV_i generates data transfer along the ECV_i edges to the nodes responsible for the direct execution of the computational process.
- After data processing tasks have been completed and data has been sent along the edges of EPV_i, PV_i generates an outgoing message to DS_i or DS_{out} if it is the final result.

Using this approach to partition a monolithic workflow into a set of independent micro-workflow computing services brings the following advantages:

- decoupling a strongly coupled computational process in time and space, switching to an asynchronous communication model;
- supporting independent micro-workflows deployment on different nodes in a distributed computing environment, taking into account the data source's geographical location, the required computing resources, etc.
- the possibility of seamless integration of IoT device data into the data processing process at any stage of the computing process;
- the ability to independently scale individual micro-workflows;
- the transfer of the computational process of the micro-workflow to a different computational node, without loss of intermediate data, and the need to repeat previous data processing.

3. Experiments

To evaluate the refactoring practicability and possible overhead of dividing the workflow into micro-workflows, we consider data processing from a typical DEBS 2012 data challenge[4] task. As part of this task, it is necessary to provide near-real-time processing of industrial Internet of Things data that transmits information about the state of manufacturing equipment.

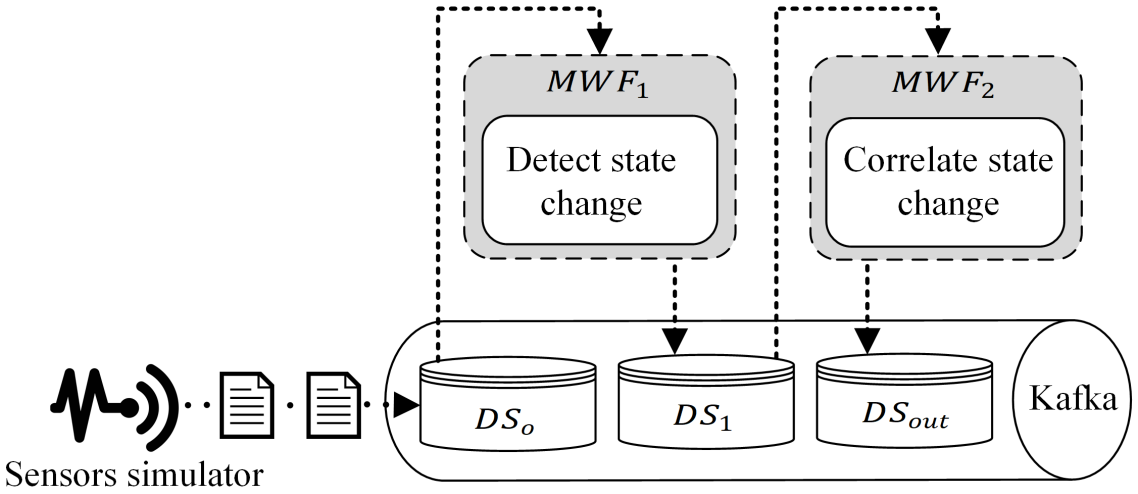


Figure 4. The organization of the experiment

Based on the proposed micro-workflow approach and model, the DEBS 2012 Query 1 monolith workflow has been refactored into two micro-workflows. Each of the micro-workflows has been implemented using the Kepler workflow system and has been packed in a separate Docker container. We have used Apache Kafka[5] as an event streaming platform to organize the message exchange between the micro-workflows. The data stream stores have been implemented as Kafka topics. The organization of the experiment is presented on the Fig. 4. To organize the data exchange between the Kepler workflow and the Apache Kafka platform, we have implemented specialized Kepler actors: KafkaConsumer, which acts as a consumer vertex, and KafkaProducer, which acts as a producer vertex for corresponding micro-workflows. We have

[4]https://debs.org/grand-challenges
[5]https://kafka.apache.org/

also developed the sensors simulator that feeds the initial sensors data stream into the DS_0 topic of the Apache Kafka.

The sensor simulator receives data from the pre-recorded sensor readings database, serializes the messages in a predetermined format, and transmits them to the DS_0 Kafka topic. Before generating and sending the following message in both experiments, we add a delay of 8 ms after receiving a successful message reception confirmation from Apache Kafka. On the one hand, this allows to approximate the data generation frequency to the one determined in the initial data stream (about 10 ms). On the other hand, we are able to keep the exact value of the introduced delay in both experiments.

The structure of the first micro-workflow MWF_1 is presented on the Fig. 5. MWF_1 is a micro-workflow that consumes the sensor data from the DS_0 Kafka topic and processes it using the *DetectStateChange* Kepler actor. The data processing results are published to the intermediate Kafka topic DS_1.

The structure of the second micro-workflow MWF_2 is presented on the Fig. 6. MWF_2 consumes the data from the DS_1 Kafka topic. It implements the second stage of data processing, including the correlation of the change of state of the sensor and the change of state of the valve by calculating the time difference between the occurrence of the state changes. The correlation estimation is performed in the *CorrelateStateChange* Kepler Actor.

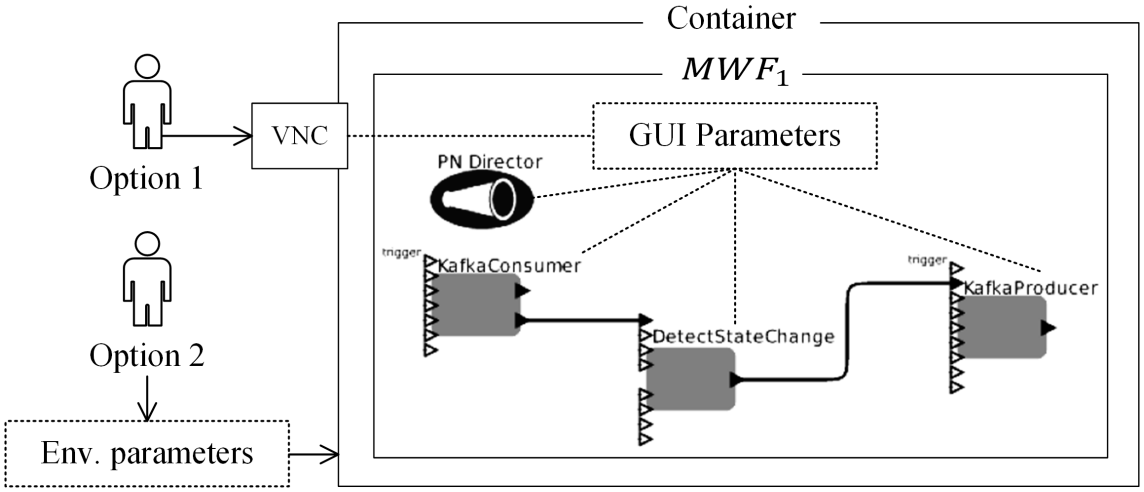


Figure 5. The two options of running MWF_1 in the developed Kepler model

3.1. Micro-Workflow Deployment Parameterization

Before launching the micro-workflow, it is necessary to configure its parameters, including the information needed to communicate with the outside world. The micro-workflow should be provided with information about the location of the endpoint address of the event streaming platform (Apache Kafka in our case), the location of the message schema repositories, and the identifiers of the data stream stores for reading and writing messages.

In our previous work [27], we have implemented the parameterization using remote desktop access to the GUI of each workflow and manually enter all execution parameters to the workflow. In this paper, we improve the implementation of the micro-workflow deployment model using the headless option. In this case, micro-workflow parameters can be provided as execution parameters into the docker container at the micro-workflow container's launch. The execution

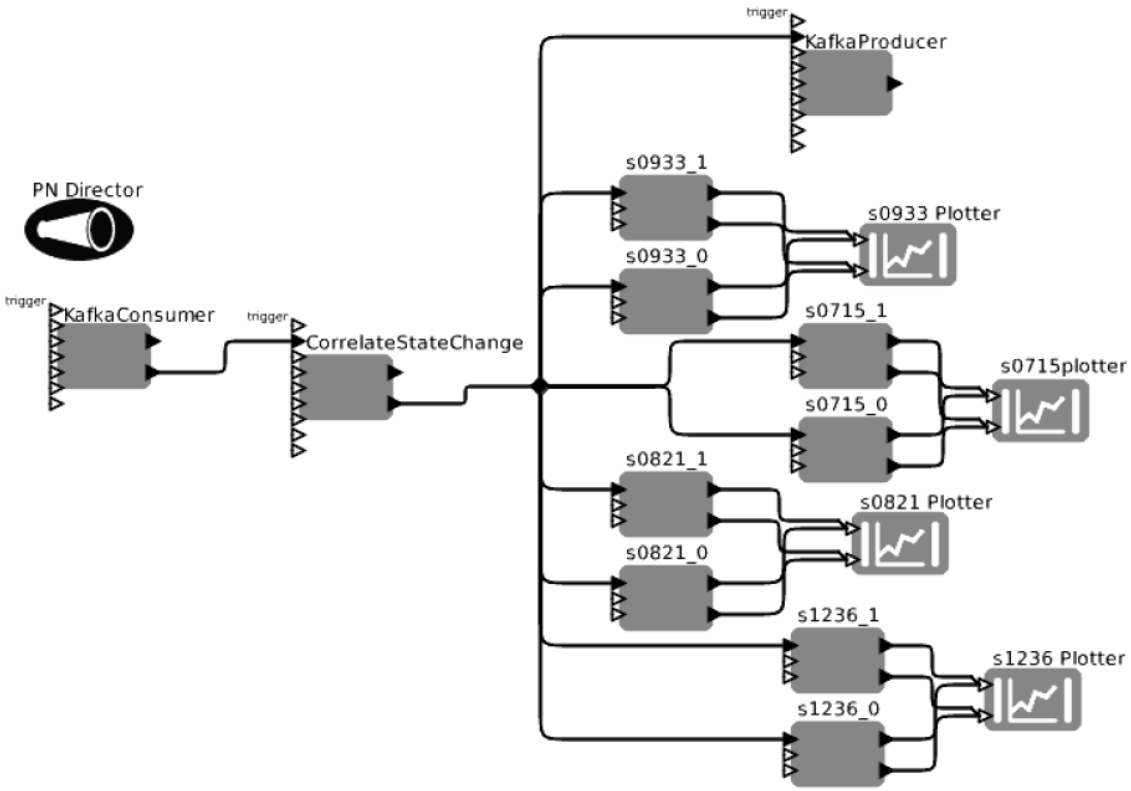


Figure 6. MWF_2 in the developed Kepler model

starts with new parameters automatically without GUI access. Figure 5 schematically shows the two options for organizing the launch of the micro-workflow described above.

In the current experiments, we have developed a Docker image that contains micro-workflow implementations using Kepler (Kepler MWF). To run a container, we need to pass the environment variables into the process. These variables include:

- MWF: the name of MWF;
- KAFKASERVER: the location of the Kafka server;
- KAFKAPORT: the port of the Kafka server endpoint;
- SCHEMAREGISTRY: the location of schema registry server;
- SCHEMAPORT: the port of the schema registry server endpoint;
- TOPICSOURCE: Kafka source topic;
- TOPICDESTINATION: Kafka destination topic.

Figure 7 shows the docker run command schema used to run a micro-workflow container.

```
docker run -it -d -p $KAFKAPORT -p $SCHEMAPORT \
        -e MWF=<?> -e KAFKASERVER=<?> \
        -e SCHEMAREGISTRY=<?> -e TOPICSOURCE=<?> \
        -e TOPICDESTINATION=<?> <developed_docker_image>
```

Figure 7. Docker command schema used to run a micro-workflow container

3.2. Experiment Deployment

We have implemented two deployment layouts for our micro-workflows during the experiment: a local deployment on a single node and a deployment in a distributed environment, where each micro-workflow has been deployed on a separate node.

The first deployment (see Fig. 8) on a single node acts as a benchmark for evaluating system performance, excluding delays introduced by network equipment and data exchange over the network. The streaming middleware, sensor emulator, MWF_1, and MWF_2 each packed in separate containers, deployed on a single computing node (Intel Core i7-4600U 2.1 GHz dual-core processor with 8 GB of RAM).

The second deployment is implemented on the resources of the Tornado Supercomputer at South Ural State University (Fig. 9). The Sensor Emulator virtual machine ($VM1$) simulates the process of IoT sensor data generation. It consumes data from DEBS 2012 database and publishes it sequentially into the Kafka input topic, deployed in the Landoop container ($VM2$). We partition the original DEBS 2012 first query workflow into two Micro-Workflows, MWF_1 and MWF_2, and pack each Micro-Workflow in a separate container, deployed on $VM2$ and $VM3$ virtual machines. There is no direct connection between the Virtual machines and nodes. All the communications need to cross a network server. Both $VM1$ and $VM3$ run on the same physical node (4 GB RAM and 4 cores of Intel Xeon X5680 CPU). $VM2$ runs on a separate physical node (12 GB RAM and 8 cores of Intel Xeon X5680 CPU). The interconnection between virtual machines is organized via an external physical node that acts as a virtual router, connected via a Gigabit Ethernet network.

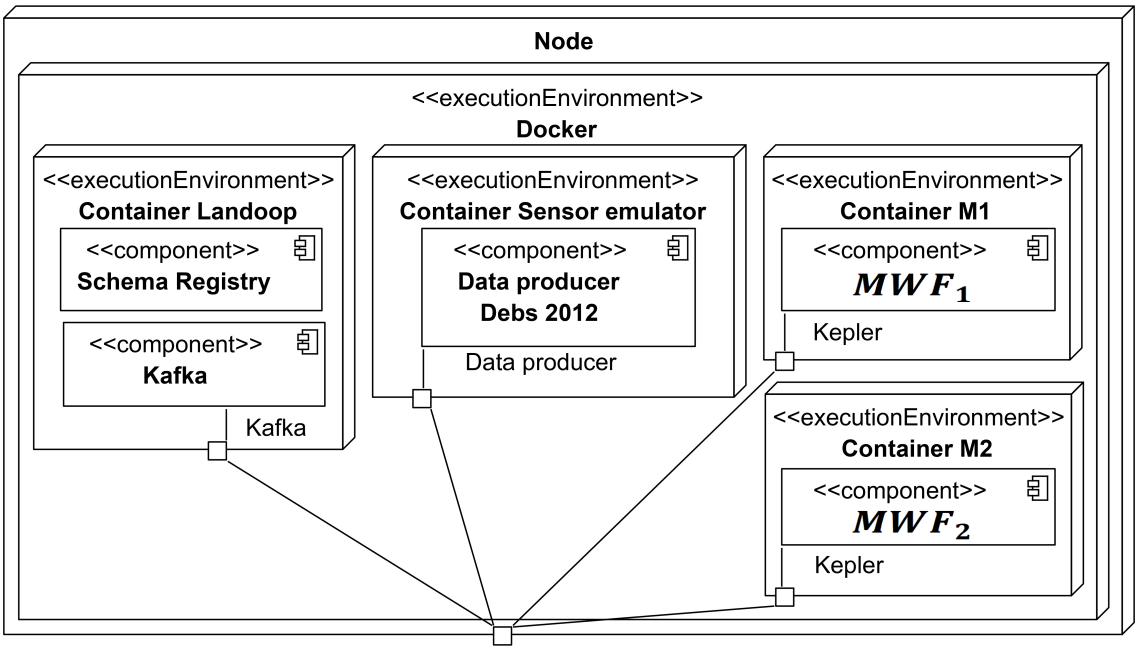


Figure 8. Local deployment on a single node

3.3. Evaluation

The following evaluation criteria are proposed to estimate the feasibility and overhead of the micro-workflow refactoring:

- AV_{sm}: the average interval between sensor messages.

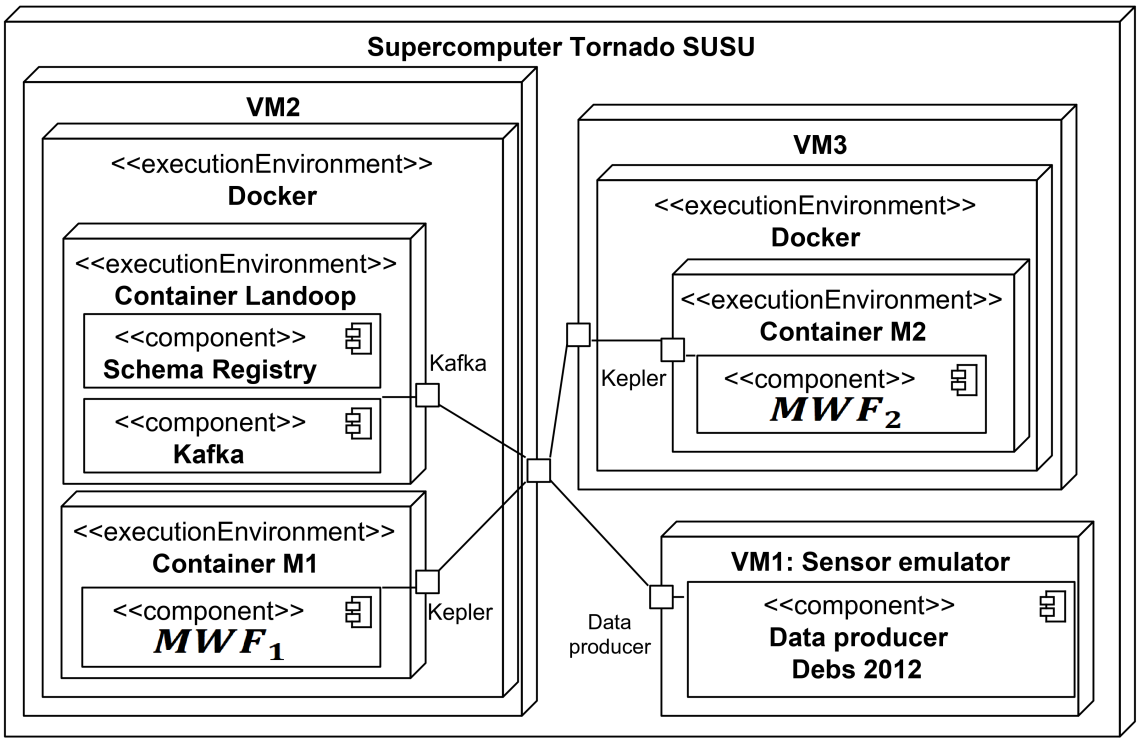


Figure 9. Deployment in a distributed computing environment on three nodes

- AV_{tat} (average processing time): the average time interval required to create one final message by a micro-workflow. The interval counted from the second requested original message has been received from the original Kafka topic to the time the final message sent to Kafka.
- AV_{dv} (average delivery time): average time interval between sending sensor message and receiving the message in MWF, including data serialization, message transfer from VM1 to VM2.
- AV_{l12}: the average network latency between VM1 and VM2.

The comparison results of the experiments are presented in Tab. 1.

Table 1. The comparison results

Parameters	Single node	Distributed mode
Testing time	24 hours	24 hours
Total messages	9 180 056	7 328 844
AV_{sm}	9.5 ms	11.8 ms
AV_{tat}	1.3 ms	3.2 ms
AV_{dv}	1.2 ms	4.4 ms
AV_{l12}	0 ms	3.5 ms

Analysis of the results of the experiment allows us to draw the following conclusions:

- Application of the workflow partitioning mechanism into independent micro-workflows allows providing IoT data processing in streaming mode, close to real-time. The average processing time in both experiments has been significantly less than the sensor's period of initial data generation.

- Computational processes of individual micro-workflows can be distributed across nodes in a distributed computing network. With such distribution, the data processing time increases by at least the latency between the computing network's corresponding nodes and the node where the event streaming platform is located.

Conclusion

We propose a model of workflow applications for stream processing in highly distributed environments such as fog computing. The ability to move part of computing from/into different nodes at a different level is the required by system architecture of fog computing. We show how tightly coupled dependencies in monolith workflow can be decoupled and refactored in the form of smaller and standalone micro-workflows and how define inputs and outputs of each resulting micro-workflow. Such decoupling supports the independence of implementation, execution, development, maintenance, and cross-platform deployment of micro-workflows as standalone computational units. An important direction of further research is to automate the refactoring workflow process into micro-workflows.

Acknowledgements

The reported study was funded by the Ministry of Science and Higher Education of the Russian Federation (government order FENU-2020-0022) and by RFBR, project number 19-37-90073.

References

1. Aazam, M., Zeadally, S., Harras, K.A.: Deploying Fog Computing in Industrial Internet of Things and Industry 4.0. IEEE Transactions on Industrial Informatics 14(10), 4674–4682 (2018), DOI: 10.1109/TII.2018.2855198

2. Alaasam, A.B.A., Radchenko, G., Tchernykh, A., Borodulin, K., Podkorytov, A.: Scientific Micro-Workflows: Where Event-Driven Approach Meets Workflows to Support Digital Twins. In: Proceedings of the International Conference Russian Supercomputing Days (RuSCDays'18), 24-25 Sept. 2018, Moscow, Russia. vol. 1, pp. 489–495 (2018)

3. Alaasam, A.B.A., Radchenko, G., Tchernykh, A.: Stateful Stream Processing for Digital Twins: Microservice-Based Kafka Stream DSL. In: 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON), 21-27 Oct. 2019, Novosibirsk, Russia. pp. 0804–0809. IEEE (2019), DOI: 10.1109/SIBIRCON48586.2019.8958367

4. Alaasam, A.B.A., Radchenko, G., Tchernykh, A., González Compeán, J.L.: Analytic Study of Containerizing Stateful Stream Processing as Microservice to Support Digital Twins in Fog Computing. Programming and Computer Software 46(8), 511–525 (2020), DOI: 10.1134/S0361768820080083

5. Badia, R.M., Ayguade, E., Labarta, J.: Workflows for science: a challenge when facing the convergence of HPC and Big Data. Supercomputing Frontiers and Innovations 4(1), 27–47 (2017), DOI: 10.14529/jsfi170102

6. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, 13-17 Aug. 2012, Helsinki, Finland. pp. 13–15. Association for Computing Machinery, New York, NY, USA (2012), DOI: 10.1145/2342509.2342513

7. Boyes, H., Hallaq, B., Cunningham, J., Watson, T.: The industrial internet of things (IIoT): An analysis framework. Computers in Industry 101(December 2017), 1–12 (2018), DOI: 10.1016/j.compind.2018.04.015

8. Cao, J., Zhang, Q., Shi, W.: Challenges and Opportunities in Edge Computing, pp. 59–70. Springer, Cham (2018), DOI: 10.1007/978-3-030-02083-5_5

9. Carvalho, O., Roloff, E., Navaux, P.O.: A Distributed Stream Processing based Architecture for IoT Smart Grids Monitoring. In: Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, 5-8 Dec. 2017, Austin, Texas, USA. pp. 9–14. ACM, New York, NY, USA (2017), DOI: 10.1145/3147234.3148105

10. Chandy, K.M.: Event Driven Architecture. In: Encyclopedia of Database Systems, pp. 1040–1044. Springer US, Boston, MA (2009), DOI: 10.1007/978-0-387-39940-9_570

11. Deelman, E., Singh, G., Su, M.H., et al.: Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming 13(3), 219–237 (2005), DOI: 10.1155/2005/128026

12. Goyal, P., Mikkilineni, R.: Policy-Based Event-Driven Services-Oriented Architecture for Cloud Services Operation & Management. In: 2009 IEEE International Conference on Cloud Computing, 21-25 Sept. 2009, Bangalore, India. pp. 135–138. IEEE (2009), DOI: 10.1109/CLOUD.2009.76

13. Haag, S., Anderl, R.: Digital twin – Proof of concept. Manufacturing Letters 15, 64–66 (2018), DOI: 10.1016/j.mfglet.2018.02.006

14. Hao, Z., Novak, E., Yi, S., Li, Q.: Challenges and Software Architecture for Fog Computing. IEEE Internet Computing 21(2), 44–53 (2017), DOI: 10.1109/MIC.2017.26

15. Hirales-Carbajal, A., Tchernykh, A., Roblitz, T., Yahyapour, R.: A Grid simulation framework to study advance scheduling strategies for complex workflow applications. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 19-23 April 2010, Atlanta, GA, USA. pp. 1–8. IEEE (2010), DOI: 10.1109/IPDPSW.2010.5470918

16. Hirales-Carbajal, A., Tchernykh, A., Yahyapour, R., et al.: Multiple Workflow Scheduling Strategies with User Run Time Estimates on a Grid. Journal of Grid Computing 10(2), 325–346 (2012), DOI: 10.1007/s10723-012-9215-6

17. Iturriaga, S., Nesmachnow, S., Tchernykh, A., Dorronsoro, B.: Multiobjective Workflow Scheduling in a Federation of Heterogeneous Green-Powered Data Centers. In: 2016 16th

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 16-19 May 2016, Cartagena, Colombia. pp. 596–599. IEEE (2016), DOI: 10.1109/CCGrid.2016.34

18. Kalske, M., Mäkitalo, N., Mikkonen, T.: Challenges When Moving from Monolith to Microservice Architecture. In: Current Trends in Web Engineering, 5-8 June 2017, Rome, Italy, pp. 32–47. Springer, Cham (2018), DOI: 10.1007/978-3-319-74433-9_3

19. Korambath, P., Wang, J., Kumar, A., Davis, J., Graybill, R., Schott, B., Baldea, M.: A Smart Manufacturing Use Case: Furnace Temperature Balancing in Steam Methane Reforming Process via Kepler Workflows. Procedia Computer Science 80, 680–689 (2016), DOI: 10.1016/j.procs.2016.05.357

20. Li, X., Song, J., Huang, B.: A scientific workflow management system architecture and its scheduling based on cloud service platform for manufacturing big data analytics. The International Journal of Advanced Manufacturing Technology 84(1-4), 119–131 (2016), DOI: 10.1007/s00170-015-7804-9

21. Luo, J., Yin, L., Hu, J., Wang, C., Liu, X., Fan, X., Luo, H.: Container-based fog computing architecture and energy-balancing scheduling algorithm for energy IoT. Future Generation Computer Systems 97, 50–60 (2019), DOI: 10.1016/j.future.2018.12.063

22. Miranda, V., Tchernykh, A., Kliazovich, D.: Dynamic Communication-Aware Scheduling with Uncertainty of Workflow Applications in Clouds. In: High Performance Computer Applications, 9-13 March 2015, Mexico City, Mexico, Communications in Computer and Information Science, vol. 595, pp. 169–187. Springer, Cham (2016), DOI: 10.1007/978-3-319-32243-8_12

23. Naseri, M., Towhidi, A.: Stateful Web Services: A Missing Point in Web Service Standards. In: Proceedings of the International MultiConference of Engineers and Computer Scientists 2007 (IMECS 2007). pp. 993–997. Hong Kong, China (2007)

24. Peiffer, C., L'Heureux, I.: System and method for maintaining statefulness during client-server interactions. (12) United States Patent (US8346848B2) (2013), `https://patents.google.com/patent/US8346848B2/en`

25. Plociennik, M., Zok, T., Altintas, I., et al.: Approaches to Distributed Execution of Scientific Workflows in Kepler. Fundamenta Informaticae 128(3), 281–302 (2013), DOI: 10.3233/FI-2013-947

26. Qamsane, Y., Chen, C.Y., Balta, E.C., et al.: A unified digital twin framework for real-time monitoring and evaluation of smart manufacturing systems. In: 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE), 22-26 Aug. 2019, Vancouver, BC, Canada. pp. 1394–1401 (2019), DOI: 10.1109/COASE.2019.8843269

27. Radchenko, G., Alaasam, A.B., Tchernykh, A.: Micro-Workflows: Kafka and Kepler Fusion to Support Digital Twins of Industrial Processes. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 17-20 Dec. 2018, Zurich, Switzerland. pp. 83–88. IEEE (2018), DOI: 10.1109/UCC-Companion.2018.00039

28. Richards, M.: Software Architecture Patterns. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472 (2015)

29. Savchenko, D., Radchenko, G., Taipale, O.: Microservices validation: Mjolnirr platform case study. In: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 25-29 May 2015, Opatija, Croatia. pp. 235–240. IEEE (2015), DOI: 10.1109/MIPRO.2015.7160271

30. da Silva, R.F., Pottier, L., Coleman, T., Deelman, E., Casanova, H.: WorkflowHub: Community Framework for Enabling Scientific Workflow Research and Development. In: 2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 12 Nov. 2020, GA, USA. pp. 49–56. IEEE (2020), DOI: 10.1109/WORKS51914.2020.00012

31. Simpkin, C., Taylor, I., Harborne, D., Bent, G., Preece, A., Ganti, R.K.: Efficient orchestration of Node-RED IoT workflows using a Vector Symbolic Architecture. Future Generation Computer Systems 111, 117–131 (2020), DOI: 10.1016/j.future.2020.04.005

32. Tao, F., Sui, F., Liu, A., et al.: Digital twin-driven product design framework. International Journal of Production Research 57(12), 3935–3953 (2019), DOI: 10.1080/00207543.2018.1443229

33. Yang, P.C., Purawat, S., U. Ieong, P., et al.: A demonstration of modularity, reuse, reproducibility, portability and scalability for modeling and simulation of cardiac electrophysiology using Kepler Workflows. PLOS Computational Biology 15(3), e1006856 (2019), DOI: 10.1371/journal.pcbi.1006856

34. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications 1(1), 7–18 (2010), DOI: 10.1007/s13174-010-0007-6

35. Zheng, C., Tovar, B., Thain, D.: Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2017), 14-17 May 2017, Madrid, Spain pp. 130–139 (2017), DOI: 10.1109/CCGRID.2017.9