EC7 AS15

User Documentation for Performance Prediction

Document reference number for ANR

contact@confiance-ai.fr | www.confiance.ai

Specific information to be provided in ANR reports only

X	Progress report		Final report
From:	AAAA/MM/JJ	To:	AAAA/MM/JJ
Brief description of the project (reminder)			
Contractual description of the project:		~~nom du fichier~~	
Reference budget:		~~nom du fichier~~	

RECOMMANDATIONS

Strikethrough text is used to illustrate and explain the headings. It should be removed and replaced with non-strikethrough text. Similarly, this recommendations block should be removed, along with the statement "Specific information to be provided in ANR reports only", prior to dissemination of the document.

Document reference: 715E

Contributors

	Name	Organisation	Role
Responsible for the deliverable	E. JENN	IRT Saint Exupéry	Action sheet leader
Scientific responsible	E. JENN	IRT Saint Exupéry	Action sheet leader
Co-authors	C. LOUIS-STANISLAS H. BOUZIDI S. NIAR	VIVERIS LAMIH LAMIH	Research engineers Ph.D student University Professor

Document control

Revision	Date	Commentary	Author
1.0	2023/12/22	First version	Cf. authors

Table of contents

A. Introduction and Abstract

A.1 General introduction to trustworthy AI challenges

Trustworthiness in AI within critical systems (systems that can directly or indirectly affect human life and moral entities) is essential for its widespread adoption (by the industry, the decision makers, the general public, etc.) and poses the following significant challenges.

- First, how to design AI models, so that, by construction, they satisfy trustworthy properties (accuracy, robustness…).
- Secondly, how to characterize these AI models, for example to understand and explain their behavior and their adequacy to the operational domain.
- Then, how to implement and embed those AI models on hardware, by making them fit for the target without losing their trustworthy properties.
- Another question is, what methods of data engineering to apply in order to, among other topics, manage important volumes of data and adapt to the evolution of the operational domain.
- At system level, what verification and certification processes to consider specifically for AI-based systems.
- Finally, a federation of all these matters is necessary to build an end-to-end methodological approach, supported by a consistent engineering environment compatible with industrial practices.

These are the challenges, among others, that the Confiance.ai program addresses.

A.2 Specific challenges

The benchmarking environment specifically addresses the third challenge concerning the implementation and deployment of an AI model. Indeed, once the AI model has been designed and trained, and functional performances have been demonstrated to comply with the requirements, the model needs to be deployed and executed on some hardware using a series a of transformations including (possibly) platform-specific optimizations, translation to specific forms, compilation, etc. This process raises multiple new challenges. First, it is highly platform-dependent, and different targets usually require – at least partially – specific tools. Second, setting up those tools can be challenging, often attributed to factors such as a lack of maturity, complex dependencies between libraries, or the need for specific skills (e.g., FPGA development) and knowledge (e.g., specific library). Third, conducting performance and resource usage comparisons across multiple hardware targets requires the purchase and setup of those targets, incurring both time and financial expenses. The benchmarking environment tries to address all those challenges by providing a set of

“encapsulated” AI development tools and hardware targets (i.e., computation boards) ready to be used with a minimum effort. The environment also provides the necessary scripting, hardware management, and orchestration means to automate a series of experiments.

A.3 Objectives of the document

In a recent study conducted by Bouzidi and colleagues, a cutting-edge approach to modeling Convolutional Neural Network (CNN) performance on edge GPU-based platforms has been introduced. Their method relies on a meticulous characterization of CNN architectures conducted at a model-level granularity. Employing this approach, the researchers implemented five efficient Machine Learning (ML) algorithms designed for predicting the performance of CNNs across various edge GPUs.

Expanding upon the prior research, this document endeavors to replicate, within a framework emphasizing trust, the methodology employed for embedding a performance prediction mechanism into the ML bench. Subsequently, it introduces this streamlined methodology, customized to meet our specific requirements, offering insights into the challenges faced during implementation. Additionally, the paper elucidates the adjustments made to ensure that the implementation aligns seamlessly with the constraints associated with system criticality.

Our experimental endeavors revealed notable outcomes, demonstrating our capability to attain a minimum accuracy of 88% for performance predictions. These findings underscore the effectiveness of our refined methodology in navigating the intricacies of trustworthy machine learning applications while adhering to the rigorous constraints inherent in critical systems.

B. Objectives

The objective of the performance prediction mechanism, meant to be embedded in the ML benchmarking environment is to provide the guidance, performance wise to (1) support and optimize the deployment environment choice on the ML-bench (2) help users to investigate other hardware configurations.

The performance prediction mechanism has 2 main usages inside the ML-bench:

1. Investigate the potential best deployment tools chain implemented on the ML-bench before the actual inference.
2. Investigate the potential best hardware target/configuration given performance requirements (execution time, ram usage…)

To reach the end objective, the implementation of this performance prediction mechanism was organized along 3 sequential phases:

- In the first one, the goal is to be able to produce a performance prediction model per embedded target architecture. For this phase, the scope is limited to GPU targets (Jetson Nano, Orin, TX2…)

- Based on the results of the first phase, the next one aims to design a generalized model (if possible). Ideally at the end of this phase, the performance prediction mechanism should be able to estimate the performance of any ML models on all the embedded targets of the ML-bench.
- Finally, during the final phase, the emphasis will be on integrating hardware components into the performance prediction mechanism. This stage will allow the investigation of features, particularly for targets that are not available on the ML-bench.

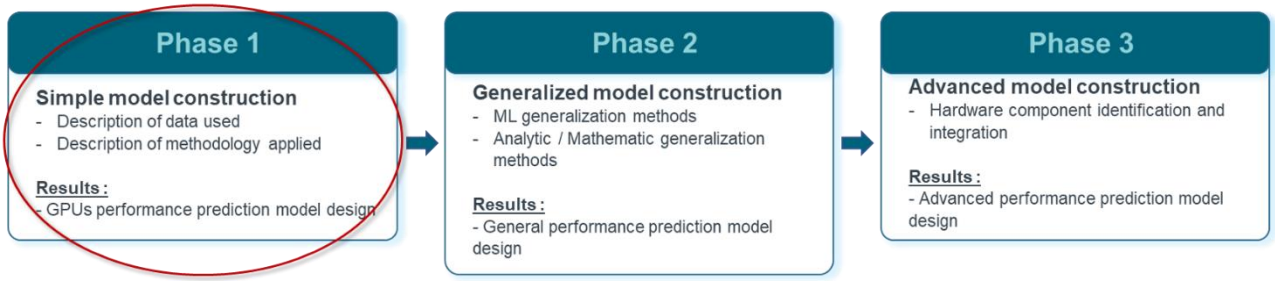

Figure 1 : Phase description

This document focuses on phase 1 (red ellipse in Figure 1).

This document is organized as follow:

1. First a state of the art of this mechanism is presented. This section mainly presents scientific works of the LAMIH, a Valenciennes University, led by H.Bouzidi and S.Niar. Their work is the foundation of the prediction mechanism highlighted in this document.
2. The next section focuses on the input data more specifically. It describes the types of data utilized, explains the rationale behind their selection, and specifies the methods employed for their collection.
3. Then, the aim of the next section is to provide the global methodology used to construct the different models. Ideally the methodology should be totally independent of the embedded target. However, heterogeneous architecture may be challenging to deal with. Therefore, the presented methodology will at least be relevant for GPU embedded targets.
4. The next section highlights the most relevant results
5. Finally, the last section concludes the document, summarizing the previous chapters and results.

C. State of the art

In this section, we explore recent developments in performance prediction techniques tailored for Computer Vision-based Convolutional Neural Networks (CNNs) deployed on edge Graphics Processing Units (GPUs). The need for efficient execution time, power consumption, and memory usage has become crucial, particularly in the context of edge platforms, which are characterized by computation and energy constraints.

Execution Time Modeling

Previous research has extensively addressed the modeling of execution time for CNNs, recognizing its significance in real-time applications with strict constraints, such as autonomous driving. Existing approaches involve Machine Learning (ML) techniques, including linear regression, Support Vector Regression (SVR), and Random Forest (RF), applied to GPU platforms [Amaris, 2016]. However, some models focus on general-purpose applications, limiting their adaptability to Deep Learning (DL) applications. Additionally, efforts have been made to model DL training workloads, identifying performance bottlenecks and optimizing training times, especially in distributed GPU environments [Wang, 2019]. Notably, predictions for training times, particularly with Stochastic Gradient Descent (SGD) optimization, have been proposed, emphasizing the importance of considering communication time, I/O processing time, and GPU processing time [Shi, 2018]. Contrastingly, our approach considers model-level granularity, offering more generalization for various CNN architectures, and even extend it to other ML models.

Latency and Memory Modeling

Latency prediction on FPGA-based platforms and memory usage modeling have been addressed as critical aspects. Various techniques involve analytical modeling of CNN execution time on FPGA platforms, including lookup tables and schedulers [Mu, 2020] [Zhang, 2020]. However, regarding GPU platforms, complexities arise in implementing schedulers, and the lack of open-source information on GPU schedulers poses challenges for edge GPUs. Memory usage modeling has focused on estimating the total memory required for CNN inference, considering parameters, intermediate activations, and DL framework computations [Stamoulis, 2018]. Our work extends these approaches by estimating total memory needs on edge GPUs, emphasizing the importance of DL framework memory allocation.

Bridging the Gap: ML-Based Performance Modeling on Edge GPUs

In a recent study by Bouzidi et al. [Bouzidi, 2022], the authors propose a Machine Learning-based modeling approach for CNN performance on edge GPU-based platforms. Their methodology involves characterizing CNN architectures at a model-level granularity and implementing five efficient ML algorithms for performance predictions on different edge GPUs. The study demonstrates the robustness of their proposed approach, achieving low prediction errors for execution time, power consumption, and memory usage.

Generalization and Future Directions

While existing works have contributed significantly to performance modeling, challenges persist, especially concerning the diversity and complexity of CNN and edge GPU architectures. The work by Bouzidi et al. presents a promising solution by a priori prediction of CNN performances on edge GPUs, offering a quick determination of the best CNN-edge hardware combination. The authors successfully generalize their approach over different exploration spaces and edge GPUs, providing a foundation for multi-objective optimization in CNN design space exploration.

In conclusion, advancements in performance modeling for Computer Vision-based CNNs on edge GPUs have made significant strides, with recent work focusing on ML-based approaches that exhibit robustness and efficiency. The integration of such models into a broader optimization framework and the exploration of diverse applications and hardware architectures present exciting avenues for future research.

D. Data

As mentioned in the state of the art, talking about “performance” in performance prediction mostly refer to 3 metrics:

- Execution time or the time needed for an inference to complete and return results
- Memory usage or the RAM consumption of an inference
- Power usage or the power consumption of an inference.

For the ML-bench end product, we decided to exclude the power consumption from the pool of tracking metrics.

To design a model able to predict these metrics depending on ML models features, 3 types of data are needed:

- Execution time, directly measured from real inference
- RAM consumption, directly measured from real inference
- Relevant ML models features that significantly impact execution time and RAM usage.

D.2 ML models features

In their work, LAMIH describes various sets of meaningful ML features according to the target metrics. A summary of these sets will be presented here.

Figure 2 highlights the feature ranking for each metric from the most to the least important. The importance is assessed by calculating the F-score criterion, using the model prediction error (MAPE) as the improvement criterion.

Relevant ML features are:

- **FLOPS** (FLoating-point OPerationS) represents the computational workload of a ML model
- **activations** represents the number of activation layer in the ML model
- **neurons** represents the number of neurons
- **conv-params** represents the number of parameters of convolutional layers
- **nb-layers** represents the total number of layers
- **input-size** represents the size of the input image
- **fc-params** represents the number of parameters of fully-connected layers
- **bn-params** represents the number of parameters of batch normalization layers
- **bn-layers** represents the number of batch normalization layers
- **conv-layers** represents the number of convolutional layers
- **fc-layers** represents the number of fully-connected layers

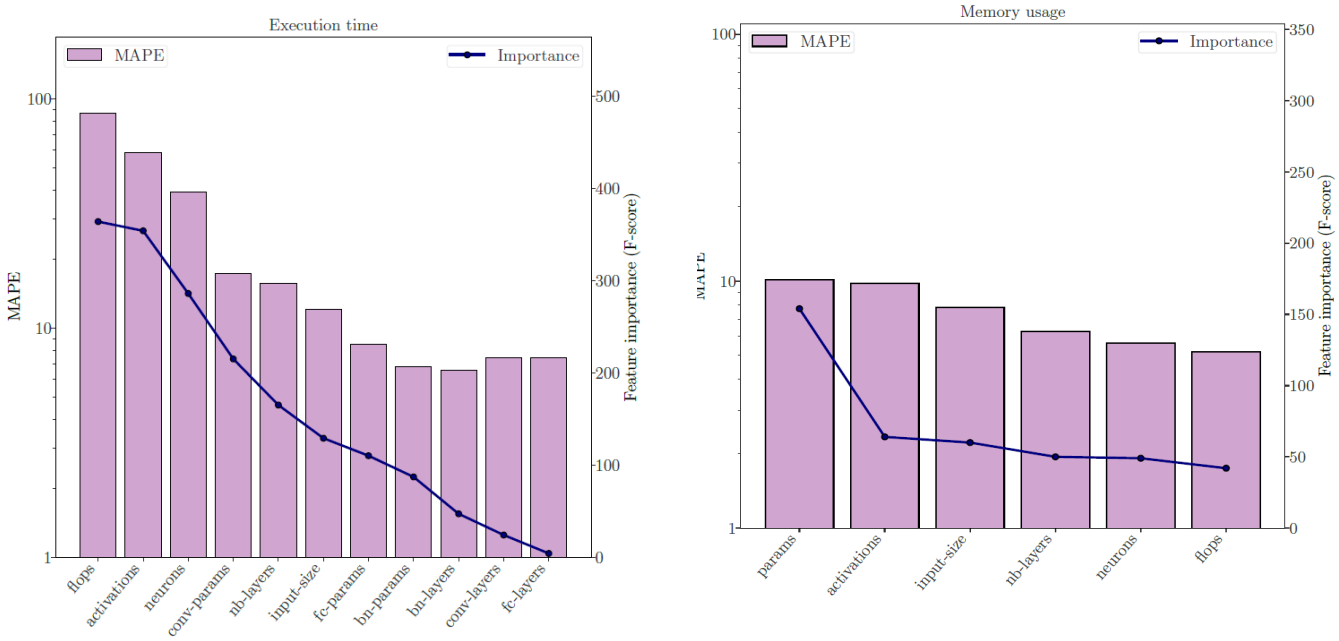

Figure 2. Results of the stepwise feature selection based on the calculated feature importance with the corresponding improvement in MAPE for each performance metric

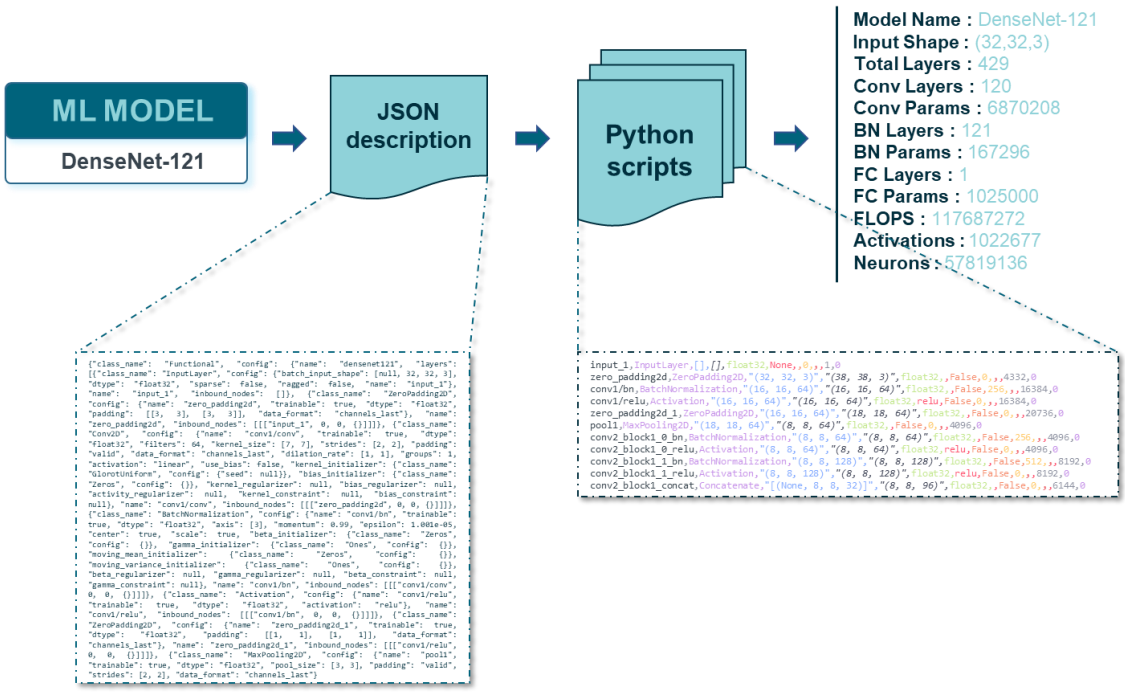

Figure 3 : Extraction feature process

Figure 3 describes the general process of feature extraction of ML models. Being totally independent from embedded target, this task only needs to be executed once. For phase 1, mainly Convolutional Neuronal Network have been used. Synthetic models had also been used for a specific purpose detailed in the “Nano” section. Nevertheless, selected features are general enough to be compatible with other types of ML models, such as Fully Connected ones. The values of all the features are stored in a database, that will be used later for the training and the testing of finals models.

D.3 Execution time measurement

Inference being launched from python script, the most intuitive way to assess the inference time is to measure the “wall-time”: so, the system clock time is captured once before the inference in the code, and the second time after the inference. The “wall-time” is the delta time between the two captures. In our case, inference time is measured for multiple predefined iterations of the same ML model, and we capture at the end the mean inference time of all iterations. This way, the measured execution is not biased by light variations.

Wall-time measurement

This time measure method seems simple and straightforward. However, after several measurements we observed four different pattern of inference time variations:

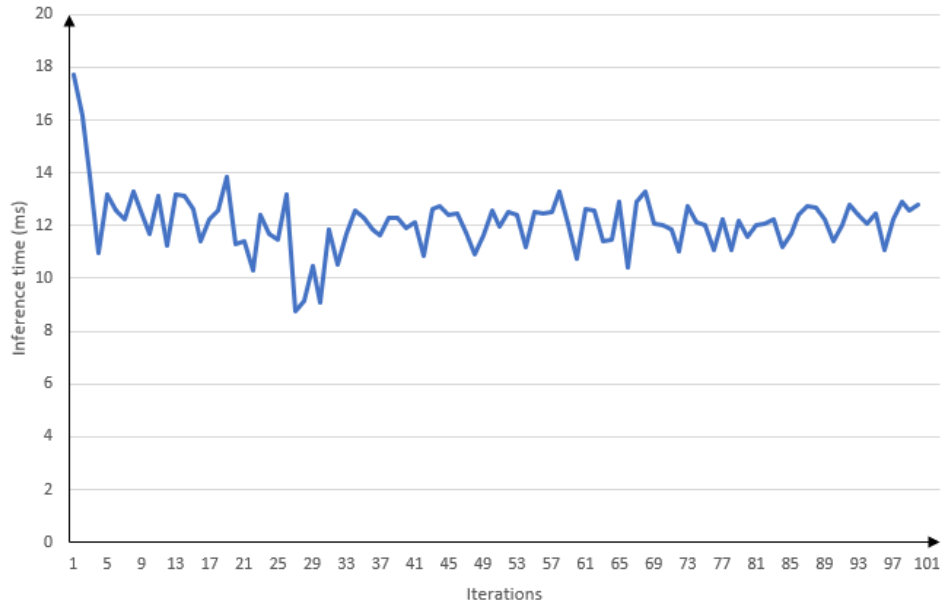

Figure 4 : White noise variation pattern

1. “White noise” variation pattern, or low noise amplitude varying around a mean value

This variation pattern is the expected one. The lights variations due to various system or inference “noise” are mitigated by the final mean value. In the example shown in **Erreur ! Source du renvoi introuvable.**, it is acceptable to confidently assume that this ML model has an execution time around 12ms.

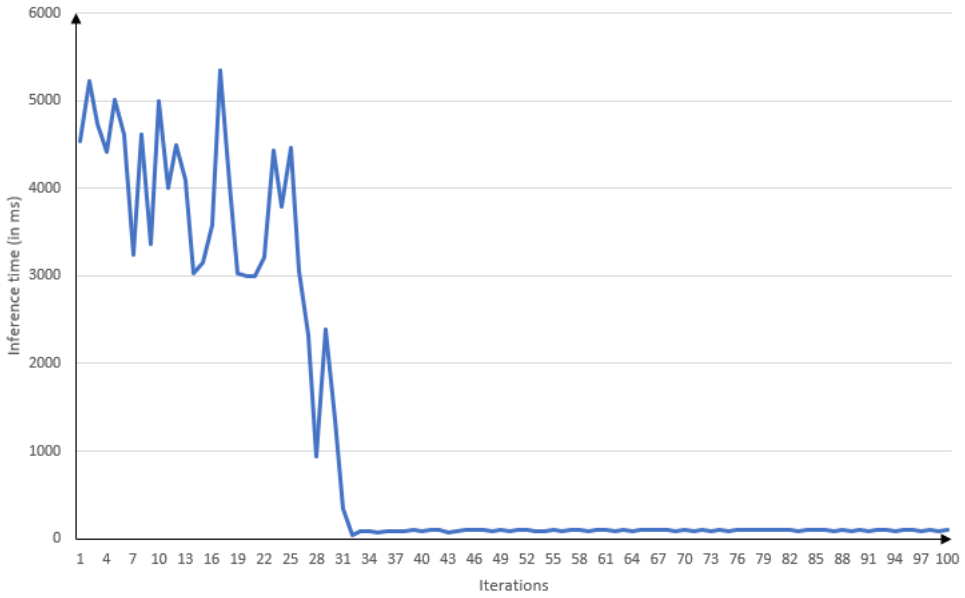

Figure 5 : Sequential variation pattern

2. Sequential variation pattern

In performance measurement and benchmarking of machine learning models, it is common to discard the results from the initial iterations of an inference: it is often referred to as the “warming up” of models. This “warming up” phase is mainly due to overheads initialization, hardware warm-up or even memory allocation.

At first it seems that this phenomenon occurs in the **Erreur ! Source du renvoi introuvable.** above. Nevertheless, the warming-up phase is generally contained within the first 2 or 3 iterations. Moreover, in other examples, we can observe this phase in the middle or at the end of the inference. Although warming-up phase was initially considered, the following patterns of the next two variations led us in another direction.

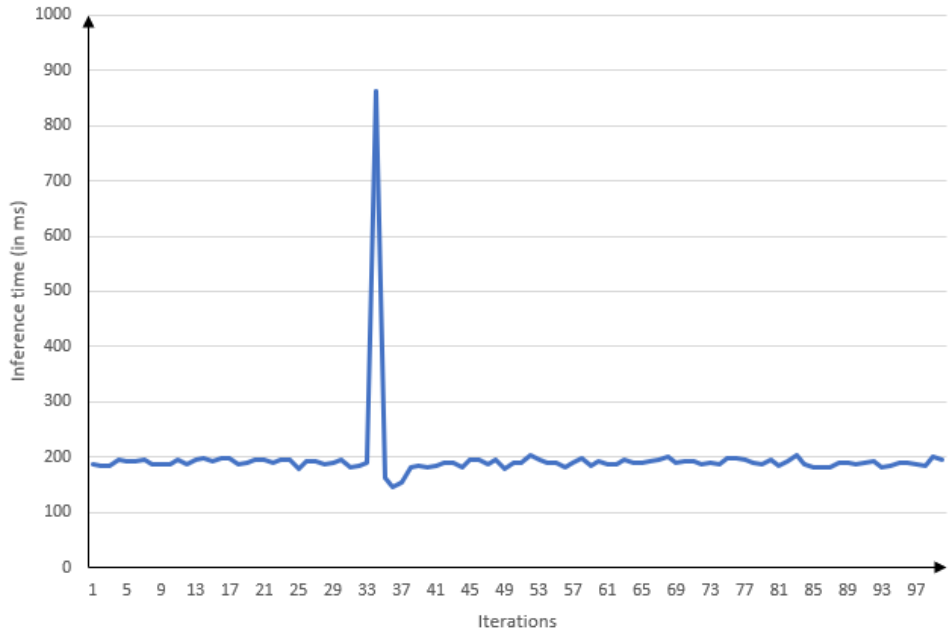

Figure 6 : Large amplitude "high" spikes pattern

3. Large amplitude "high" spikes

This variation is quite problematic. Firstly, it shows that some phenomenon can highly impact execution time to the point where only considering the final mean value leads to a wrong final performance measurement. For example, in **Erreur ! Source du renvoi introuvable.**, the mean value of this ML model inference time should be around 60ms, while we can clearly observe that most of the time the inference is done around 15ms. One potential solution could have been to consider the spike as an "outlier" and remove it from the calculation of the mean value. However, taking such action gives rise to further inquiries: how to confidently identify an outlier? Is it the inference time magnitude differential? What could be the value of this differential? How many occurrences are rare enough to be considered as outliers?

The next pattern variation convinced us that this way was not the solution.

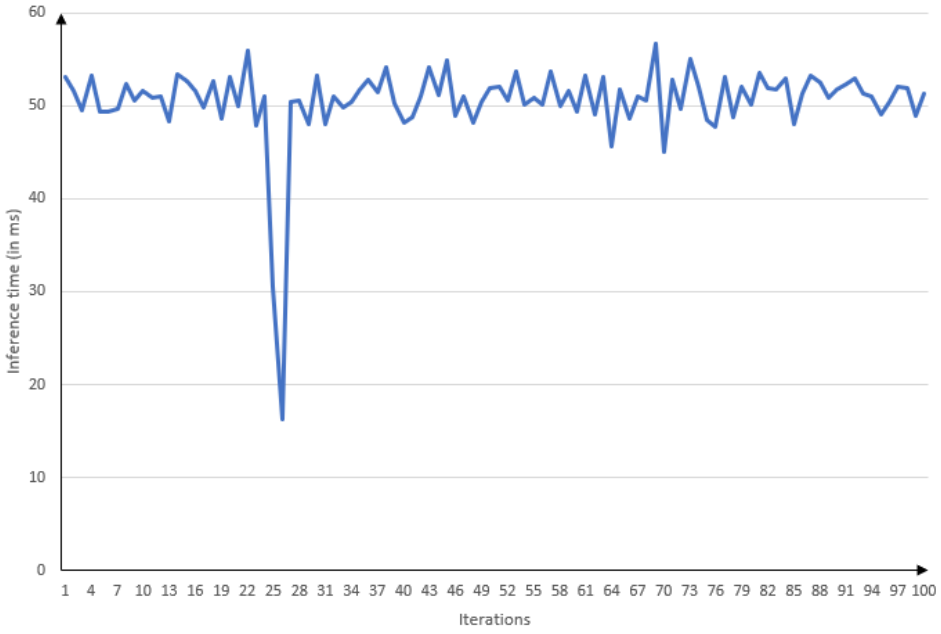

Figure 7 : Large amplitude "low" spikes pattern

4. Large amplitude "Low" spikes

This variation pattern is as problematic as the previous one, but also reflects a kind of "strange" phenomenon. For example, the **Erreur ! Source du renvoi introuvable.** shows that the inference can be performed with an inference time around 50% below the mean value.

As the previous variation pattern, we first thought of an outlier. So, for the low spikes to be considered as "outlier" we intuitively believed that the inference had not been really executed. More precisely, we considered that an error may have occurred during the inference execution, causing the script to be terminated prematurely, and thus resulting in an incorrect time measurement. Then, we investigated in this direction by capturing the results of each iteration multiple times. Results were perfectly identical every single time, while low spikes appeared. So, it means that low spikes are as legitimate as the rest of time measurements.

The observation of these variation patterns and resulting conclusion implies that wall-time measurement is strongly influenced by all the background tasks of the underlying system making it very dependent on the embedded system.

With the will to increase the control and the precision of the time measurement, we explored other means of measurement, such as:

- only take into consideration the process time, which represents the CPU time of a specific instruction
- using Nvidia measuring tools like Nsys
- using the framework measuring tools: for our case it is PerfZero, a Tensorflow tool

Process time measurement

> **time.process_time()** function always returns the float value of time in seconds. Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid
>
> *www.geeksforgeeks.org*

By using process time, the aim was to avoid interruptions from other processes (see **Erreur ! Source du renvoi introuvable.**) and only capture the elapsed time of a specific process: the inference

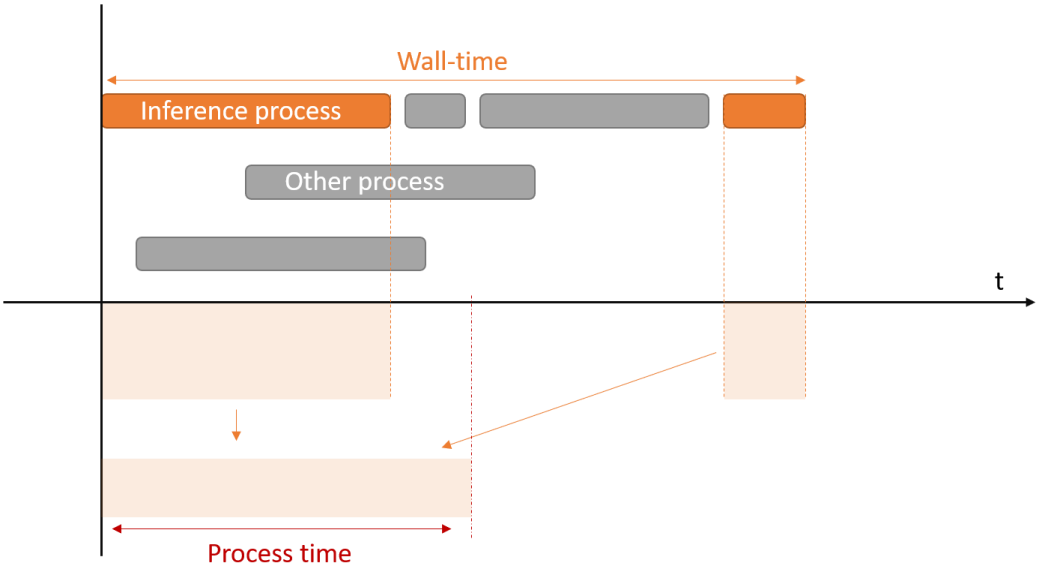

Figure 8 : Process time schema

However, when ML models inference runs on GPU targets, the GPUs take care of the majority of processing task. So, this implies the following challenge. There are a lot of interactions between CPU process allocation and GPU process execution. By measuring the process time, there is a significant risk to skip the GPU calculation time, which is the most important time of inference execution. Moreover, it appears that processing of the inference operation is parallelized between several CPU cores (see Figure 9). Then we compare wall time and process time for the same model (see **Erreur ! Source du renvoi introuvable.**)

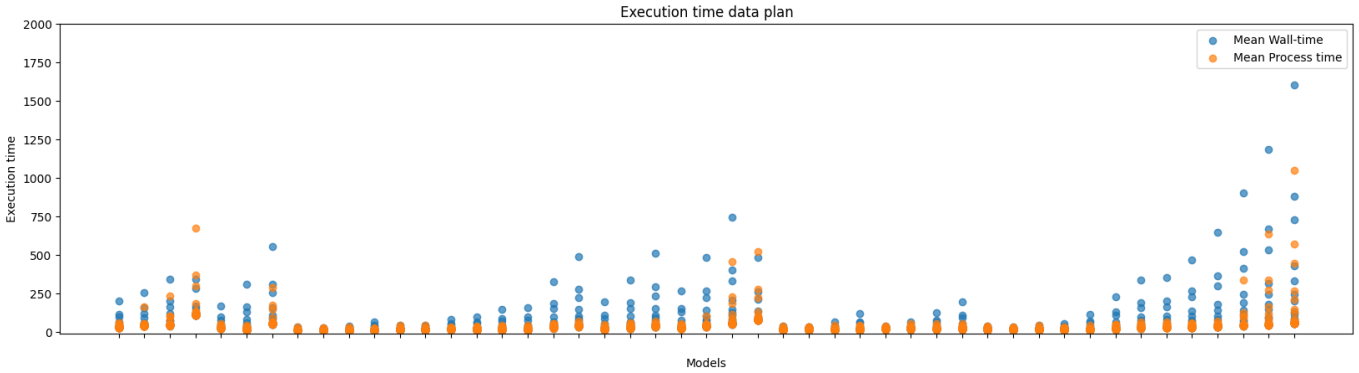

Figure 9: Inference wall-time vs process time

The **Erreur ! Source du renvoi introuvable.** highlights a significant execution time difference between the wall-time and the process time. But just from looking at it, it is impossible to draw conclusions: does process time only remove interruption time from background tasks' system, or does it also skip GPU calculation time. To still push toward a better understanding and precision of the time measurement, we decided to explore Nvidia tools seeking answers.

NVIDIA tool measurement

Working on GPU targets, it is possible to take advantage of the profiling tool provided by Nvidia: NVIDIA Nsight Systems, commonly named Nsys

NVIDIA Nsight Systems (Nsys) is a tool for monitoring specified processes with an end goal to provide a better understanding of all used resources during the sequence. Nsys provides reports in texts or timeline format

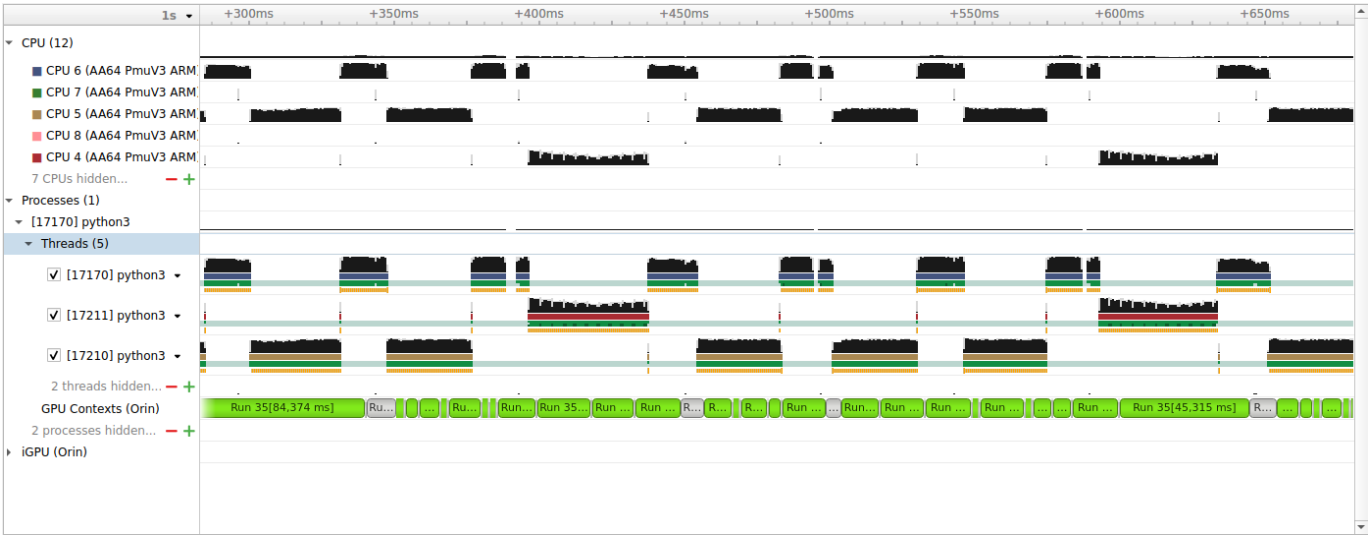

Figure 9 : Nsys timeline of one inference

Figure 9 features the Nsys timeline report of the inference of a DenseNet-121 model. The first thing to observe is that the inference is parallelized between multiple CPU cores. For instance: the process 17170-python3 (the inference) is parallelized between 3 threads using CPU 4, CPU 5 and CPU 6.

Measuring the process time in this specific case is much more complex than the optimal case (presented at **Erreur ! Source du renvoi introuvable.**)

Initially the idea of using Nsys was to have a better understanding of the relation between the CPU and the GPU to increase the precision of time measurement. Figure 9 shows CPU and GPU activities (in green), however, the exact relation between both is still quite fuzzy. Theoretically one GPU context (which represents the activity of a specific process) is linked to a CPU call stack. Nevertheless, the timeline observation at least highlighted one piece of information: CPU call stacks and GPU starting contexts are not synchronous. Indeed, in a scenario where a CPU call occurs while the GPU is busy with another context, it appears that the relevant GPU context won't start until the previous one ends. This asynchronous way of working makes precise time measurement quite challenging. Linking a CPU call stack to the start of GPU context becomes a very tedious task and more importantly, is hard to automate.

With current elements, Nsys does not offer a suitable solution for improving the understanding and precision of time measurement.

Conclusion

For various reasons, none of these solutions has proven to be truly conclusive in terms of improving measurement accuracy, or at the very least, control:

- The wall-time measurement considers the physical time of the inference (the elapsed time), but includes all potential disturbances that may occur during inference execution; interruptions, preemptions, etc…

- The process time measurement only considers the CPU time. It could have been good enough, but Nvidia tools highlighted the parallelism of the task. In this case, the process time measurement does not reflect the physical time.
- Nvidia tools measurement where not flexible enough to take an easy systematic time measurement for each inference

To precisely address this issue, we would need to identify the dependencies between tasks. This will help determine if the delay induced by a preemption on one of the threads of the application being measured affects the inference time (by inducing delays). To do this, we would need to establish the task dependency graph to identify which tasks belong to the critical path after a certain delay.

Finally, for this first phase, the overall best solution seems to measure physical time (the wall-time then), trying to reduce disturbances as far as possible. Multiple actions have been taken for this part:

- Control the priority level, and ensure that the desired thread has the highest possible priority
- Disable the Dynamic Voltage and Frequency Scaling (DVFS) which is a technique used to improve energy efficiency of the GPU
- Enable the persistence daemon that provides the kernel to initialize and deinitialize the target GPU each time a target GPU application starts and stops

Once disruptions have been reduced as much as possible, measurement conditions and environment should be precisely defined to ensure reproducibility. Therefore, it is reasonable to consider that future predictions, that should be done in the same environment, should be representative of real measure.

Finally, with a better understanding, removing proven outliers should also increase the quality of the measurements, and consequently, the accuracy of future predictions.

D.4 Ram usage measurement

For the RAM usage two methods of measurement have been identified:

- System tool: free
- Tensorflow tool: tf.config.experimental.get_memory_info

Free

> **free** displays the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel.
>
> *linux.die.net*

The first idea to monitor RAM usage was to probe periodically (every 5 seconds for instance) the system memory during an inference. Doing so, we made a first observation: the system automatically uses swap memory when RAM reaches ~80% of total capacity.

The swap is a "virtual" physical memory, often installed in Linux in an independent partition. Part of the hard disk is reserved for this virtual memory, which will relieve the system in the event of overload.

In this context where the end goal is to control the process from start to finish, using swap is unsuitable. It can affect RAM and time measurement in an unpredictable way. Consequently, dealing with this issue was mandatory.

The simplest way to prevent the system from using swap memory was to disable it using "**`swapoff -a`**" Linux command. However, by doing so, we have seen an increase in the instability of systems with limited RAM memory initially. For instance, the Jetson Nano froze more often with the swap off, when an inference required just a little less RAM than available. This behavior also has a significant impact on both RAM and processing time.

The second way to prevent the system to use swap without disabling it involved a combination of multiple commands. Working on a docker container, the first step was to prevent the docker from swapping. The second step was to change the system threshold "**`vm.swappiness`**". This parameter sets the RAM usage value (in percent) at which the system can start swapping. Setting this parameter to 0 will drive the system into using the maximum RAM available before swapping.

Despite this cleaner system environment, RAM measurement from free probing was still a little too fuzzy. Using free as measurement tool means that the whole system is captured. Even if the system is cleaner than before, it is not totally free from any interference.

Consequently, we changed the measurement tool and set our sights on Tensorflow tools.

Tensorflow tool: tf.config.experimental.get_memory_info

> Get memory info for the chosen device, as a dictionary with keys *'current'* and *'peak'*, specifying the current and peak memory usage respectively.
>
> *tensorflow.org*

The usage of this tool was quite straight forward: it returns precisely the value of the max RAM usage by the GPU during the inference. The GPU being responsible for the majority of the computational part of the inference, we have estimated the returned values to be fairly representative and kept them for the final models.

E. Methodology

This section focusses on the description of the methodology used to construct the final model, step-by-step, from the collection of ML models to the training and testing of the final model. These steps can be summarized as follow:

1. Collect a variety of ML models
    a. State-of-the-art models
    b. Synthetic models specifically tailored to our needs
2. Collect data
    a. CNN feature
    b. Time measurement
    c. Ram measurement
3. Modeling
    a. Training data selection
4. Test models

This methodology, first presented by Bouzidi and all, is summarized in the Figure 10 below.

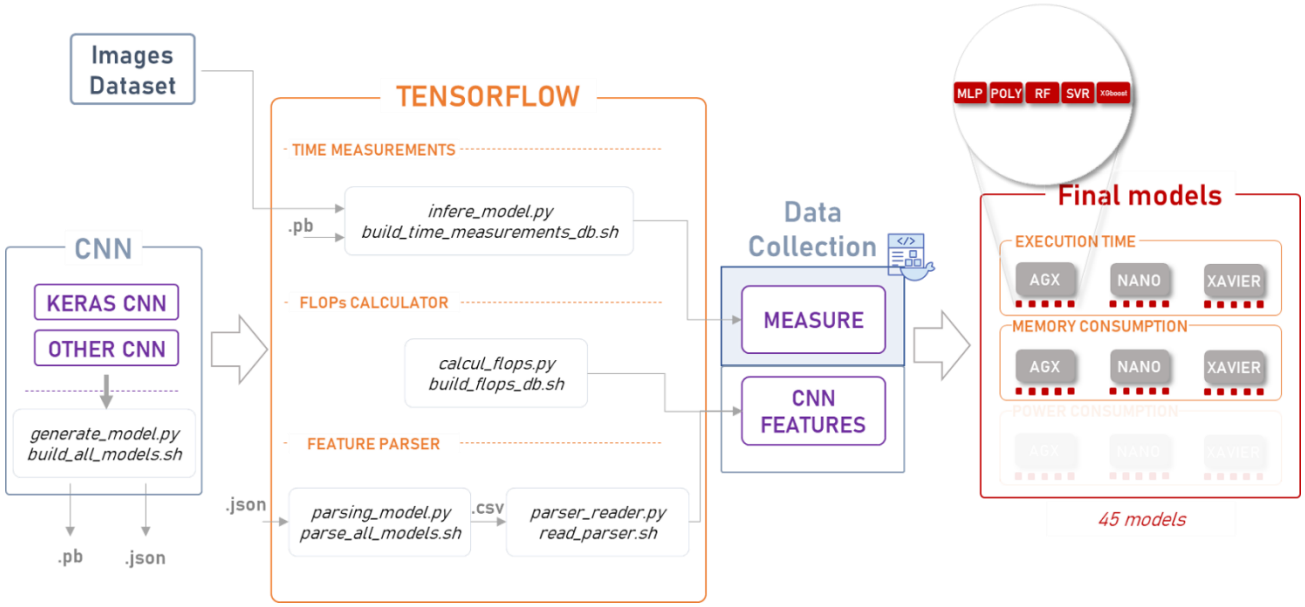

Figure 10 : Methodoloy building blocs

E.1 ML Models collection

The collection of ML models should be done only once the first time. Once the ML model's database is built, it can be easily reused for any other target. The database is made of different models from:

- TensorFlow Hub repository such as DenseNet or ResNet
- Own conception named "basic_model"

Self-designed models are intended to (1) meet specific hardware requirements (for example limited amout of RAM) (2) test the resilience of final model

For the sake of reproducibility, the methodology is very generic and can be applied to various types of ML model. But in this first phase mainly CNN models have been used.

For state-of-the-art ML models, the first step is to load a pre-trained model thanks to various tools like Keras or Tensorflow (we used both of them in our case, but it could be any other tool with the same function). For a proper initialization, the size of the input image will need to be specified when calling this script. Once loaded, the next step is to save the model locally thanks to specific file type, the protobuf file format. This protobuf file will be used later for the actual inference of the model. Finally, the model must also be saved in a descriptive json format. This format outlines the model layer by layer and is useful for the feature collection of the model.

For self-designed models, they are created based on 4 parameters: the number of convolution layer, the size of filters, the number of filters, the number of strides. With the goal to design unseen models, the combination of these 4 parameters is unique for each model and chosen randomly within a predefined range of possible values per parameter. Impossible combinations are automatically discarded by design. All generated "basic_model_convLayer_filterSize_numFilters_numStrides" follow then the same procedure as state-of-the-art models: locally saved in protobuf and json format.

At the end of this first step, a csv file is generated with the name, the input size and the location of all treated models. This file constitutes the model database which is available for performance measurement as described in the next step of the methodology.

E.2 Data collection

As a reminder, the ultimate objective is to create a performance prediction model. In this step we gather all relevant data for training and testing the final prediction model. The ML models features will serve as inputs for the model, while the execution time/RAM usage will be the values to predict, based on benchmarking.

Model features extraction

Model features are extracted from the model file descriptor in json format, produced in the previous step, thanks to an parser. Since standard and value than in **Source du** range. Note that features are

CNN feature	Range
# of hidden layers	[10 – 4072]
# of CONV layers	[4 – 1825]
# of BN layers	[0 – 265]
# of FC layers	[0 – 18]
# of filters per CONV layer	[3 – 2688]
# of units per FC layer	[0 – 5625x10^3]
Filter size	[(1x1) – (11x11)]
Input size	[32 – 2400]
# of CONV layers parameters	[0.27x10^6 – 87.1x10^6]
# of BN layers parameters	[0 – 0.8x10^6]
# of FC layers parameters	[0 – 119.8x10^6]
Total number of FLOPs	[3.1x10^6 – 5.2x10^{12}]
Sum of intermediate activations	[0.1x10^6 - 37.6x10^9]

Figure 11 : Details of ML features

updated version of Bouzidi's state-of-the-art ML models are identical, we retrieved the same Bouzidi's work, and **Erreur ! renvoi introuvable.** shows their for self-designed ML models, acquired differently.

Benchmarking

All ML models are loaded into a specific target via ssh or external drive. From there a specific docker environment is created with the following parameters:

Docker
Image : l4t-tensorflow:r32.7.1 -tf2.7-py3
Mémoire RAM: (depending on target)
Mémoire SWAP: 0 GB
TensorFlow : 2.7.0

During inference the execution time and the RAM usage are measured following the best appropriate method, described in the Data section. To ensure a representative measurement, each model is inferred 250 times. The first 50 measures are automatically discarded to avoid the warmup phase, which can alter the time measurements. As illustrated in **Erreur ! Source du renvoi introuvable.**, we observe that measurements follow a normal distribution. We can therefore consider the mean as a representative tendency, just as H.Bouzidi did in her research. At the end, for each model, the max. time, the mean time, the min. time and the peak RAM usage are saved in a database, used next for the final model modeling.

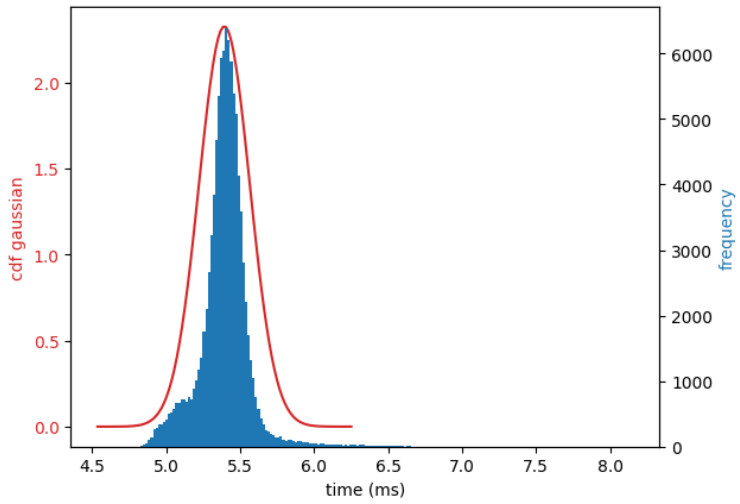

Figure 12 : Time measurements histogram

E.3 Modeling

The final models are based on the prediction algorithms below, elaborated further in Bouzidi's paper:

- Ridge Polynomial Regression (Poly)
- Support Vector Regression (SVR)
- Multi-Layer Perceptrons (MLP)
- Random Forest (RF)
- eXtreme Gradient Boosting (XGBoost)

To optimize the accuracy of final predictions, a specific dataset strategy has been defined in Bouzidi's work. For instance, 3 specific test domains have been designed to evaluate the performance of final models following different scenarios.

In our case, the objective is not to evaluate the best algorithm for the final model, but simply to implement the final model with the best accuracy. Moreover, depending on the target used for measurement, the produced database is different (for example, there is more basic_model in the Nano database, because of RAM limitations). Consequently, we used another methodology.

We considered that the most general approach to find the best overall prediction algorithm with different input databases was to randomize training and testing datasets. So, with a defined ratio of 80%, meaning that the training dataset represents 80% of initial database (and the testing dataset represents the remaining 20%) each entry of the database is assigned randomly in training or testing dataset. Once the datasets have been assembled, we provide them to the 5 prediction algorithms. This process is performed 10'000 times, each time having a different random seed. The seed that produced the best accuracy overall is saved for further reproducibility.

Note that the accuracy is calculated following the MAPE method described in Bouzidi's paper.

E.4 Results

The results of all experiments are summarized in the following tab:

	Number of inputs	Time prediction		RAM Usage prediction	
		Algorithm	Accuracy	Algorithm	Accuracy
Nano	1016	SVR	92.44 %	SVR	91.12 %
Orin	1159	XGB	88.09 %	SVR	96.12 %
Halima Nano	1612	XGB	92.01 %	Poly	93.57 %

Table 1 : Predictive mechanism results on GPU targets

F. Conclusion and future work

In conclusion, the exploration and analysis conducted in this document shed light on the landscape of trustworthiness in AI within critical systems. The challenges outlined, ranging from the design and characterization of AI models to their implementation on hardware, present hurdles that demand comprehensive solutions.

This document has undertaken a comprehensive exploration of the performance prediction mechanism within the ML benchmarking environment on GPUs, focusing specifically on Phase 1 which is encapsulated within the red ellipse in Figure 1. We started by acknowledging the work of researchers at Valenciennes University, specifically H. Bouzidi and S. Niar, who laid the foundation for our performance prediction approach. Then, we dived into the details of the data we use, explaining why we chose it and how we collect it.

In pursuit of our objectives, we made several adjustments to attain the results showcased in Table 1. Primarily, we revised the tools employed to enhance the sustainability of the predictive mechanism over time. Additionally, to align with the imperative of trustworthiness in AI within critical systems, we implemented modifications to the experimental configurations. Consequently, the acquired data deviated from its exact replication, necessitating adaptations in the methodology to align with our specific requirements.

These modifications enabled us to attain results that (1) closely align with the support offered by Bouzidi and colleagues, and (2) adequately meet our requirements.

As we look ahead, Phase 2 beckons, a stage where the goal is to design a generalized model. The ambition is lofty: to extend the performance prediction mechanism to estimate the performance of any ML model across all embedded targets within the ML-bench. This broader vision aligns with our commitment to scalability, adaptability, and comprehensive utility. To further advance in this trajectory, phase 2 will also tackle the outstanding issues highlighted in this document, particularly focusing on improving the precision and confidence associated with time measurements.

G. Bibliography

[Bouzidi, 2022] Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. 2022. Performances Modeling of Computer Vision-based CNN on Edge GPUs. J. ACM XX, X, Article XXX (March 2022), 33 pages. https://doi.org/10.1145/1122445.1122456

[Amaris, 2016] Marcos Amarís, Raphael Y. de Camargo, Mohamed Dyab, Alfredo Goldman, and Denis Trystram. 2016. A comparison of GPU execution time prediction using machine learning and analytical modeling. In 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA) (Cambridge, MA, USA). 326–333. https://doi.org/10.1109/NCA.2016.7778637

[Wang, 2019] Mengdi Wang, Chen Meng, Guoping Long, ChuanWu, Jun Yang,Wei Lin, and Yangqing Jia. 2019. Characterizing Deep Learning Training Workloads on Alibaba-PAI. In 2019 IEEE International Symposium on Workload Characterization (IISWC) (Orlando, FL, USA). 189–202. https://doi.org/10.1109/IISWC47752.2019.9042047

[Shi, 2018] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. 2018. Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs. In 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech) (Athens, Greece). 949–957. https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.000-4

[Mu, 2020] Jiandong Mu, Wei Zhang, Hao Liang, and Sharad Sinha. 2020. Optimizing OpenCL-Based CNN Design on FPGA with Comprehensive Design Space Exploration and Collaborative Performance Modeling. 13, 3, Article 13 (jun 2020), 28 pages. https://doi.org/10.1145/3397514

[Zhang, 2020] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2020. DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-Based DNN Accelerator (ICCAD'20). Association for Computing Machinery, New York, NY, USA, Article 61, 9 pages. https://doi.org/10.1145/3400302.3415609

[Stamoulis, 2018] Dimitrios Stamoulis, Ermao Cai, Da-Cheng Juan, and Diana Marculescu. 2018. HyperPower: Power- and memory constrained hyper-parameter optimization for neural networks. In 2018 Design, Automation Test in Europe Conference Exhibition (DATE) (Dresden, Germany). 19–24. https://doi.org/10.23919/DATE.2018.8341973

Title : Performance prediction definition document

Keywords : Performance evaluation bench, machine learning, performance prediction

The performance evaluation bench developed as part of the Confiance.ai program provides an industrial user with a set of software tools and hardware cards to implement and assess the performance of ML models in different deployment configurations.

To enable these evaluations quickly and in cases where the hardware targets are not available – either because they are already in use, not installed in the bench, or simply not yet on the market – obtaining initial performance estimates based on Machine Learning prediction models can be beneficial.

This document presents the implementation and initial results of applying this technique, specifically to NVidia's Jetson targets.

Titre : Performance prediction definition document

Mots clefs : Banc d'évaluation de performance, machine learning, prediction de performances

Le banc d'évaluation de performance déveoppé dans le cadre du programme Confiance.ai met à disposition d'une utilisateur industriel un ensemble d'outils logiciels et de cartes matérielles lui permettant d'implémenter et d'évaluer les performances de modèles ML dans différentes configurations de déploiement.

Afin de permettre de réaliser ces évaluations de façon rapide et dans le cas où les cibles matérielles ne sont pas disponibles – parce qu'elles sont déjà utilisées, non installées dans le banc, ou tout simplement non encore commercialisées –, il peut être intéressant d'obtenir de premières estimations de performance sur la base de modèles de prediction basés sur des techniques de Machine Learning.

Ce document présente la mise en oeuvre et les premiers résultats d'application de cette technique, appliquée aux cibles Jetson de NVIdia.

Our partners